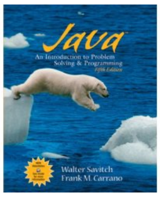
# 5. Defining Classes and Methods

Harald Gall, Prof. Dr.
Institut für Informatik
Universität Zürich
http://seal.ifi.uzh.ch/info1

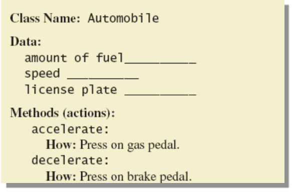


## Objectives

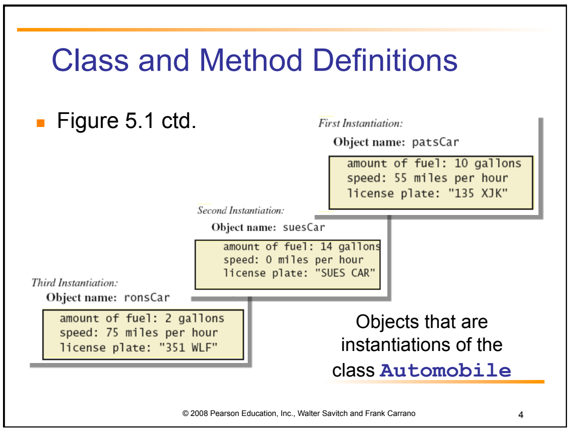
- Describe and define concepts of class, class object
- Describe use of parameters in a method
- Use modifiers `public`, `private`
- Define *accessor*, *mutator* class methods
- Write method *pre- and postconditions*
- Describe purpose of `javadoc`
- Describe references, variables, parameters of a class type

## Class and Method Definitions

- Figure 5.1 A class as a blueprint



```
Class Name: Automobile
Data:
  amount of fuel________
  speed ________
  license plate ________
Methods (actions):
   accelerate:
     How: Press on gas pedal.
   decelerate:
     How: Press on brake pedal.
```



## Class and Method Definitions

- Figure 5.1 ctd.

*First Instantiation:*
**Object name:** patsCar

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

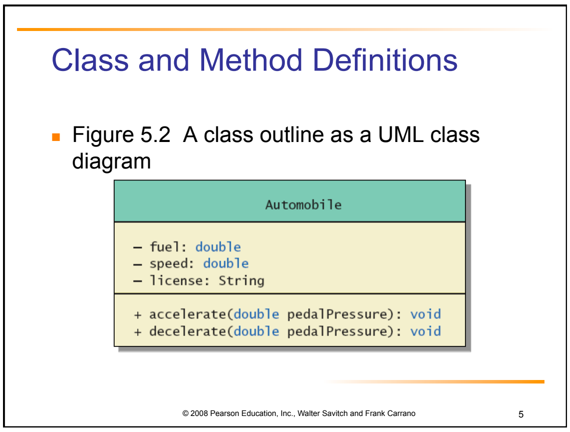
*Second Instantiation:*
**Object name:** suesCar

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

*Third Instantiation:*
**Object name:** ronsCar

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

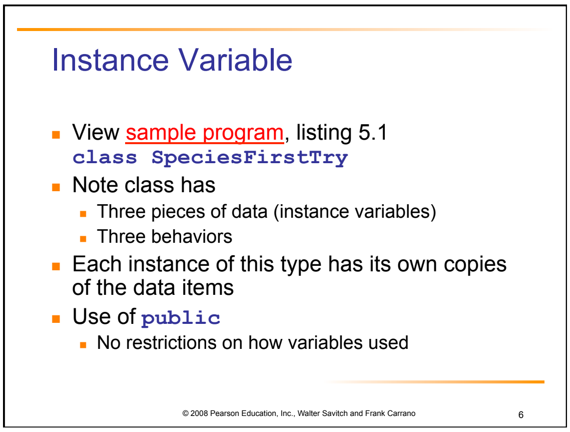
Objects that are instantiations of the class **`Automobile`**



## Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram

```
Automobile

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

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

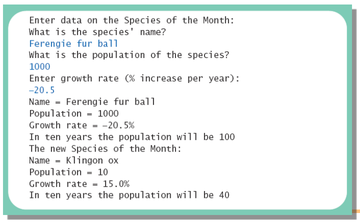


## Instance Variable

- View sample program, listing 5.1 **`class SpeciesFirstTry`**
- Note class has
  - Three pieces of data (instance variables)
  - Three behaviors
- Each instance of this type has its own copies of the data items
- Use of **`public`**
  - No restrictions on how variables used

## Using a Class and Its Methods

- **class SpeciesFirstTryDemo**



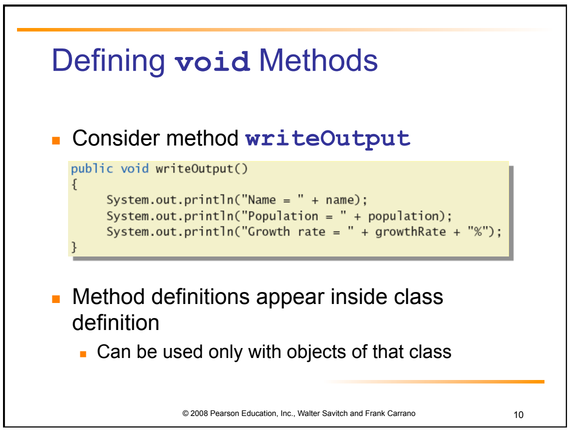
```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (% increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In ten years the population will be 100
The new Species of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

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



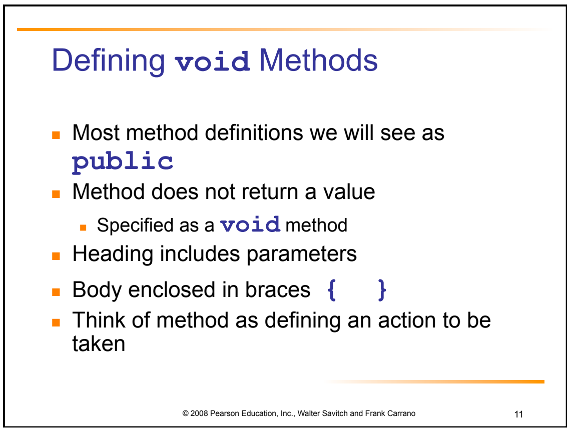
## Defining **void** Methods

- Consider method **writeOutput**

```
public void writeOutput()
{
     System.out.println("Name = " + name);
     System.out.println("Population = " + population);
     System.out.println("Growth rate = " + growthRate + "%");
}
```

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

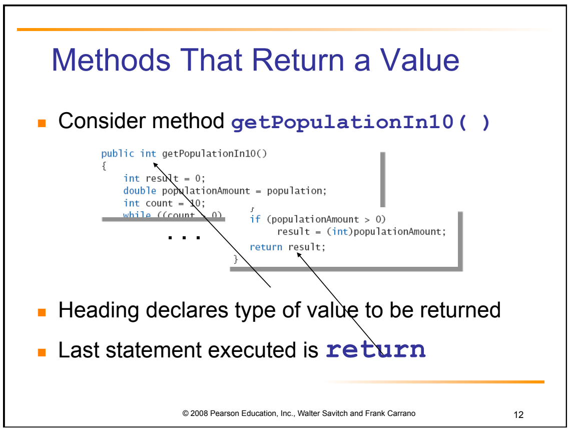
10



## 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

11



## Methods That Return a Value

- Consider method **getPopulationIn10( )**

```
public int getPopulationIn10()
{
    int result = 0;
    double populationAmount = population;
    int count = 10;
    while ((count > 0)
          . . .
        }
        if (populationAmount > 0)
            result = (int)populationAmount;
        return result;
    }
```

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

12

## 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

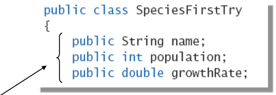
## The Keyword **this**

- 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

- Note beginning of class in listing 5.1

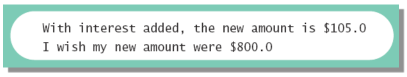


```
public class SpeciesFirstTry
{
    public String name;
    public int population;
    public double growthRate;
```

- Variables declared inside the class are considered *local* variables
  - May be used only inside this class
- Variable with same name inside a different class is considered a different variable
- All variables declared in method **main** are local to **main**

## Local Variables

- `class BankAccount`
- `class LocalVariablesDemoProgram`
- Note two different variables `newAmount`
  - Note different values output

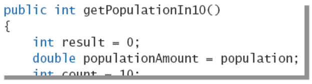


```
With interest added, the new amount is $105.0
I wish my new amount were $800.0
```

## Blocks and scope

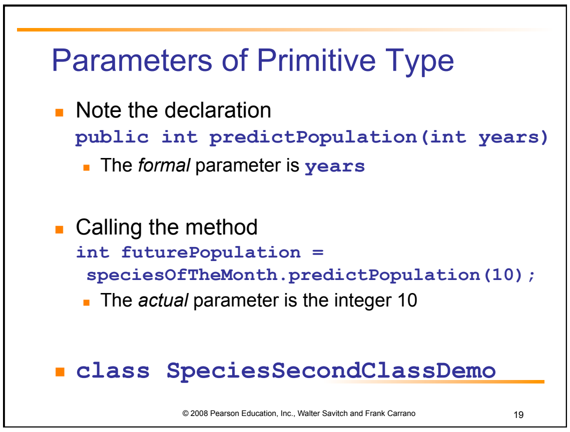
- Recall compound statements
  - Enclosed in braces **{ }**
- When you declare a variable within a compound statement
  - The compound statement is called a *block*
  - The scope of the variable is from its declaration to the end of the block
- Variable declared outside the block usable both outside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

