Defining Classes and Methods

Chapter 4

Objectives

- become familiar with the concept of
  - a class
  - an object that instantiates the class
- learn how to
  - define classes
  - define and use methods
  - create objects
  - use parameters in methods

Objectives, cont.

- learn about
  - information hiding and encapsulation
  - the notion of a reference
    - understand class variables and class parameters

Outline

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

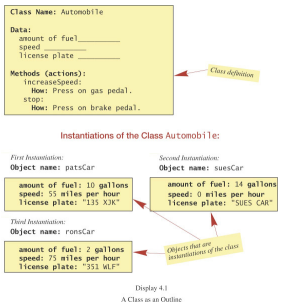
Basic Terminology

- A *class* defines a kind of objects:
  - specifies the kinds of *attributes* (*data*) an object of the class can have.
  - provides *methods* specifying the *actions* an object of the class can take.
- An object is an *instance* of the class.
- Person is a class
  - Alice and Bob are objects of the Person class.

What does a class have?

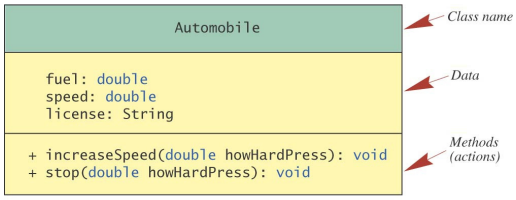
- *Members* of a class*:*
  - *Attributes (instance variables, data)*
    - For each instance of the class (object), values of attributes can vary, hence instance variables
  - *Methods*
- Person class
  - Attributes: name, address, phone number
  - Methods: change address, change phone number
- Alice object
  - Name is Alice, address is …
- Bob object
  - Name is Bob, address is …

A Class as an Outline



Display 4.1
A Class as an Outline

A UML Class Diagram



Display 4.2
A Class Outline as a UML Class Diagram

Method Definitions

- Methods belong to a class
  - Defined inside the class
- *Heading*
  - Return type (e.g. `int, float, void`)
  - Name (e.g. `nextInt, println`)
  - Parameters (e.g. `println(…)`)
  - More…
- *Body*
  - enclosed in braces `{}`.
  - Declarations and/or statements.

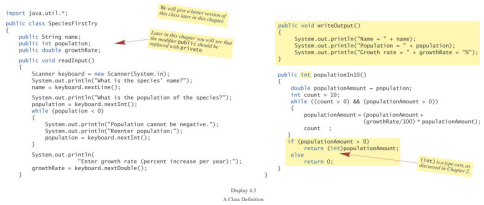
The Method `main`

- A program
  - a class that has a method named `main`.
  - Execution begins in the `main` method
- So far
  - no attributes (instance variables)
  - no methods other than method `main`.


```
------------ Person.java ------ defining Person ---------------------
public class Person
{
  private String _name;
  private String _iceCream;

  public void setName(String newName)
  {
    this._name = newName;   // this. is optional
  }

  public void setIceCream(String newIceCream)  { … }

  public void print()
  {
     System.out.println(this._name + " likes " + this._IceCream); // this. optional
  }
}

------------ PersonTest.java ----- using Person ------------------
public class PersonTest
{
  public static void main(String[] args)
  {
    Person joe = new Person();
    joe.setName("Joseph");
    joe.setIceCream("Rocky Road");

    Person mary = new Person();
    mary.setName("Mary");
    mary.setIceCream("Chocolate Fudge");
    mary.print();
  }
}
```

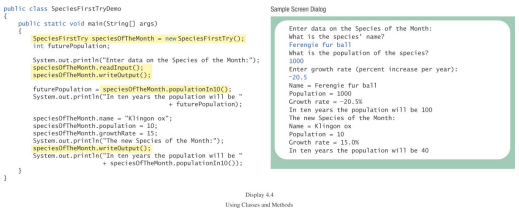

Example

- `class SpeciesFirstTry`



Example, contd.

