5. Defining Classes and Methods

[ITP20003] Java Programming

Agenda

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

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 like person and book,
  - or Abstractions like shape and color

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.

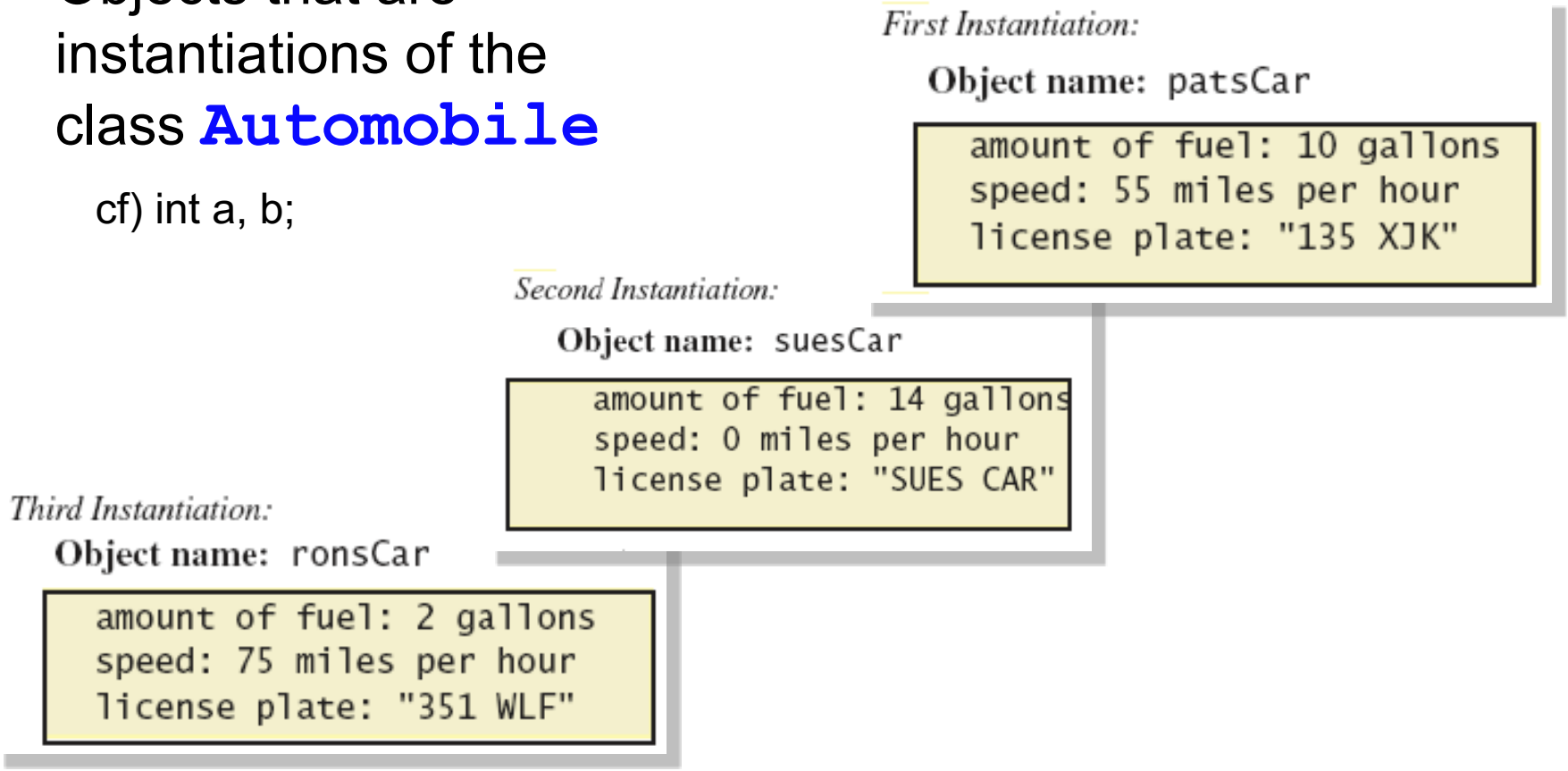
add more data items if you want
e.g., color, year

add more actions if you want
e.g., change gear, turn

Class and Method Definitions

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

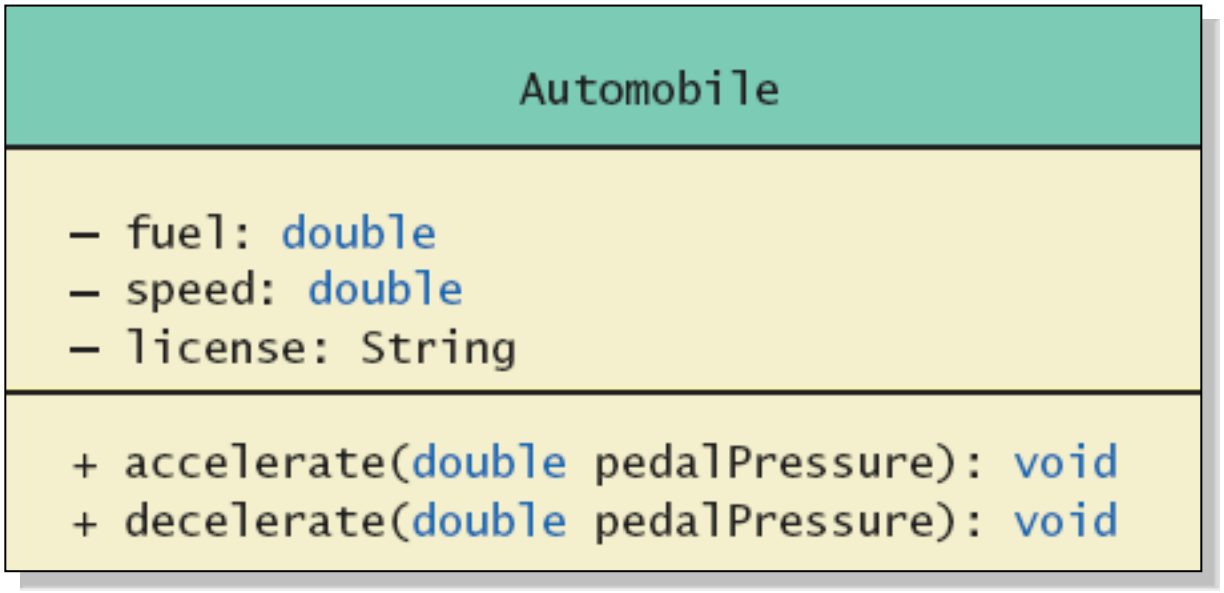
cf) int a, b;