## Parameters of Primitive Type



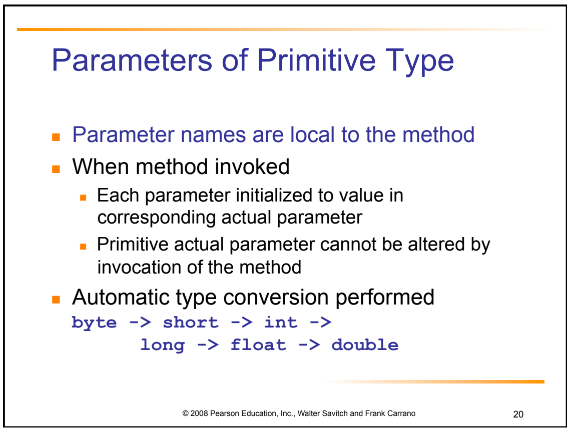
```
public int getPopulationIn10()
{
    int result = 0;
    double populationAmount = population;
    int count = 10;
```

- Recall method declaration in listing 5.1
  - 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`



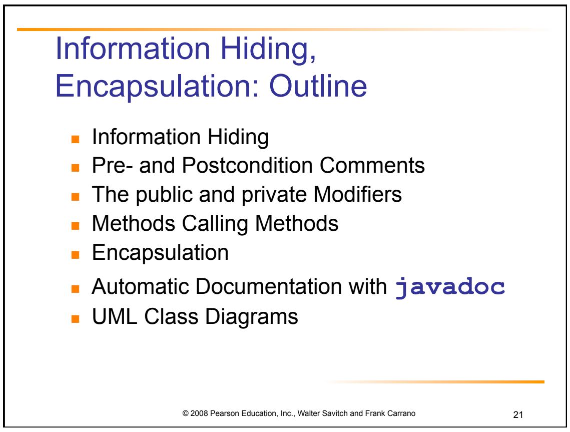
## Parameters of Primitive Type

- 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`



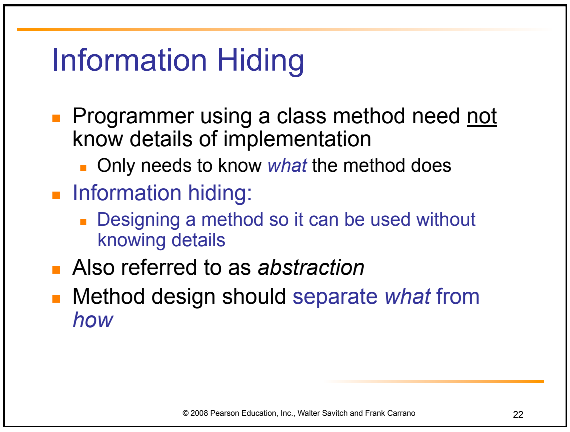
## Parameters of Primitive Type

- Parameter names are local to the method
- When method invoked
  - Each parameter initialized to value in corresponding actual parameter
  - Primitive actual parameter cannot be altered by invocation of the method
- Automatic type conversion performed
  ```
  byte -> short -> int ->
        long -> float -> double
  ```



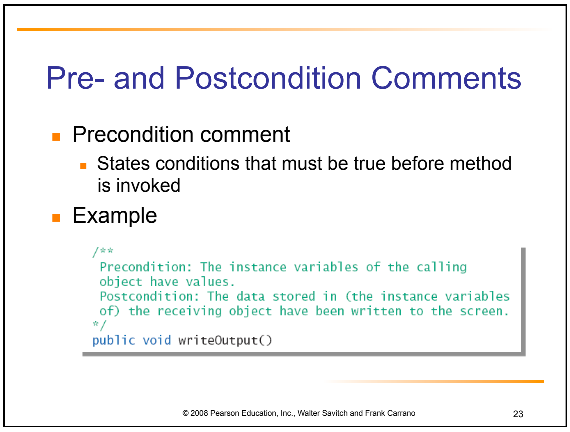
## Information Hiding, Encapsulation: Outline