- `class SpeciesFirstTryDemo`



```
public class SpeciesFirstTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
        int futurePopulation;

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

        futurePopulation = speciesOfTheMonth.populationIn10();
        System.out.println("In ten years the population will be "
                                + futurePopulation);

        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.populationIn10());
    }
}
```

Sample Screen Dialog

```
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 (percent 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
```

Display 4.4
Using Classes and Methods

- Each object of type `SpeciesFirstTry` has its own values for the three attributes

Class Files and Separate Compilation

- One class definition per file
- File name and class name are the same
  - File extension is `.java`
- After compilation
  - byte code is stored in a file with extension `.class`
- Execution
  - `.class` is run in the JVM
- Multiple `.class` files in the same directory
  - No need to *import* them

Two Types of Methods

1. Return a value
   - `next = keyboard.nextInt();`
   - `keyboard` is the *calling object.*
2. Don't return a value, called *`void` method*
   - `System.out.println("Enter data:");`
   - `System.out` is the calling object

`void` Method Definitions

- example

```
public void writeOuput()  //heading
{  //body
   System.out.println("Name: " + name);
   System.out.println("Age: " + age);
}
```


Using `return` in a `void` Method

- form

```
return;
```

- usage
  - end the invocation of the method prematurely for dealing with some problem
- caution
  - better ways to deal with a potential problem ("exceptions") [later...]

Defining Methods That Return a Value

- example

```
public float density(float area) // heading
{  // body
   return population / area;
}
```

Defining Methods That Return a Value, cont.

- must contain *return statement*
  `return` *`Expression`*`;`
  - *`Expression`* must have a type that matches the return type
- multiple `return` statements are allowed
  - a single `return` statement is better (one exit point for the method).

Naming Methods

- Use a verb to name methods
  - Actions
  - getBalance, deposit, changeAddress
- Start a method name with a lowercase letter.

`public` Method Definitions

- syntax for a `void` method

```
public void methodName(parameters)
{
    <statement(s)>
}
```


`public` Method Definitions

- syntax for methods that return a value

```
public returnType methodName(parameters)
{
    <statement(s), including a
     return statement>
}
```


Accessing Attributes

- Inside the definition of the same class
  - `<attributeName>` or `this.<attributeName>`
  - `name = "Lara";` or `this._name = "Lara";`
- Outside the class definition
  - i.e. in *another* class definition
  - `<objectName>.<attributeName>`
  - `myBestFriend._name = "Lara";`
  - Not recommended, later…

Example Using `this`

- Defining the `writeOutput` method

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

- Using the `writeOutput` method
  - `tiger.writeOutput()`
    - `this` refers to the `tiger` object

Example Using `this`

- Defining the `addPopulation` method:

```
public int addPopulation(SpeciesFirstTryDemo
   species2)
{
    return this.population + species2.population;
}
```

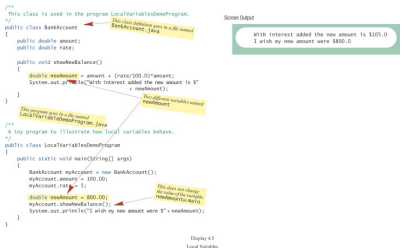
- Using the `addPopulation` method
  - `tiger.addPopulation(lion)`
    - `this` refers to the `tiger` object
    - `species2` refers to the `lion` object

Local Variables

- Declared within a method
  - "local to" (confined to) the method definition
  - can't be accessed outside the method
- *Not attributes (instance variables)*

Local Variables, cont.

- `class BankAccount`
- `class LocalVariablesDemoProgram`



Blocks

- *Block* and *compound statement*
  - a set of Java statements enclosed in braces { }.
- Variable declared within a block
  - local to the block.
  - when the block ends, the variable "disappears."
- Variable declared outside multiple blocks
  - spanning multiple blocks

Scope of a local variable

- The region in the program where the local variable can be accessed
  - Start: declaration of the variable
  - End: the closing } of the block in which the variable is declared

Example of Scope

```
public void method1()
{ // begin of block
  int x;

  if (…)
    { // begin of block
      int y;  // can't have int x here, confusing anyway
      …
    }
  else
    { // begin of block
      …   // can y be used here?
    }
  …       // can y be used here?
}

public int method2()
{ // begin of block
  int x;  // can I do this?

  …
  return x;
}
```

Example of Scope

```
public void method()
{
   for (int i=0; i < 10; i++) // start of a block
   {
     int y=0;
     …
   }
   // Are i and y accessible here?
}
```


Variables in `for` Statements

- Variable declared outside the for statement

```
int n;
for (n = 1; n <10; n++) {…}
```

  – variable `n` still exists when the for statement ends
- Variable declared inside the for statement

```
for (int n = 1; n <10; n++) {…}
```

  – variable `n` disappears when the for statement ends

Passing Values to a Method: Parameters

- Input values for methods (within the program, not from user)
  - *passed* values or *parameters*
- More flexibility for methods
- *formal* parameters
  - Part of method definition
  - After the method name, within parentheses
    - *type*
    - *name*
- *arguments*, or *actual* parameters
  - Calling a method with an object within the parentheses
    - matching data type
    - in the same order