Class and Method Definitions

- A class outline as a UML class diagram

  (UML: Universal Modeling Language)



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 (a.java → a.class)
  - Helpful to keep all class files used by a program in the same directory
  - will discuss how to use files from more than one directory in chapter 6.

Dog class and Instance Variables

```
public class Dog {          // see Listing 5.1
   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;
   }
}
```

violates several important design principles

Dog class and Instance Variables

- View sample program, listing 5.1
- 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 (in DogDemo.java)

```
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.
```

DogDemo (in DogDemo.java)

- In Java, new is a unary operator that we use to create objects of a class.

```
Dog scooby =new Dog();
```

- it creates an object of the class Dog.
- The new operator then returns the memory address of the object.
- An object can have variables inside of it, namely, the instance variables of the object.
- The new operator places these instance variables inside of the object when it creates the object.

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
    - 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

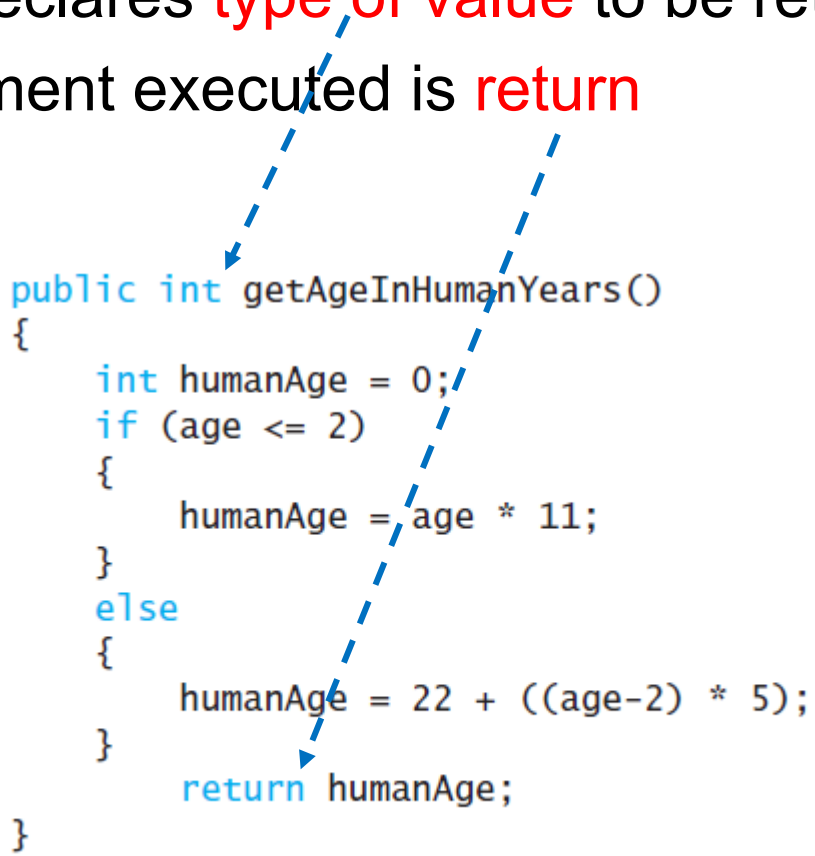
```
public void writeOutput()
{
    System.out.println("Name: " + name);
    System.out.println("Breed: " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();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

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;
}
```

Second Example – Species Class

- Class designed to hold records of endangered species
- View the class listing 5.3
  `class SpeciesFirstTry`
  - Three instance variables, three methods
  - Will expand this class in the rest of the chapter
- View demo class listing 5.4
  `class SpeciesFirstTryDemo`

Second Example – Species

```
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();
        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 + "%");
    }
```

bad example

public -> private

Second Example – Species

```
    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;
    }
}
```

Second Example – Species

```
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());
    }
}
```

Second Example – Species

- 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

  Syntax) *ObjectName*.*VariableName*

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

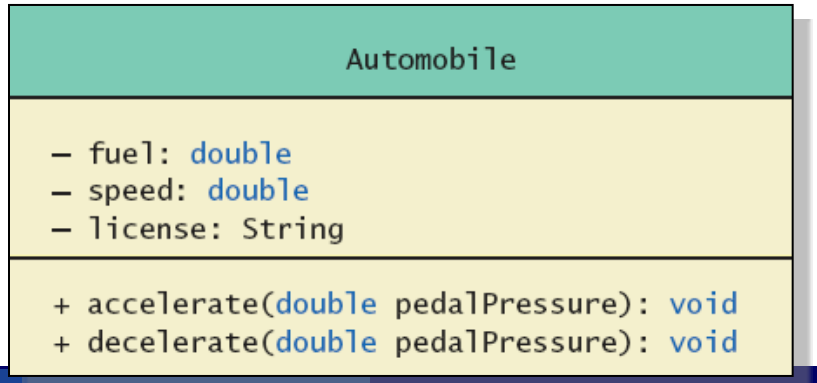
- Inside the class the unnamed object can be referred to with the name *this*