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



## Information Hiding

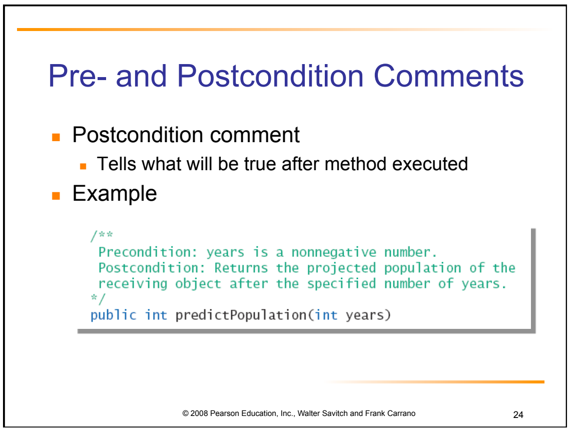
- Programmer using a class method need <u>not</u> know details of implementation
  - Only needs to know *what* the method does
- Information hiding:
  - Designing a method so it can be used without knowing details
- Also referred to as *abstraction*
- Method design should separate *what* from *how*



## Pre- and Postcondition Comments

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

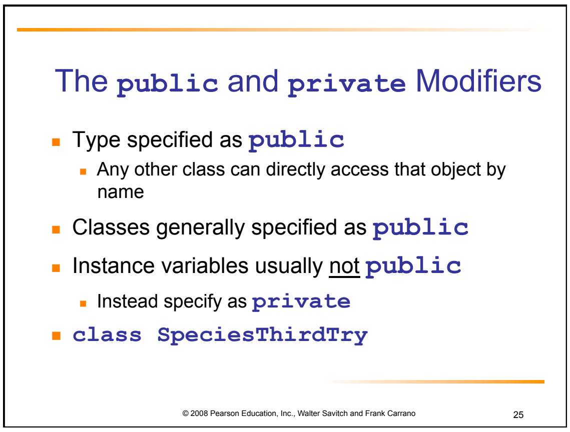
```
/**
 Precondition: The instance variables of the calling
 object have values.
 Postcondition: The data stored in (the instance variables
 of) the receiving object have been written to the screen.
*/
public void writeOutput()
```



## Pre- and Postcondition Comments

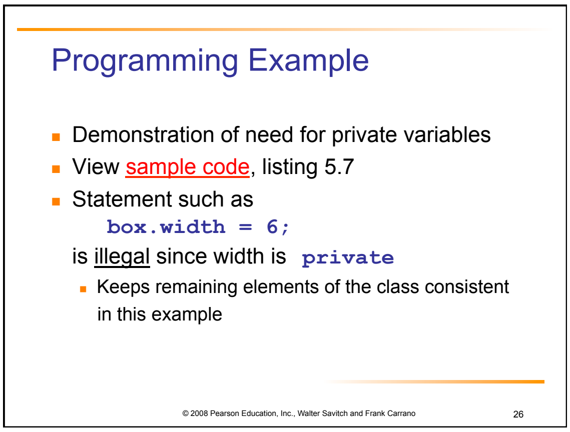
- Postcondition comment
  - Tells what will be true after method executed
- Example

```
/**
 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`

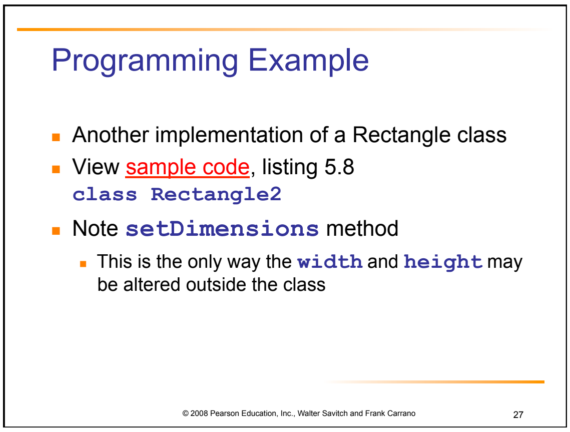


## Programming Example

- Demonstration of need for private variables
- View sample code, listing 5.7
- Statement such as

  `box.width = 6;`

  is illegal since width is `private`
  - Keeps remaining elements of the class consistent in this example



## Programming Example

- Another implementation of a Rectangle class
- View sample code, listing 5.8

  `class Rectangle2`
- Note `setDimensions` method
  - This is the only way the `width` and `height` may be altered outside the class

## Accessor and Mutator Methods

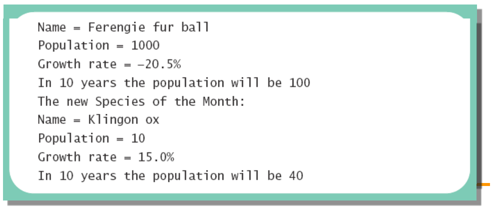
- When instance variables are private must provide methods to access values stored there
  - Typically named `get`*`SomeValue`*
  - Referred to as an accessor method
- Must also provide methods to change the values of the private instance variable
  - Typically named `set`*`SomeValue`*
  - Referred to as a mutator method

## Accessor and Mutator Methods

- Consider an example class with accessor and mutator methods
- View sample code, listing 5.9
  `class SpeciesFourthTry`
- Note the mutator method
  - `setSpecies`
- Note accessor methods
  - `getName`, `getPopulation`, `getGrowthRate`

## Accessor and Mutator Methods

- Using a mutator method
- `classSpeciesFourthTryDemo`

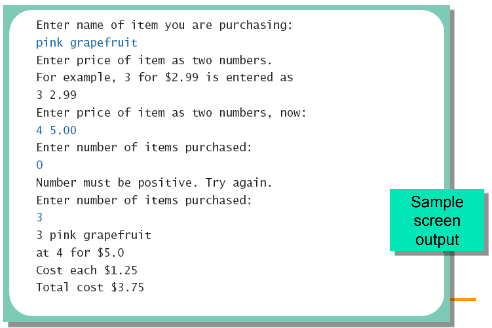