Formal vs Actual Parameters

```
public static void main(String[] args)
{
  print("George Bush");
}

public static void print(String name)
{
  System.out.print("The name is: " + name);
}
```


Scope of Formal Parameters

- Start: begin of the method
- End: end of the method

```
public float square(float num)
{  // begin of num's scope
   …
}  // end of num's scope
```


Parameter Passing Example

```
//Definition of method to double an integer
public int doubleValue(int numberIn)
{
   return 2 * numberIn;
}
```

```
//Invocation of the method... somewhere in main...
...
int next = keyboard.nextInt();
System.out.println(someObj.doubleValue(next));
```

- What is the formal parameter in the method definition?
  - `numberIn`
- What is the argument (actual parameter) in the method invocation?
  - `next`

Type Matching/Constraint

- The type of each argument should match the corresponding formal parameter.
- If appropriate, Java automatically performs type casting:
  - Define: `float square(float num)`
  - Invoke: `int x=5; square(x);`

```
byte --> short --> int --> long --> float -->
  double
```


Pass-By-Value: *Primitive* Data Types as Parameters

- When the method is called
  - *value* of each argument is ***copied*** (assigned) to its corresponding formal parameter
- Formal parameters
  - initialized to the values passed
  - local to their method
- Variables used as arguments cannot be changed by the method
  - the method only gets a copy of the variable's value

Example for Pass-by-Value

```
public static void main(String[] args)
{
  int x = 10, num = 20;
  int sq = MyClass.square(x);
  System.out.println(x);
  System.out.println(num);
}

public static int square(int num)
{
   num = num * num;
   return num;
}
```


Arguments to Methods

- An argument in a method invocation can be
  - a literal such as `2` or `'A'`
  - a variable
  - an expression that yields a value

    [technically a literal or variable is also an "expression"]

Example for Pass-by-Value

```
public static void main(String[] args)
{
  int x = 10, area;
  area = MyClass.square(x);
  area = MyClass.square(5);
  area = MyClass.square(x + 5 % 2);
}

public static int square(int num)
{
   return num * num;
}
```

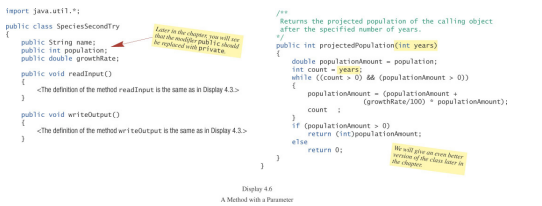

Multiple Arguments to Methods

```
anObject.doStuff(42, 100, 9.99, 'Z');
public void doStuff(int n1, int n2, double d1,
  char c1);
```

- arguments and formal parameters are matched by position
- Corresponding types need to match

Method with a Parameter

- `class SpeciesSecondTry`



Display 4.6
A Method with a Parameter

Using a Method with a Parameter

- `class SpeciesSecondTryDemo`

Display 4.7
Using a Method with a Parameter

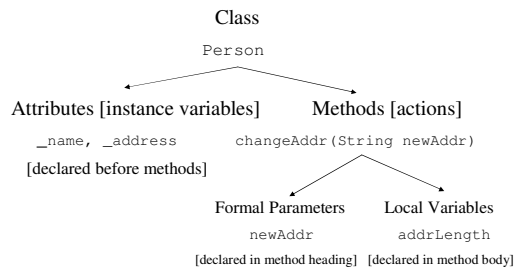
Summary of Class Definition Syntax

```
/*****************************************
 * Class description
*****************************************/
public class ClassName
{
   <attribute (instance variable) definitions>

   //Method definitions of the form
   /*******************************
    * Method description
    *******************************/
   public returnType methodName(type1 parmameter1, ...)
   {
      <statements defining the method>
   }

}
```


class SpeciesSecondTry

- Display 4.6, page 244
- http://www.cs.fit.edu/~pkc/classes/cse1001/src/ch04/SpeciesSecondTry.java

Attributes, Formal Parameters, Local Variables



Attributes, local variables, parameters

```
public class Person
{
   private String name; //attribute (instance variable)

   public void method1(String yourName)//parameter
   {
      String myName;   // local variable

      … this.name;      //?  #1
      … this.myName;    //?  #2
      … this.yourName;  //?  #3

      … name;           //?  #4
      … myName;         //?  #5
      … yourName;       //?  #6

   }
}
```