  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(…) {
                ...
        }
        …
}
```



- 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* (or parameter)

- Calling the method

  int futurePopulation = speciesOfTheMonth.predictPopulation(10);

  - The actual parameter is the integer 10 (or argument)

- View sample program, listing 5.7
  class SpeciesSecondClassDemo

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
- Graphics Supplement

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*

```
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.>
}
```

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;
    }
}
```

```
Rectangle box =new Rectangle( );
```

➔ 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(): 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 (or getter)
- Must also provide methods to change the values of the private instance variable
  - Typically named *setSomeValue()*
  - Referred to as a mutator method (or setter)

Accessor and Mutator Methods

- Consider an example class with accessor and mutator methods
  - View sample code, listing 5.11
  - Note the mutator method
    - setSpecies()
  - Note accessor methods
    - getName(), getPopulation(), getGrowthRate()

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;
    }
```

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

Accessor and Mutator Methods

```
    }
    public String getName()
    {
        return name;
    }
    public int getPopulation()
    {
        return population;
    }
    public double getGrowthRate()
    {
        return growthRate;
    }
}
```

Why do we use setter and getter instead of 'public' variable?

Accessor and Mutator Methods

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

```
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
```

Accessor and Mutator Methods

```
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();
        ...
```

Programming Example

- View sample code, listing 5.13, class Purchase
  - Note use of private instance variables
  - Note also how mutator methods check for invalid values

```
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 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;
    }
```

```
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 <=
    { //Try again:
        System.out.println("Both numbers must
                           "be positive. Try a
        System.out.println("Enter price of " +
                           "item as two number
        System.out.println("For example, 3 for
                           "$2.99 is entered a
```

```
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;
}
}
```

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
```

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.

- View sample code, listing 5.15
  class Oracle
  - chat() is *public*, but other methods are *private*
  - chat() calls answer();
  - answer() calls seekAdvice()

- View demo program, listing 5.16
  class OracleDemo
  - main() of OracleDemo class calls the chat() method of Oracle class

```
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();
    }
```

this.answer();

Encapsulation

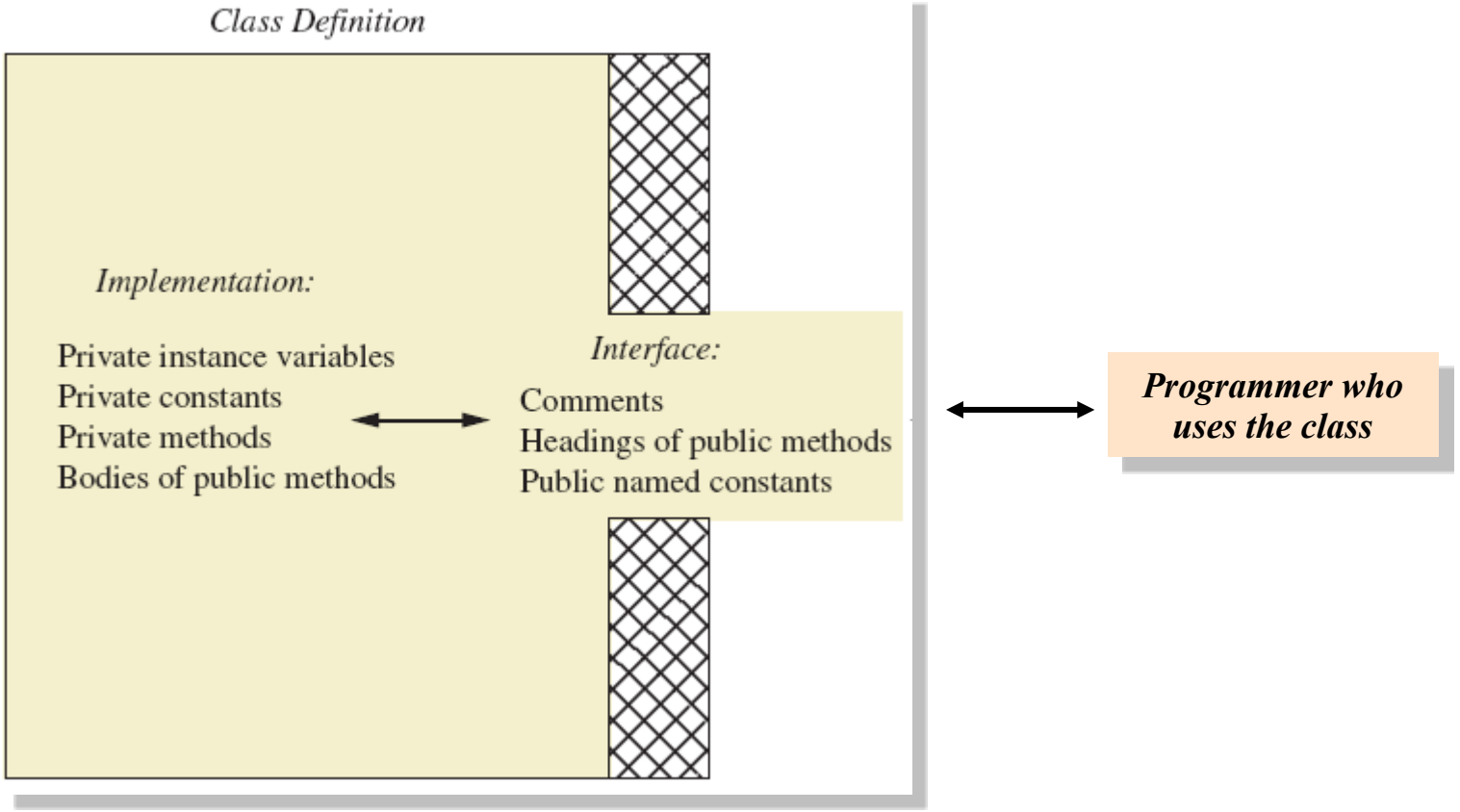
- Encapsulation groups instance variables and methods into a class
- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs
- Encapsulation divides class definition into
  -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

