Defining Classes and Methods

Chapter 5

Objectives

- Describe concepts of class, class object

- Create class objects

- Define a Java class, its methods

- Describe use of parameters in a method

- Use modifiers `public`, `private`

- Define accessor, mutator class methods

- Describe information hiding, encapsulation

- Write method pre- and postconditions

Objectives

- Describe purpose of **`javadoc`**

- Draw simple UML diagrams

- Describe references, variables, parameters of a class type

- Define boolean-valued methods such as **`equals`**

- In applets use class **`Graphics`**, labels, **`init`** method

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

- Figure 5.1 A class as a blueprint

Class Name: Automobile

Data:
    amount of fuel_____
    speed _____
    license plate _____

Methods (actions):
    accelerate:
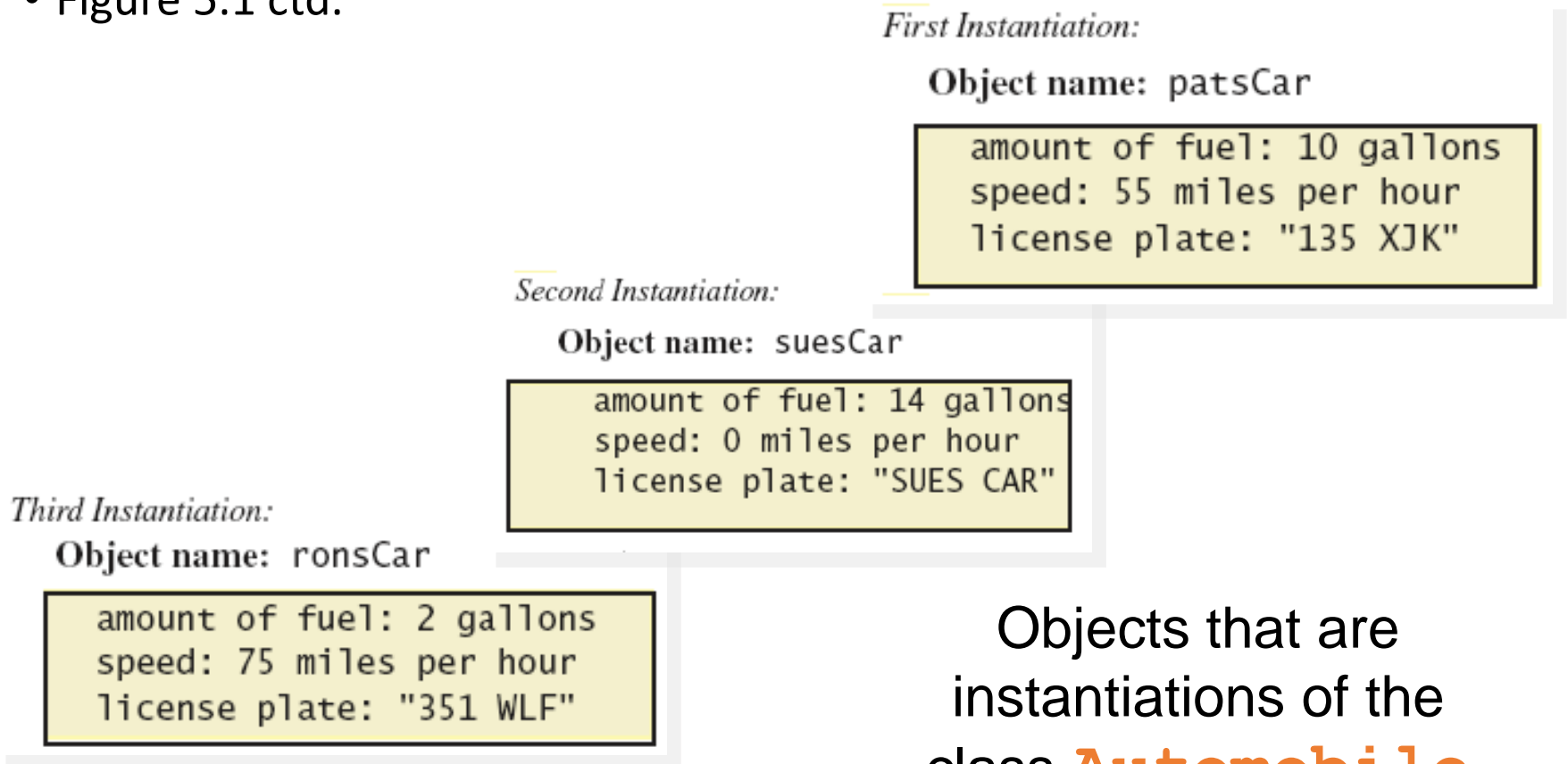   **How:** Press on gas pedal.
    decelerate:
        How: Press on brake pedal.
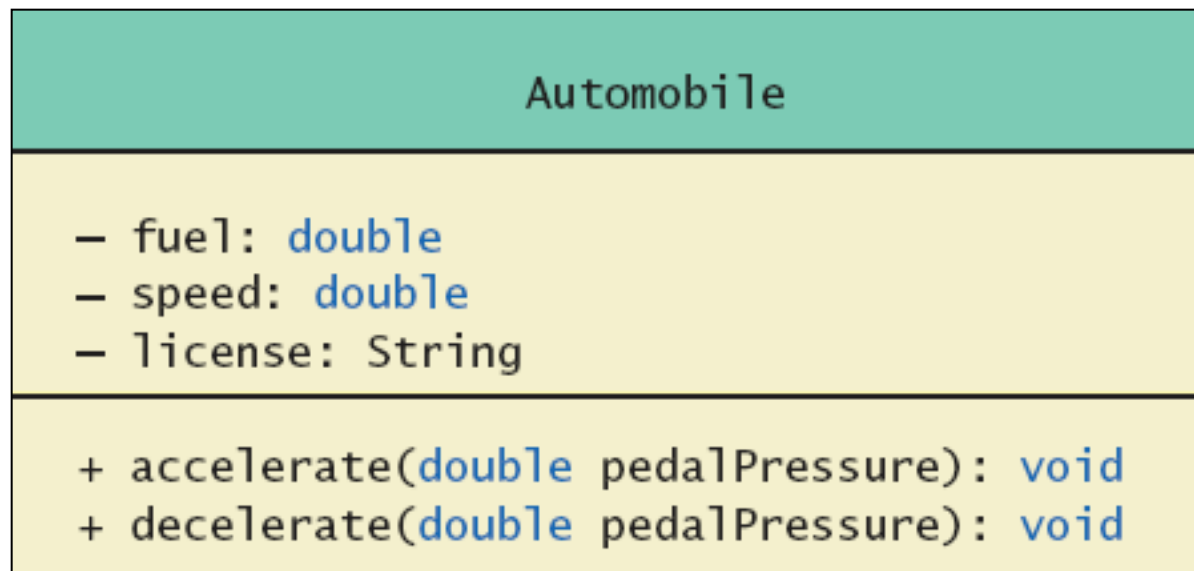
Class and Method Definitions