```
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In 10 years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In 10 years the population will be 40
```

## Programming Example

- A Purchase class
- View sample code, listing 5.11
  **`class Purchase`**
  - Note use of private instance variables
  - Note also how mutator methods check for invalid values
- View demo program, listing 5.12
  **`class purchaseDemo`**

## Programming Example



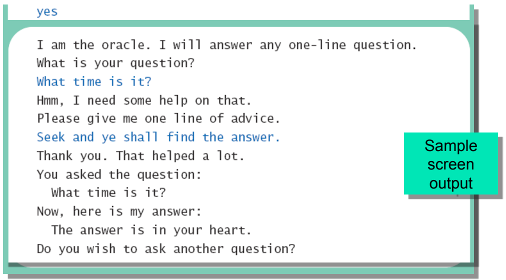
```
Enter name of item you are purchasing:
pink grapefruit
Enter price of item as two numbers.
For example, 3 for $2.99 is entered as
3 2.99
Enter price of item as two numbers, now:
4 5.00
Enter number of items purchased:
0
Number must be positive. Try again.
Enter number of items purchased:
3
3 pink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

Sample screen output

## Methods Calling Methods

- A method body may call any other method
- If the invoked method is within the same class
  - Need not use prefix of receiving object
- View sample code, listing 5.13
  class Oracle
- View demo program, listing 5.14
  class OracleDemo

## Methods Calling Methods



```
yes

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

Sample screen output

## Encapsulation

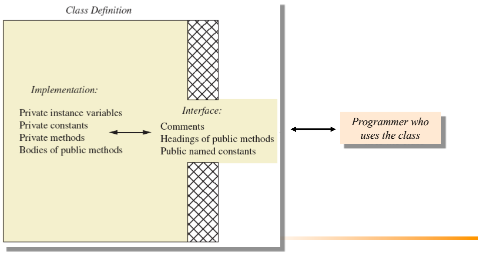
- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs
- Encapsulation divides class definition into
  - Class interface
  - Class implementation

## 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

- Figure 5.3  A well encapsulated class definition



*Class Definition*

*Implementation:*

Private instance variables
Private constants
Private methods
Bodies of public methods

*Interface:*

Comments
Headings of public methods
Public named constants

*Programmer who uses the class*

## Encapsulation

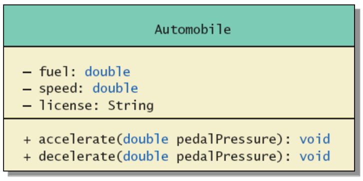
- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private
- Provide public accessor methods to retrieve data Provide public methods manipulating data
  - Such methods could include public mutator methods.
- Place a comment before each public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

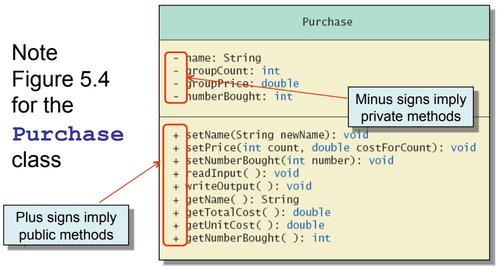
- Recall Figure 5.2  A class outline as a UML class diagram