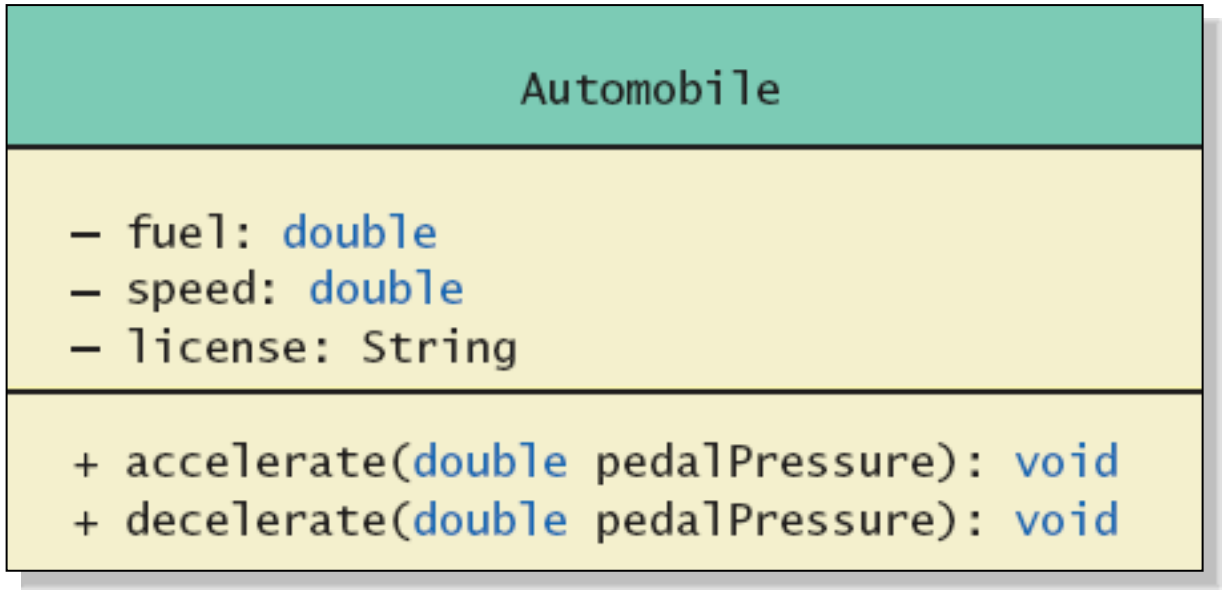
- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data
- Provide public methods manipulating data
  - Such methods could include public mutator methods.
- Place a comment before each public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