- Figure 5.1 ctd.

First Instantiation:

Object name: `patsCar`

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

Second Instantiation:

Object name: `suesCar`

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

Third Instantiation:

Object name: `ronsCar`

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

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

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram

Automobile
— fuel: double — speed: double — license: String
+ accelerate(double pedalPressure): void + decelerate(double pedalPressure): void

Class Files and Separate Compilation

- Each **Java** class definition usually in a file by itself
  - File begins with name of the class
  - Ends with .**java**
- Class can be compiled separately
- Helpful to keep all class files used by a program in the same directory

Dog class and Instance Variables

- View [sample program](#), listing 5.1
  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**

Using a Class and Its Methods

- View [sample program](#), listing 5.2
  `class DogDemo`

```
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.
```

Sample screen output

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 Listing 5.1

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

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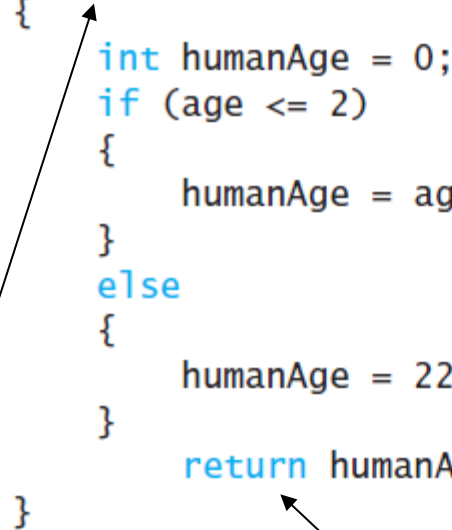 - Can be used only with objects of that class

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( )**

```java
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**

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

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

- View [sample file](#), listing 5.5A
  class BankAccount

- View [sample file](#), listing 5.5B
  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

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

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

- Recall method declaration in listing 5.3
  - Note it only works for 10 years
  - We can make it more versatile by giving the method a parameter to specify how many years

- Note sample program, listing 5.6
  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

- 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
```

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 <u>not</u> **public**

  - Instead specify as **private**

- View [sample code](#), listing 5.8
  class SpeciesThirdTry

Programming Example

- Demonstration of need for private variables

- View sample code, listing 5.9

- Statement such as

  box.width = 6;

  is <u>illegal</u> 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 5.10
  ## `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 **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
- View [sample code](#), listing 5.11

  ## `class SpeciesFourthTry`

- Note the mutator method

  - **`setSpecies`**

- Note accessor methods

  - **`getName`**, **`getPopulation`**, **`getGrowthRate`**

Accessor and Mutator Methods

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

Programming Example

- A Purchase class

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

- View demo program, listing 5.14
  ## 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 5.15
  class Oracle