```
Automobile

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

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

## UML Class Diagrams

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



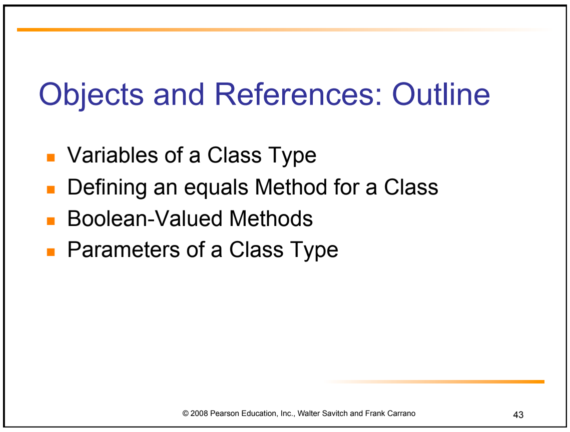
```
Purchase

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

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

Minus signs imply private methods

Plus signs imply public methods

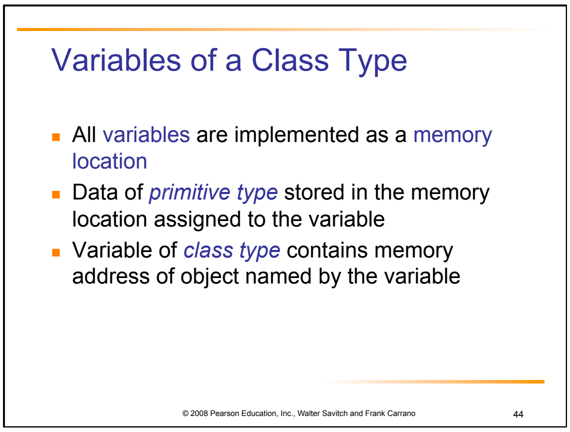
## UML Class Diagrams

- 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



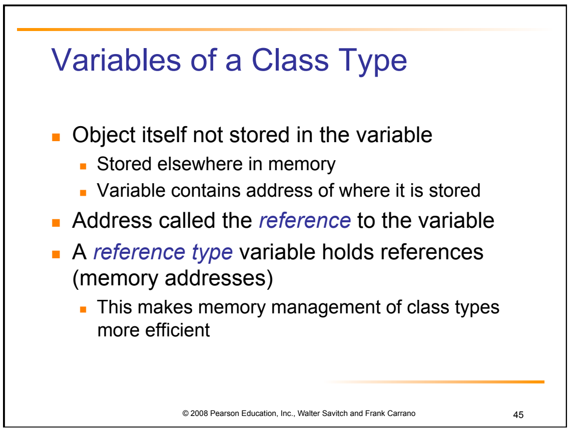
## Objects and References: Outline

- 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

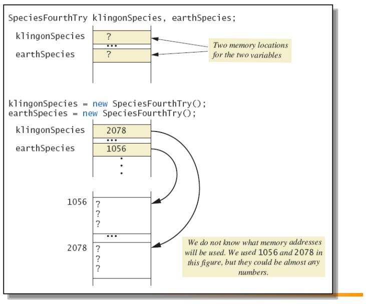


## Variables of a Class Type

- 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



```
SpeciesFourthTry klingonSpecies, earthSpecies;
```

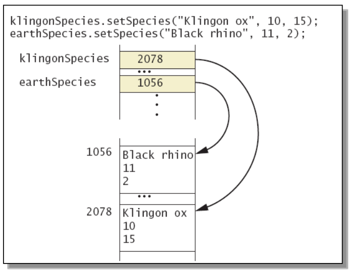
*Two memory locations for the two variables*