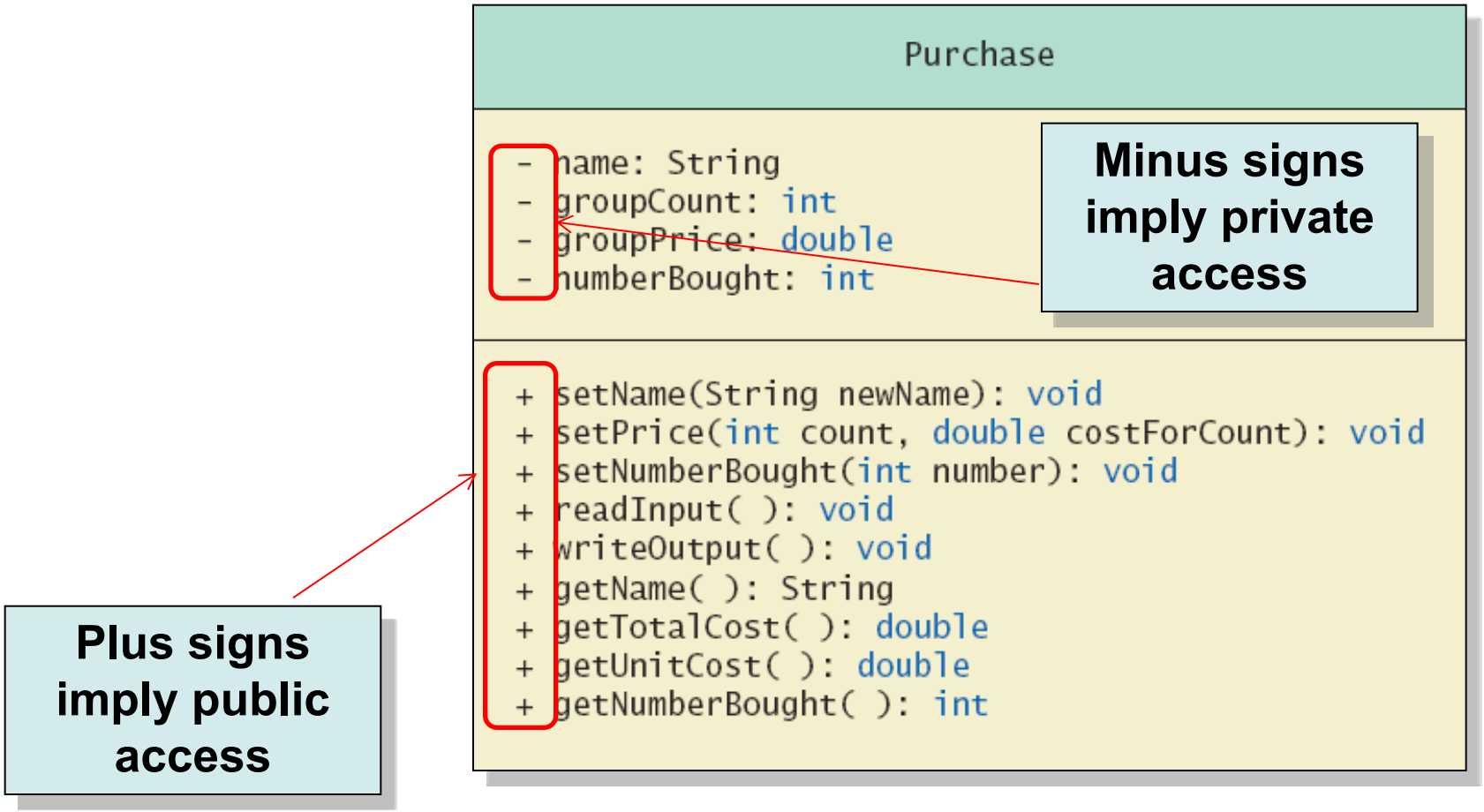
UML Class Diagrams

- A class outline as a UML class diagram



UML Class Diagrams

- The 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

Information hiding vs. encapsulation

- “Encapsulation is the grouping of related ideas into one unit, which can thereafter be referred to by a single name.” - Meilir Page-Jones
  - Ex. function, object
- information hiding is achieved by Encapsulation
  - Ex. interface

Agenda

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

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

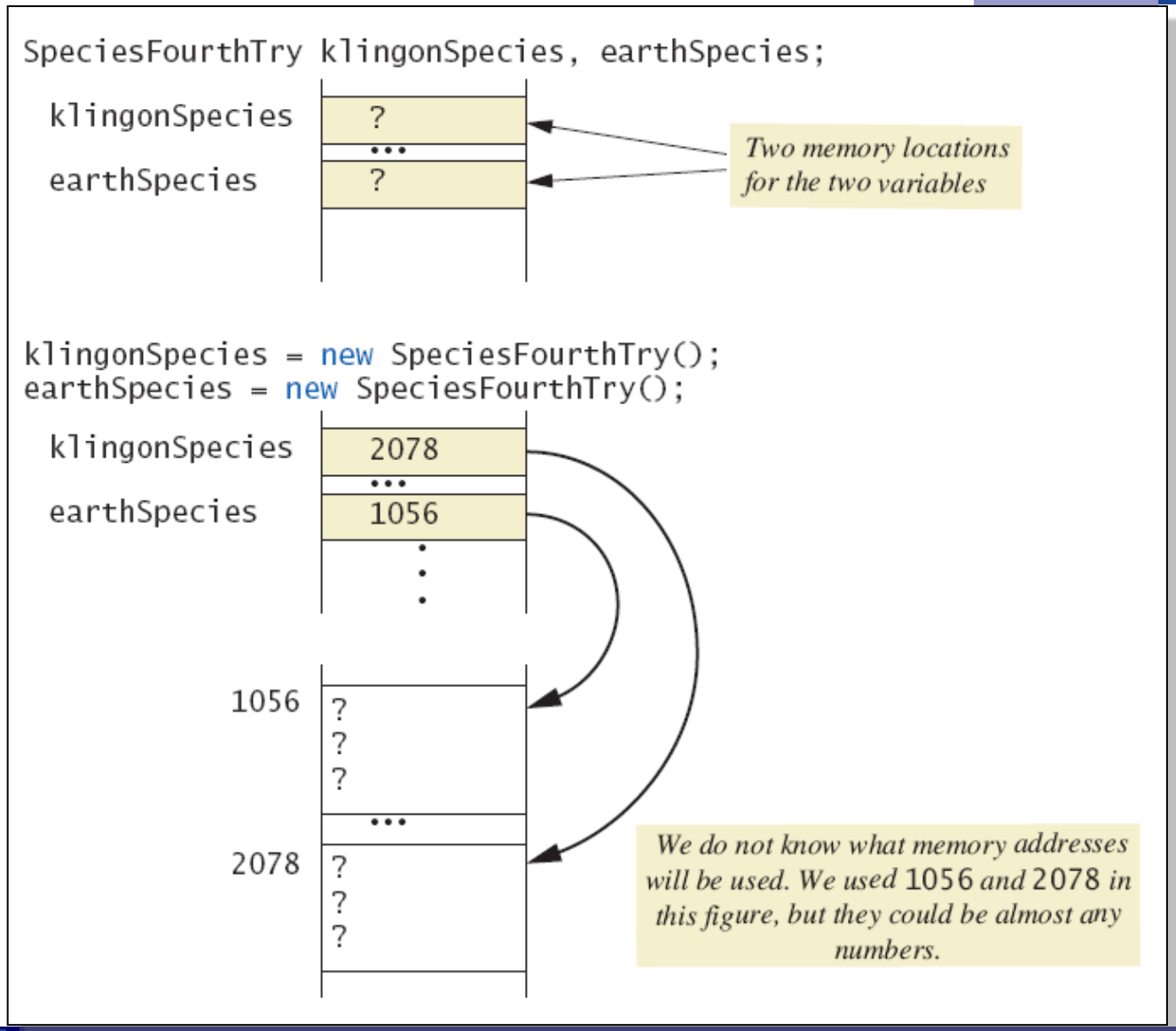
  Ex) MyClass obj = new MyClass();