- View demo program, listing 5.16
  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 <u>what</u> they do
  - We do <u>not</u> see mechanical details of <u>how</u> they do their jobs
- Encapsulation divides class definition into
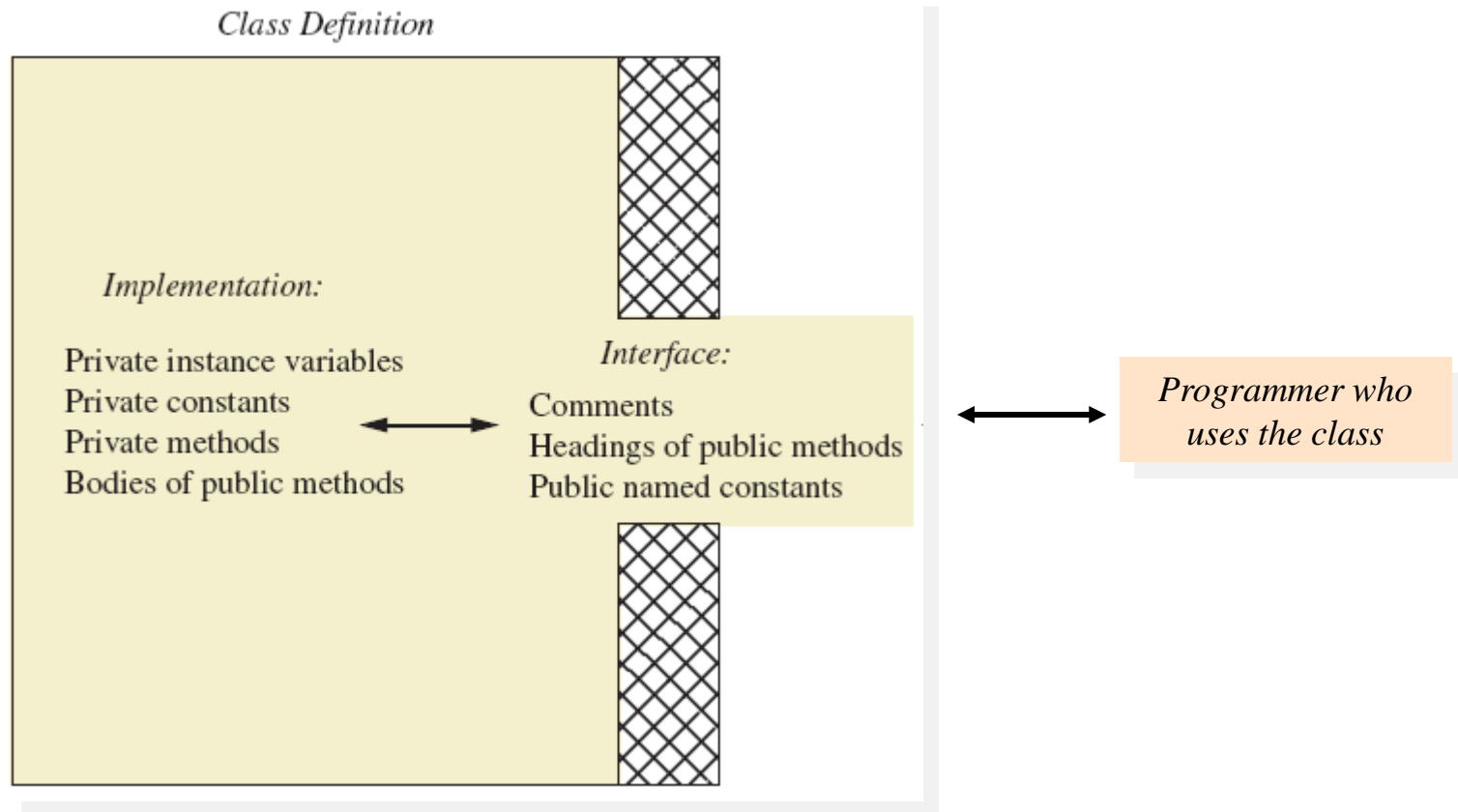 - Class interface
  - Class implementation

Encapsulation

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

Encapsulation

- Figure 5.3 A well encapsulated class definition

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
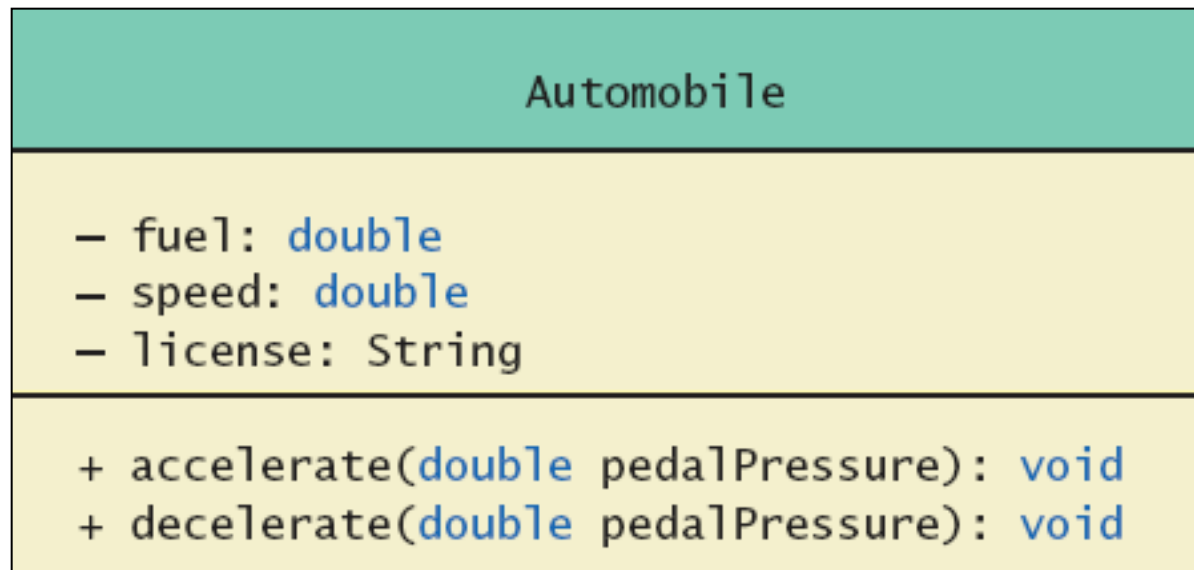
Encapsulation