Information Hiding and Encapsulation: Outline

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

Information Hiding

- To drive a car, do you need to know how the engine works? Why?
- `println` method
  - need to know ***what*** the method does
  - but not ***how*** `println` does it
- Provide a more abstract view and hide the details

Information Hiding and Encapsulation

- Both are forms of *abstraction*

Information hiding	Encapsulation
• protect data inside an object	• Use classes and objects
• do not allow direct access	• Objects include both data items and methods to act on the data

`public` and `private`

public
- Attribute (instance variable)
  - any class can directly access/change
- Method
  - any class can invoke

private
- Attribute (instance variable)
  - only the same class can access/change
- Method
  - only the same class can invoke

`private` or `public` ?

- Attributes (instance variables)
  - should be `private`, why?
- Methods
  - usually `public`, why?
  - sometimes `private`
- Default is `public` in Java
  - Convention is to explicitly state `public` or `private`

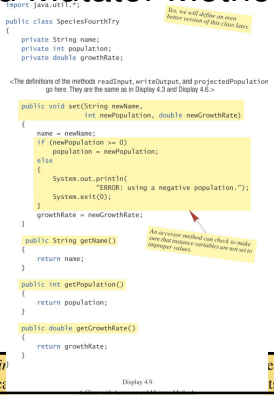
Accessors and Mutators

- ***accessor methods***—public methods that allow attributes (instance variables) to be read
- ***mutator methods***—public methods that allow attributes (instance variables) to be modified
  - Check to make sure that changes are appropriate.
  - **Much better** than making instance variables `public`

`private` attributes (instance variables) with **`public`** accessor and mutator methods

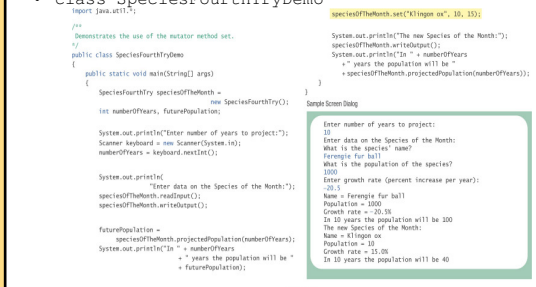
Accessor and Mutator Methods

- class SpeciesFourthTry



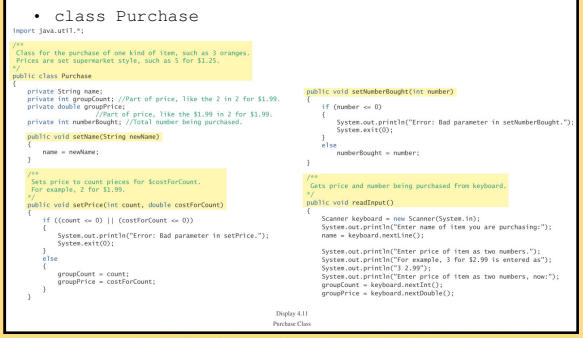
Accessor and Mutator Methods

- class SpeciesFourthTryDemo

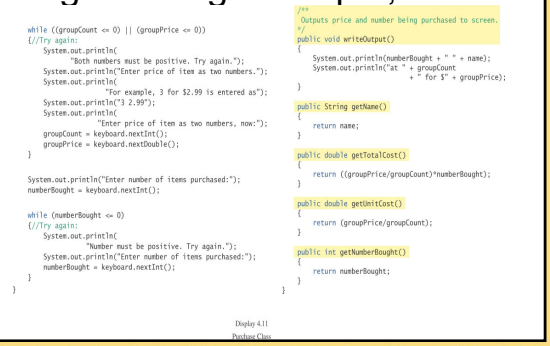


Programming Example

- class Purchase

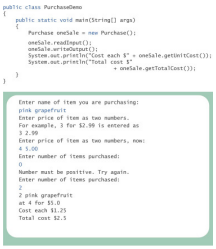


Programming Example, contd.



Programming Example

- class PurchaseDemo



Precondition and Postcondition Comments

- efficient and standard way to tell what a method does
- ***precondition***—states conditions that must be true before method is invoked
- ***postcondition***—tells the effect of a method call
- Example:

```
/**
Precondition: years is a nonnegative number
Postcondition: Returns the projected population after the specified
    number of years
*/
```

- Note that the terms preconditions and postconditions are not always used, particularly if the only postcondition describes the return value of the method.