```
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

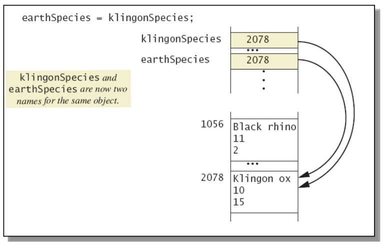
- Behavior of class variables



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

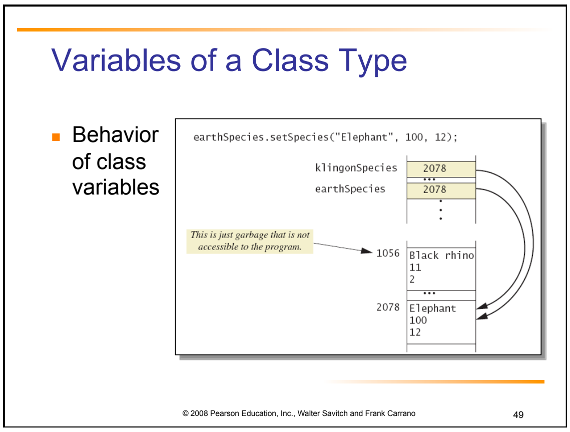
## Variables of a Class Type

- Behavior of class variables



```
earthSpecies = klingonSpecies;
```

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



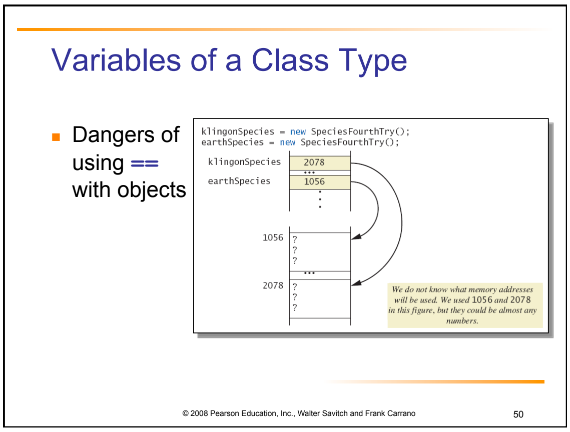
## Variables of a Class Type

- Behavior of class variables

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

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

© 2008 Pearson Education, Inc., Walter Savitch and Frank Carrano 49



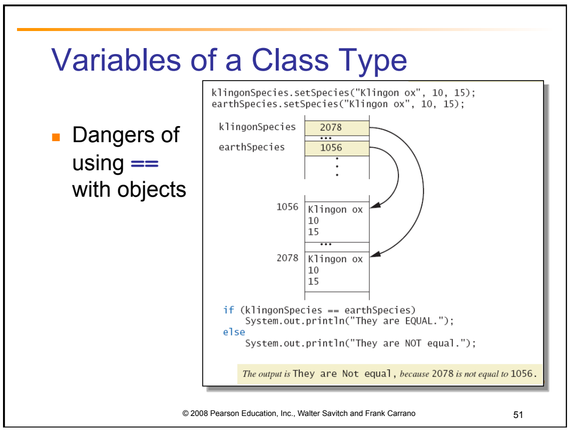
## Variables of a Class Type

- 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.*

© 2008 Pearson Education, Inc., Walter Savitch and Frank Carrano 50



## 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.

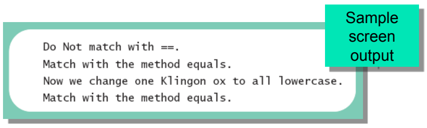
© 2008 Pearson Education, Inc., Walter Savitch and Frank Carrano 51

## Defining an **equals** Method

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

## Demonstrating an **equals** Method

- View sample program, listing 5.16  
  **class SpeciesEqualsDemo**
- Note difference in the two comparison methods **==** versus **.equals( )**

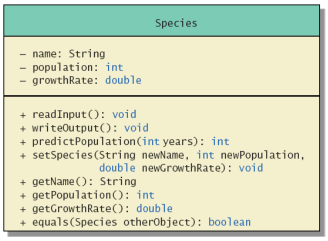


```
Do Not match with ==.
Match with the method equals.
Now we change one Klingon ox to all lowercase.
Match with the method equals.
```

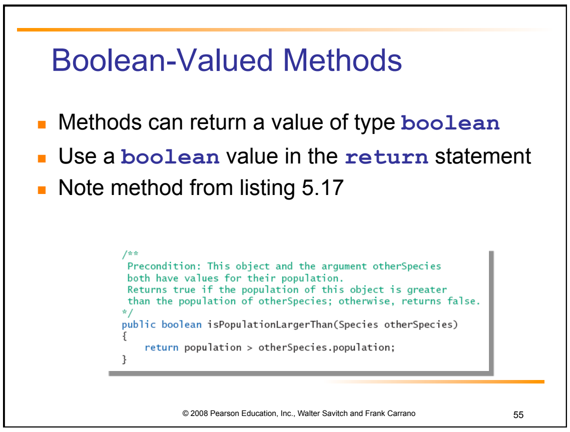
Sample screen output

## Programming Example

- View sample code  
  class **Species**
- Figure 5.7  
  Class Diagram  
  for the class  
  **Species**  
  in listing 5.17