- 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
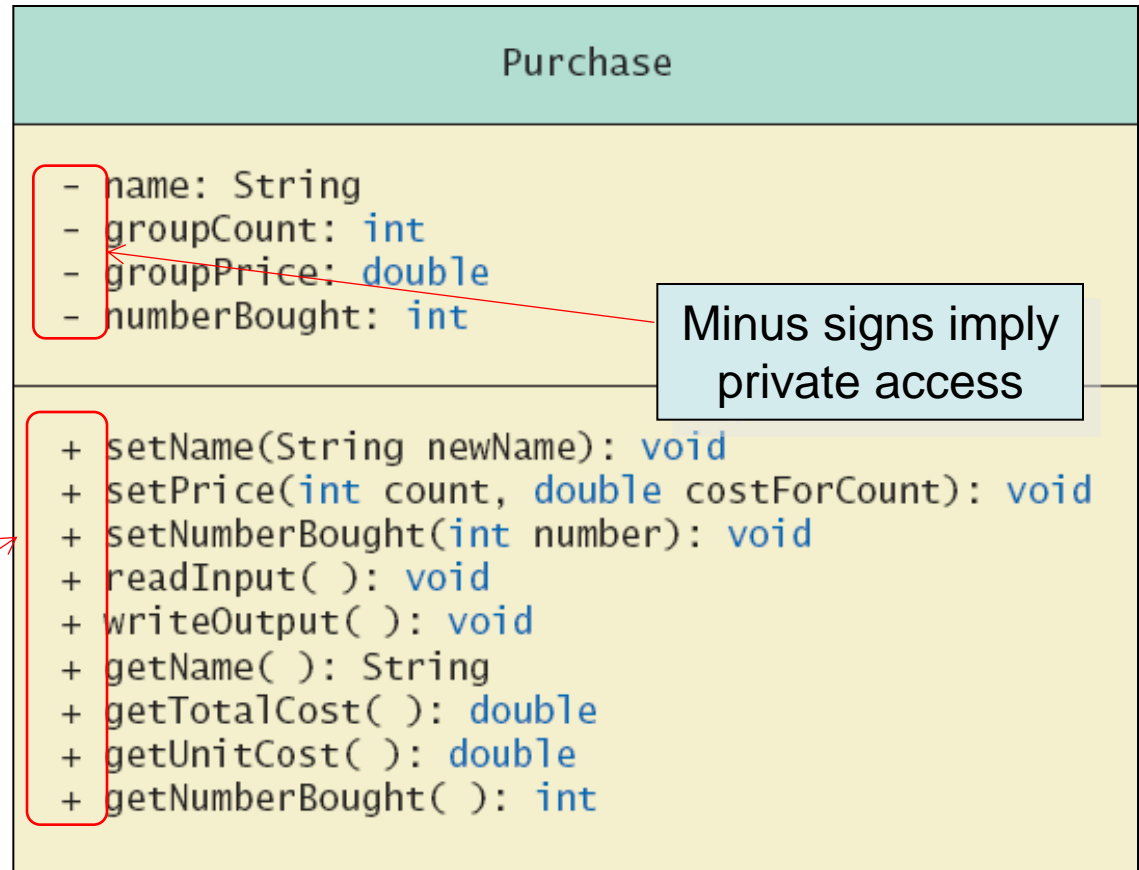
UML Class Diagrams

- 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 access

Plus signs imply public access

UML Class Diagrams

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

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)
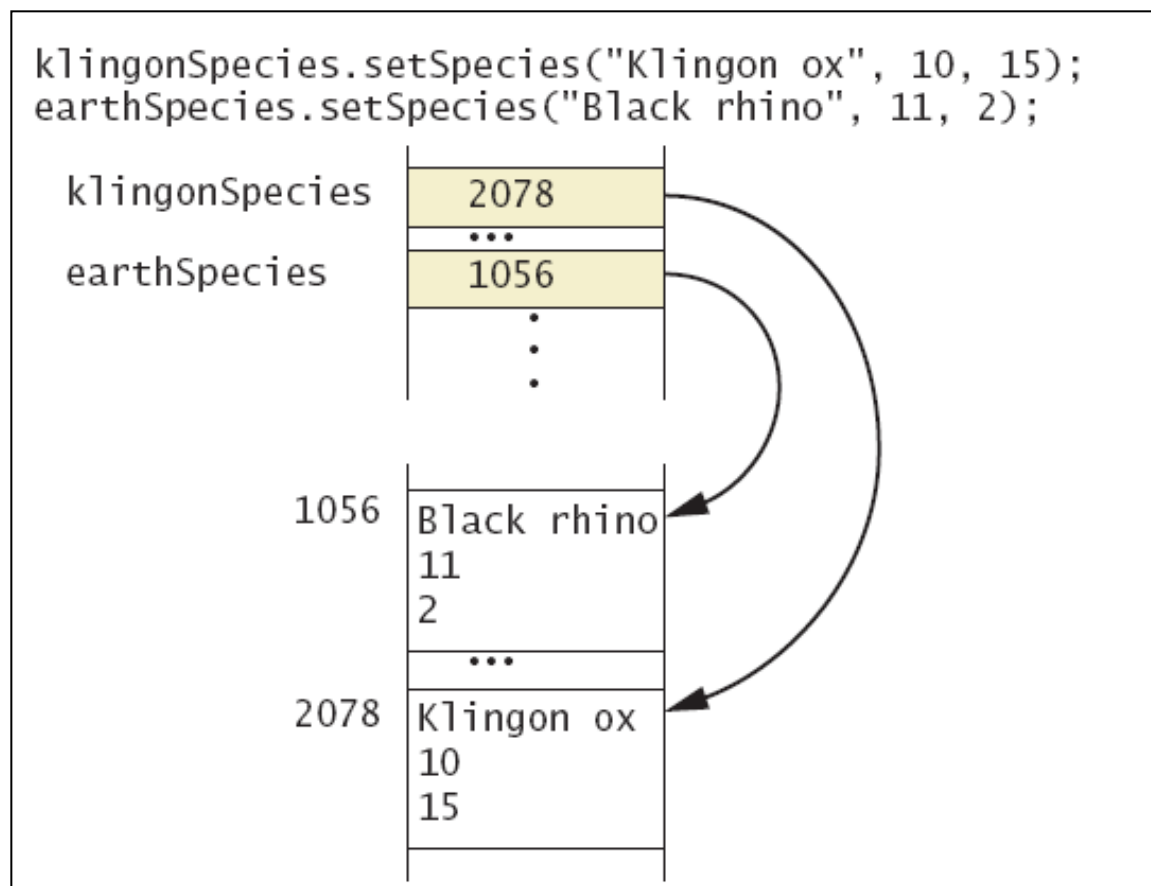 - This makes memory management of class types more efficient