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

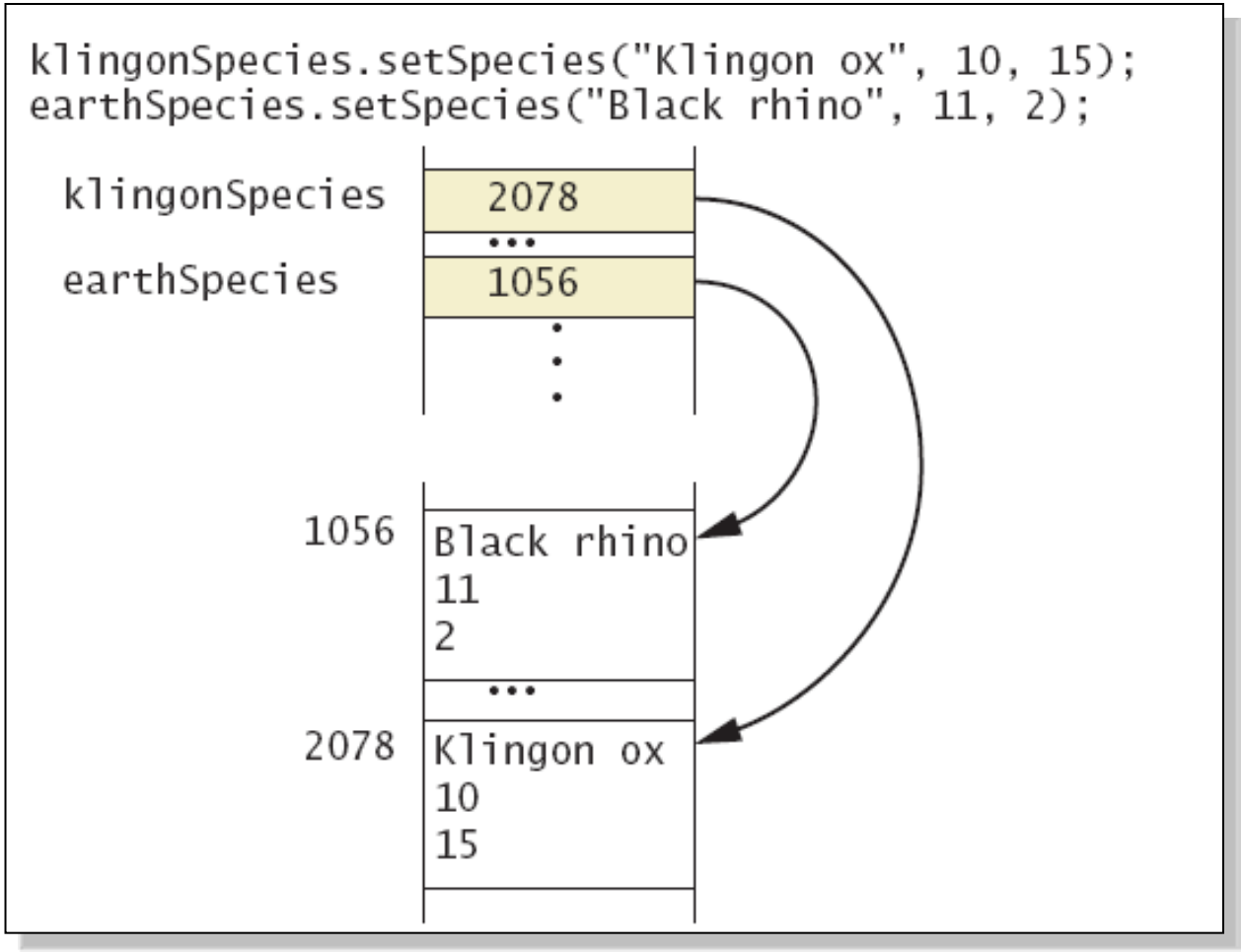
Variables of a Class Type

- Behavior of class variables



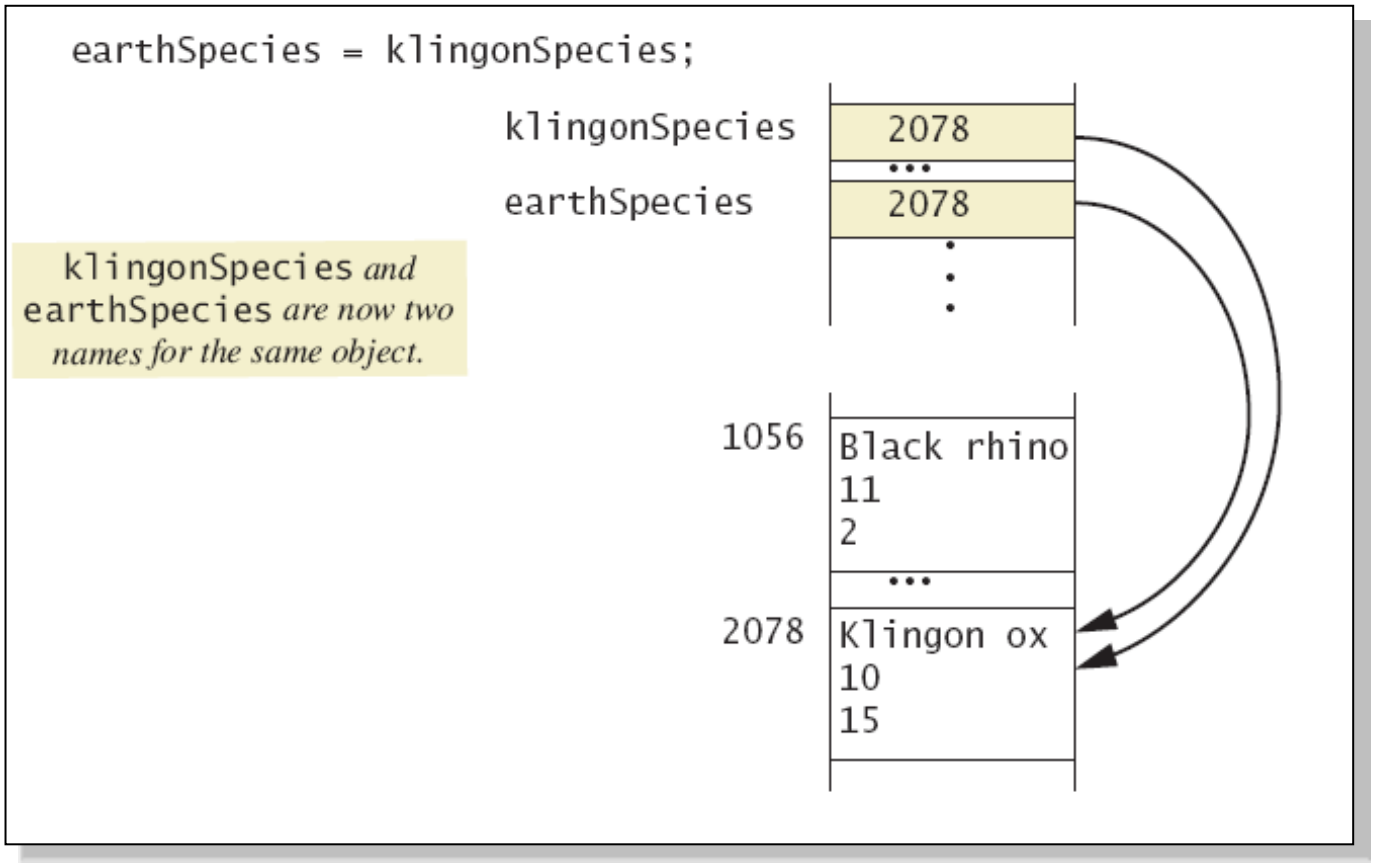
Variables of a Class Type

- Behavior of class variables



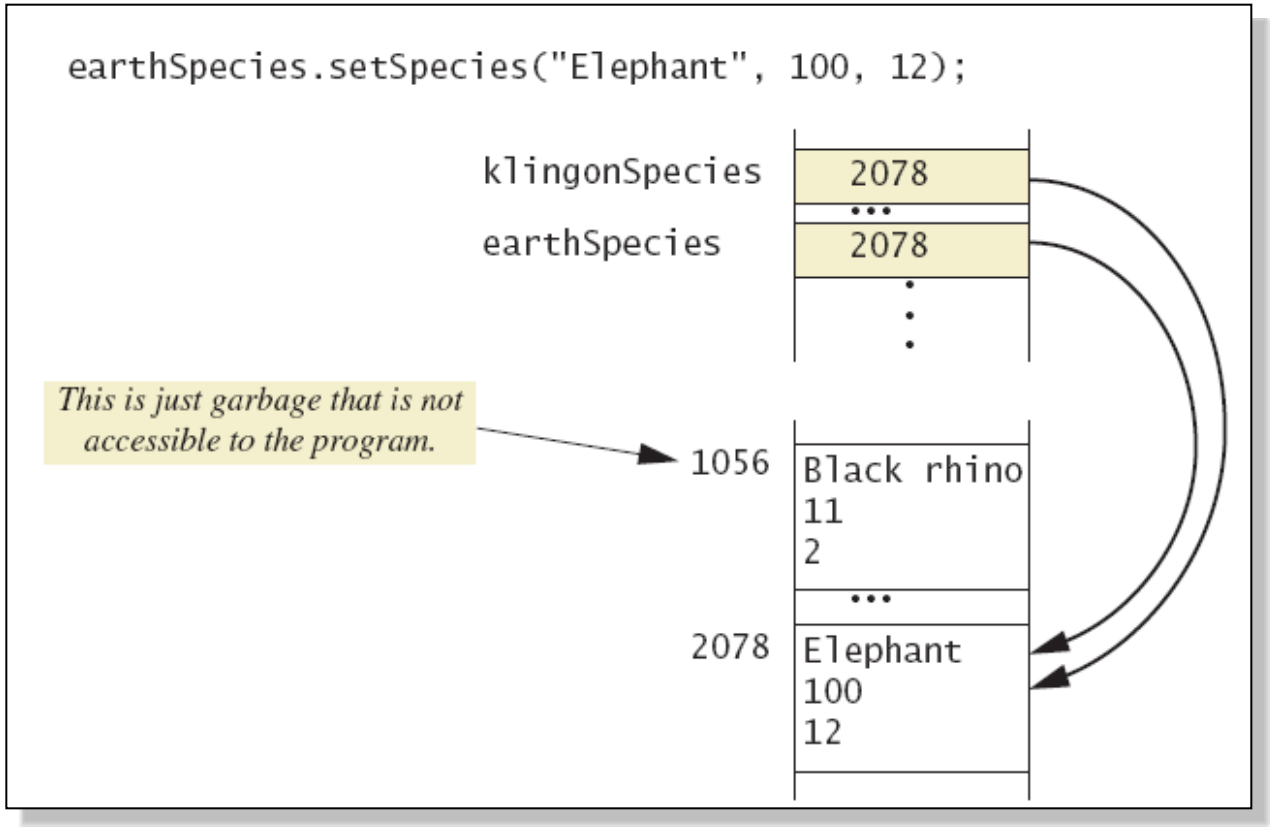
Variables of a Class Type

- Behavior of class variables



Variables of a Class Type

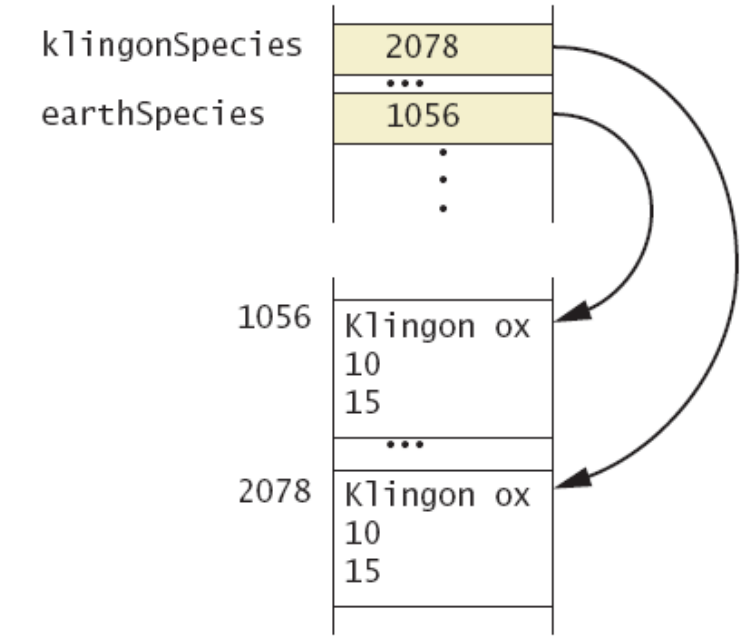
- Behavior of class variables



Variables of a Class Type