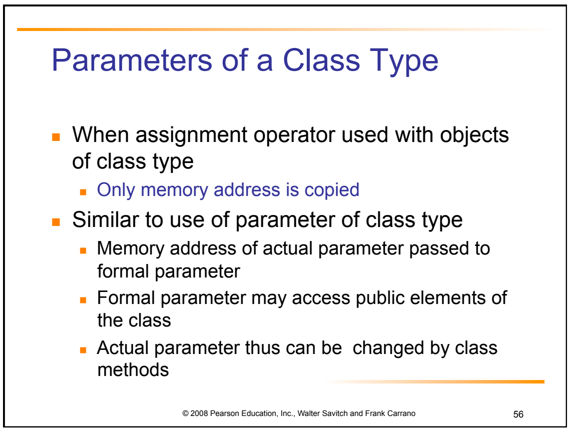
```
Species

– 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

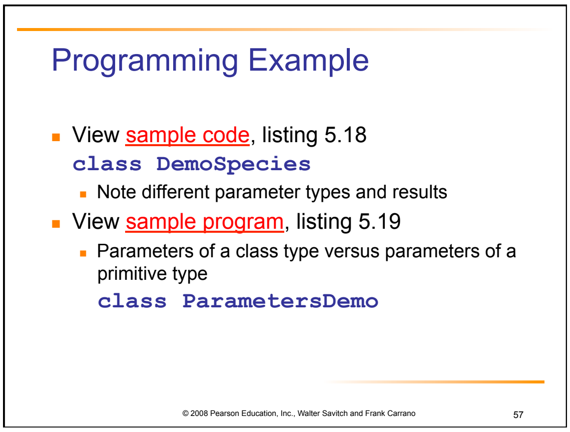
- Methods can return a value of type **boolean**
- Use a **boolean** value in the **return** statement
- Note method from listing 5.17

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



## Parameters of a Class Type

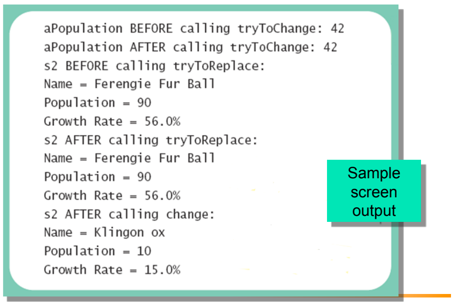
- When assignment operator used with objects of class type
  - Only memory address is copied
- Similar to use of parameter of class type
  - Memory address of actual parameter passed to formal parameter
  - Formal parameter may access public elements of the class
  - Actual parameter thus can be  changed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



## Programming Example

- View sample code, listing 5.18
  **class DemoSpecies**
  - Note different parameter types and results
- View sample program, listing 5.19
  - Parameters of a class type versus parameters of a primitive type
    **class ParametersDemo**

## Programming Example



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

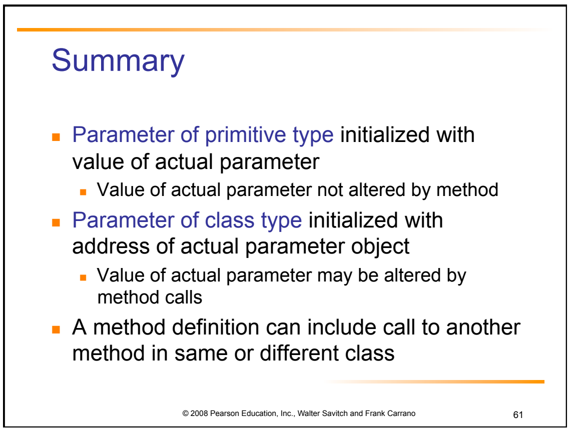
Sample screen output

## Summary

- Classes have
  - Instance variables to store data
  - Method definitions to perform actions
- Instance variables should be private
- Class needs accessor, mutator methods
- Methods may be
  - Value returning methods
  - Void methods that do not return a value

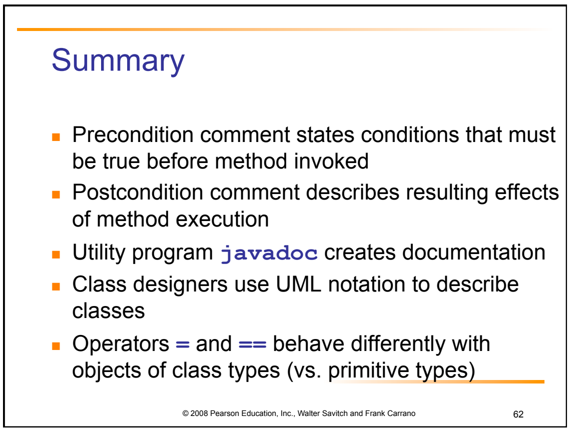
## Summary

- Keyword **`this`** used within method definition represents invoking object
- Local variables defined within method definition
- Formal arguments must match actual parameters with respect to number, order, and data type
- Formal parameters act like local variables



## Summary

- Parameter of primitive type initialized with value of actual parameter
  - Value of actual parameter not altered by method
- Parameter of class type initialized with address of actual parameter object
  - Value of actual parameter may be altered by method calls
- A method definition can include call to another method in same or different class



## Summary

- Precondition comment states conditions that must be true before method invoked
- Postcondition comment describes resulting effects of method execution
- Utility program `javadoc` creates documentation
- Class designers use UML notation to describe classes
- Operators = and == behave differently with objects of class types (vs. primitive types)