Variables of a Class Type
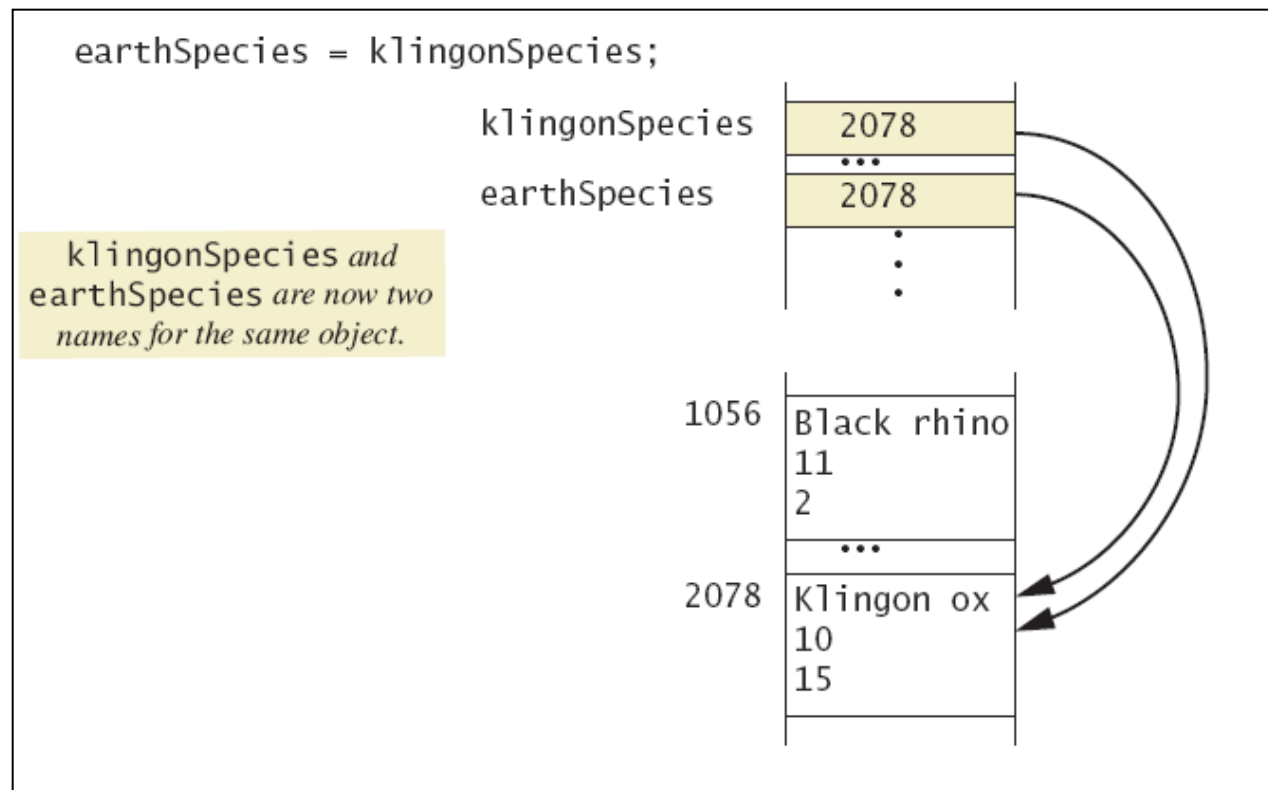
- Figure 5.5a Behavior of class variables
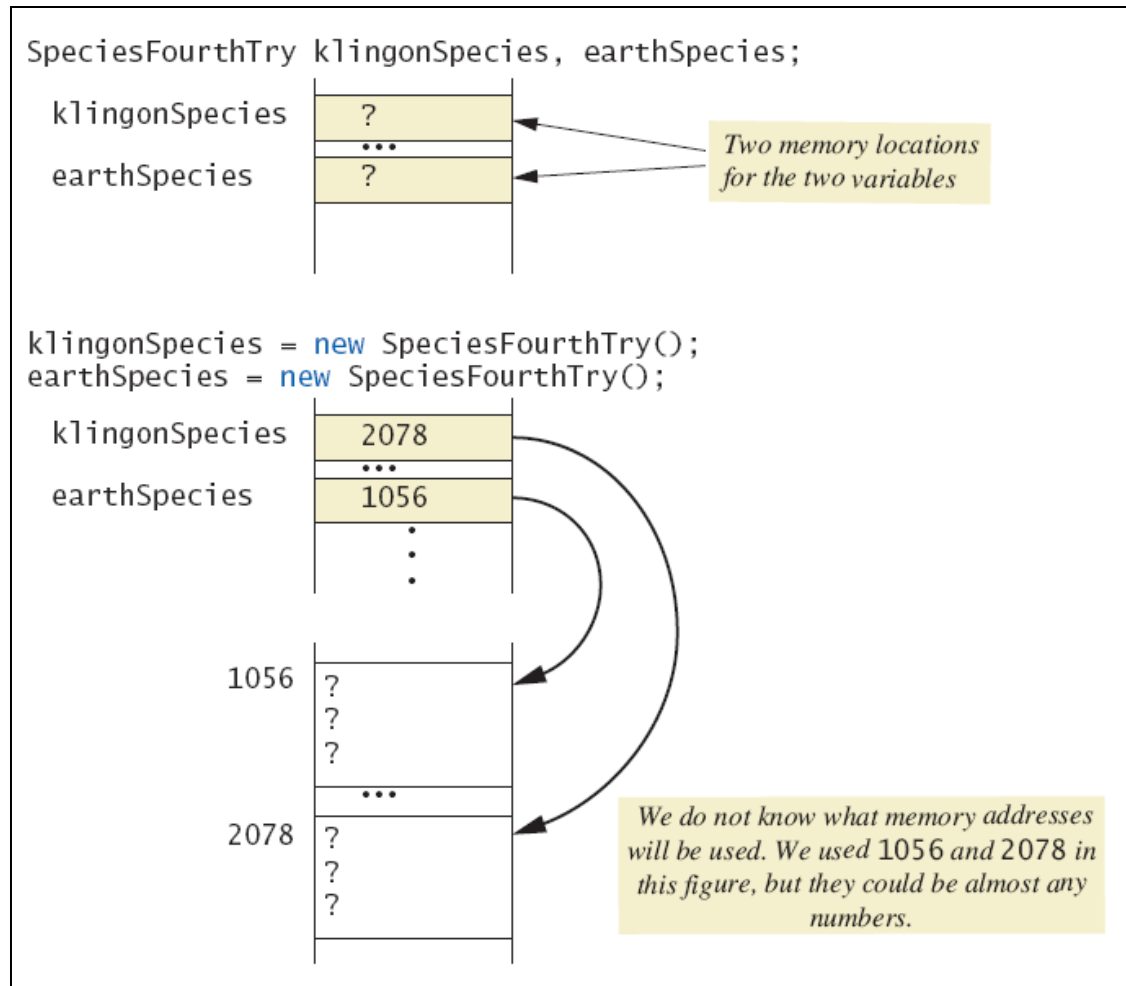
Variables of a Class Type