- Dangers of using == with objects

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

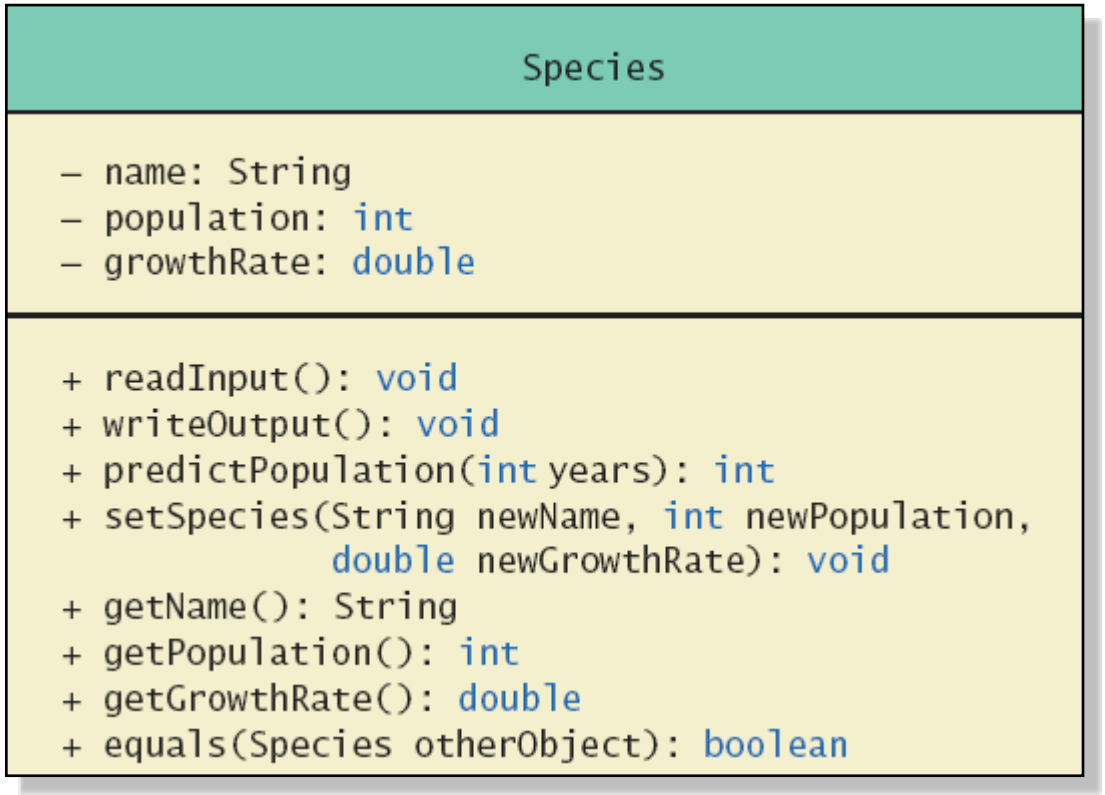
Demonstrating an *equals* Method

- Result

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

Complete Programming Example

- View sample code, listing 5.19
  class Species
- Class Diagram for the class Species



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 to make sure everything still works is regression testing
- View sample code, listing 5.20
  class SpeciesTest

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

- View sample code, listing 5.21
  class DemoSpecies
  - Note different parameter types and results

- View sample program, listing 5.22
  - Parameters of a class type versus parameters of a primitive type
    class ParametersDemo

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, which is unchanged.

```
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%
```

questions or comments?

hchoi@handong.edu