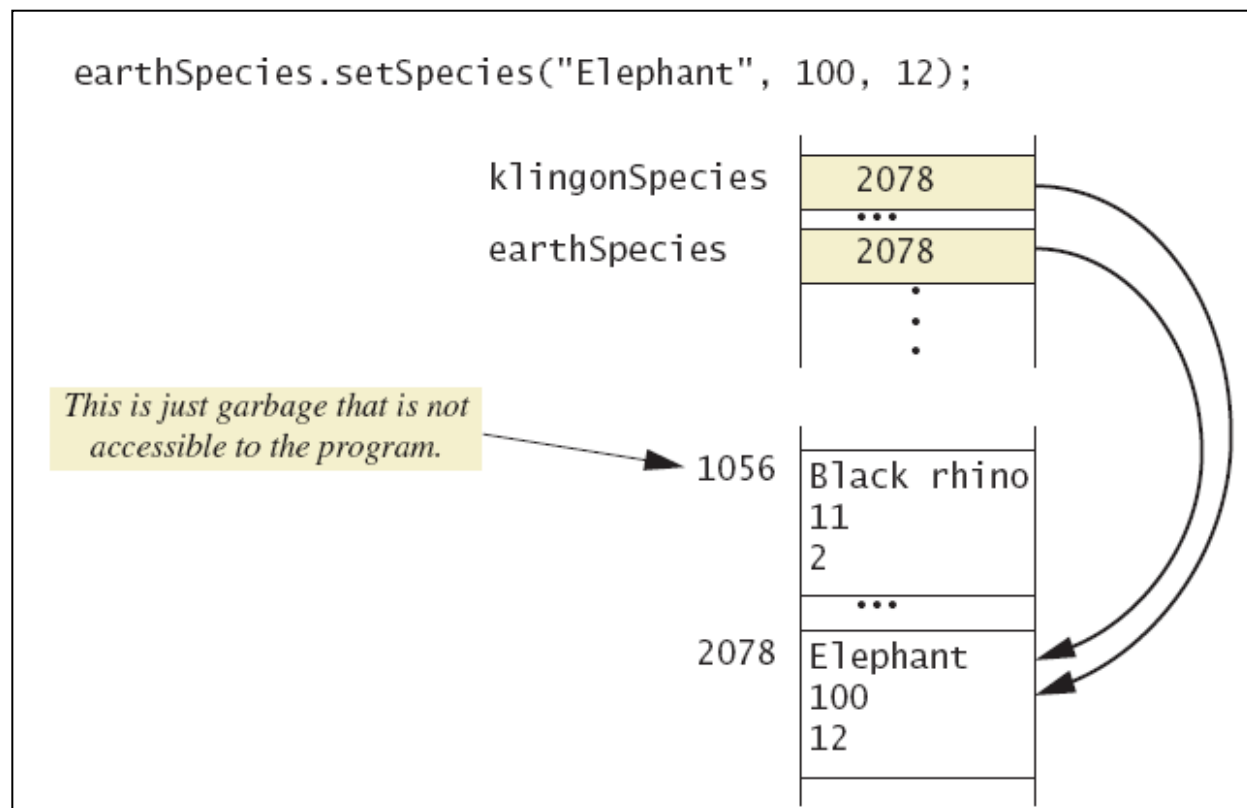- Figure 5.5b Behavior of class variables

Variables of a Class Type
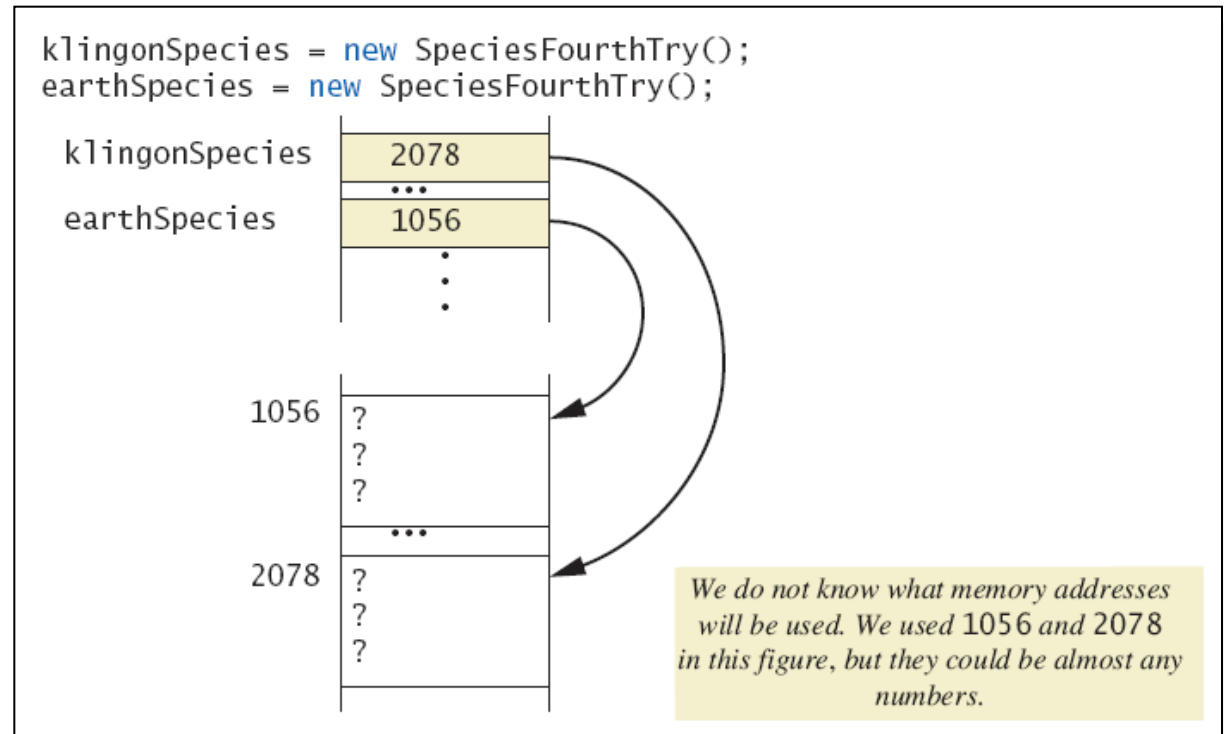
- Figure
  5.5c
  Behavior
  of class
  variables


```
earthSpecies = klingonSpecies;
```

klingonSpecies | 2078 |
                  | ... |
earthSpecies | 2078 |
                  | . |
                  | . |
                  | . |

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

1056 | Black rhino |
     | 11 |
     | 2 |
     | ... |
2078 | Klingon ox |
     | 10 |
     | 15 |

Variables of a Class Type

- Figure 5.5d Behavior of class variables

Variables of a Class Type

- Figure 5.6a Dangers of using **==** with objects


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

klingonSpecies   2078
...
earthSpecies   1056
.
.
.

1056   ?
       ?
       ?
...
2078   ?
       ?
       ?

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

- Figure 5.6b Dangers of using **==** with objects


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

klingonSpecies   2078
                  ...
earthSpecies   1056
                  ⋮

1056   Klingon ox
       10
       15
       ...
2078   Klingon ox
       10
       15
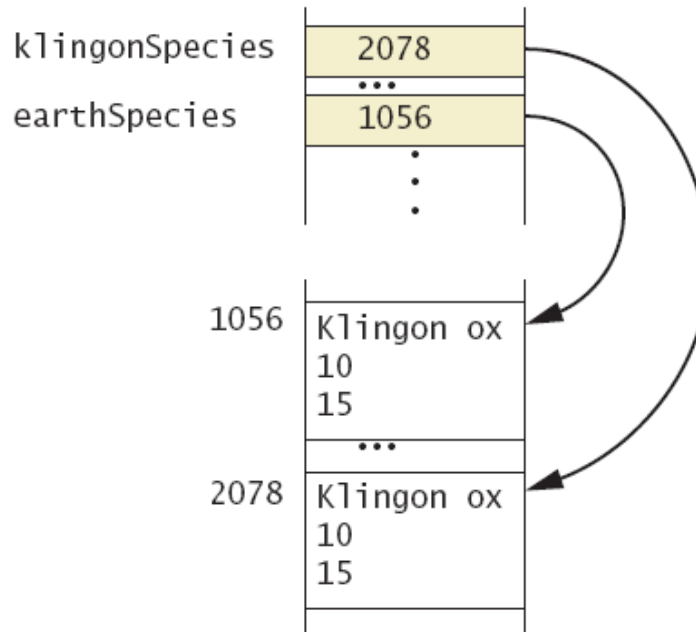
```
if (klingonSpecies == earthSpecies)
    System.out.println("They are EQUAL.");
else
    System.out.println("They are NOT equal.");
```

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
- View sample code, listing 5.17
  class Species

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

Demonstrating an **equals** Method

- View [sample program](), listing 5.18
  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

Complete Programming Example

- View sample code, listing 5.19 class **Species**

- Figure 5.7
  Class Diagram
  for the class
  Species
  in listing 5.19

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 5.19

```
/**
 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**

**View <u>sample code</u>, 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`

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

Graphics Supplement: Outline

- The Graphics Class

- The **init** Methods

- Adding Labels to an Applet

The **Graphics** Class

- An object of the **Graphics** class represents an area of the screen

- Instance variables specify area of screen represented

- When you run an Applet

  - Suitable **Graphics** object created automatically

  - This object used as an argument in the **paint** method

The **Graphics** Class

- Figure 5.8a Some methods in class **Graphics**

Graphics_Object.`drawOval`(*X*, *Y*, *Width*, *Height*)   Draws the outline of an oval having the specified width and height at the point (*X*, *Y*).
Graphics_Object.`fillOval`(*X*, *Y*, *Width*, *Height*)   Same as `drawOval`, but the oval is filled in.

Graphics_Object.`drawArc`(*X*, *Y*, *Width*, *Height*, *Start_Angle*, *ArcAngle*)
  Draws an arc—that is, draws part of an oval. See the graphics supplement section of
  Chapter 1 for details.

Graphics_Object.`fillArc`(*X*, *Y*, *Width*, *Height*, *Start_Angle*, *ArcAngle*)   Same as `drawArc`, but the visible portion of the oval is filled in.

The **Graphics** Class

- Figure 5.8b Some methods in class **Graphics**

Graphics_Object.drawRect(*X*, *Y*, *Width*, *Height*)
   Draws the outline of a rectangle of the specified width and height at the point (*X, Y*).

Graphics_Object.fillRect(*X*, *Y*, *Width*, *Height*)
   Same as drawRect, but the rectangle is filled in.

Graphics_Object.drawLine(*X1*, *Y1*, *X2*, *Y2*)
   Draws a line between points (*X1, Y1*) and (*X2, Y2*).

Graphics_Object.drawString(*A_String*, *X*, *Y*)
   Writes the specified string starting at the point (*X, Y*).
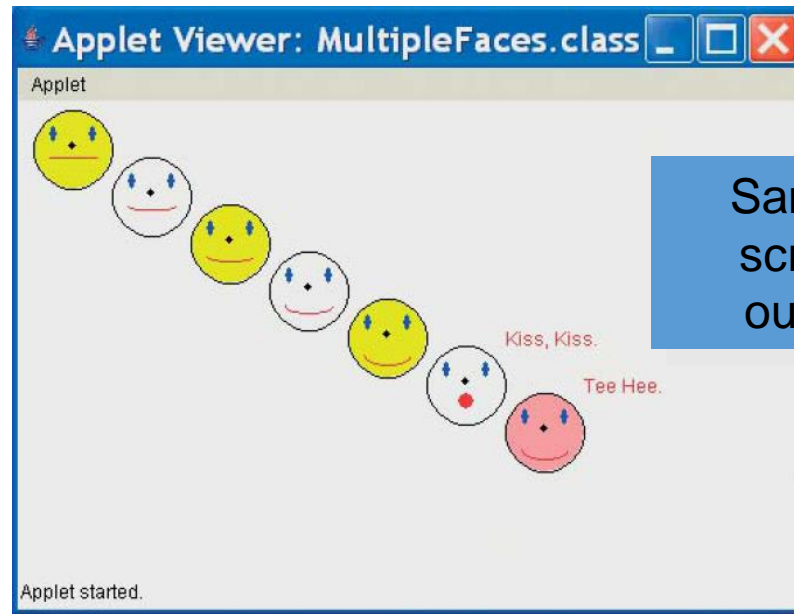
Graphics_Object.setColor(*Color_Object*)
   Sets the color for subsequent drawings and text. The color stays in effect until it is changed by another invocation of setColor.

Programming Example

- Multiple faces – using a Helping method
- View sample code, listing 5.23
  ## class MultipleFaces

Sample screen output

Graphics2D

- Java includes a Graphics2D class with more functionality than the Graphics class
  - Extends AWT
  - Support for more geometric shapes
  - Textures
  - Gradients
  - Font and shape properties
  - Filtering
  - Compositing
  - Transforming
- Features that would be useful for rich media applications or video games
- View sample program, listing 5.24, for reading an image file and drawing it with transparency

The `init` Method

- Method `init` may be defined for any applet

- Like `paint`, method `init` called automatically when applet is run

- Method `init` similar to method `main` in an application program

Adding Labels to Applet

- Provides a way to add text to an applet

- When component (such as a label) added to an applet

  - Use method **`init`**

  - Do <u>not</u> use method **`paint`**

Adding Labels to Applet

- View sample applet, listing 5.25
  ## class LabelDemo

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)

Summary

- Designer of class should include an `equals` method
- Graphics drawn by applet normally done from within `paint`
  - Other applet instructions placed in `init`
- Parameter of `paint` is of type `Graphics`
- Method `setBackground` sets color of applet pane
- Labels added to content pane within the init method