Assertion Checks

- ***assertion***—statement that should be true if there are no mistakes in the program
- Preconditions and postconditions are examples of assertions.
- Can use `assert` to see if assertion is true.
- Syntax:

```
assert Boolean_Expression;
```

- Example:

```
assert n >= limit;
```

- If assertion is false when checked, the program ends and an error message is printed.
- Assertion checking can be turned on and off.
  - The exact way to enable or disable assertions depends on your development environment.

Encapsulation

- *Encapsulation* is the process of hiding details of a class definition that are not needed to use objects of the class.
- Encapsulation is a form of information hiding.

Encapsulation, cont.

- Two parts of a class definition:
  1. *user interface*
     - communicates everything needed to use the class
  2. *Implementation*
     - all the members of the class.
- A class defined this way is said to be *well encapsulated.*

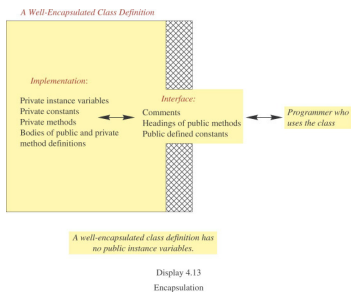
User Interface

- consists of
  - headings for the public methods
  - defined public constants
  - comments telling the programmer how to use the public methods and the defined public constants.
- contains everything needed to use the class.

Implementation

- consists of
  - private attributes (instance variables)
  - private defined constants
  - definitions of public and private methods.
- Java code contains both the user interface and the implementation.

Encapsulation



Display 4.13
Encapsulation

Encapsulation Guidelines

- Precede the class definition with a comment that shapes the programmer's view of the class.
- Declare all the attributes (instance variables) in the class `private`.
- Provide appropriate public accessor and mutator methods.

Encapsulation Guidelines, cont.

- Provide `public` methods to permit the programmer to use the class appropriately.
- Precede each `public` method with a comment specifying how to use the method.
- Declare other methods `private`.
- Use `/*...*/` or `/**...*/` for user interface comments and `//` for implementation comments.

Encapsulation Characteristics

- permit implementation changes without changes to the interface.
- combine data and methods into a single entity, "hiding" the details of the implementation.

ADT

- An *abstract data type (ADT)* is basically the same as a well-encapsulated class definition.

Automatic Documentation with `javadoc`

- A program named `javadoc` automatically generates user interface documentation.
- The documentation contains everything needed to use the class(es).
- Properly commented class definitions (using `/**...*/`) can be used to produce and display the user interface.
- Documents produced by javadoc are in HTML.

UML Class Diagrams

- UML diagrams are mostly self-explanatory.
- plus sign (+) indicates public
- minus sign (-) indicates private
- Typically, the class diagram is created before the class is defined.
- A class diagram outlines both the interface and the implementation.

UML Class Diagrams, cont.

Display 4.14
UML Class Diagram

Object-Oriented (OO) Design

- Object-Oriented
  - Think what the objects (data) are first
  - Then their functionality (methods)
  - Then how the objects and their functionality can be used to create more complex functionality
- Each class encapsulates the data (attributes, instance variables) and functionality (methods) of the objects.
- When a method is called on an object, *"the object knows what to do on its own."* The caller doesn't need to know what to do.
- OO design is not always appropriate…
- http://www.cs.fit.edu/~pkc/classes/cse1001/OOmoney/

Objects and Reference: Outline

- Variables of a Class Type and Objects
- Boolean-Valued Methods
- Class Parameters
- Comparing Class Parameters and Primitive-Type Parameters

Variables: Class Type vs. Primitive Type

- What does a variable hold?
  - *primitive* type
    - *value* of the variable
  - *class* type
    - *memory address* (*reference*) of the object
      - not the value(s) of the object
      - objects generally do not have a single value and they also have methods, so what's its "value?"

Post Office Analogy

What's the pink card for and why?

Mailboxes and Java

- Letters: smaller, standard sizes (primitive type objects)
- Packages: larger, no standard sizes (class type objects)
- Mailbox (variable):
  - Primitive type: Letter itself (value/content of a primitive type object)
  - Class type: Pink card to get the package (reference/pointer/address of a class type object)

Advantages of separate treatment

- Mailboxes are tightly packed and well organized (primitive types)
  - Efficient access and storage
- Packages are not as well organized (classes types)
  - Less efficient access and storage
- Different memory segments for primitive type objects (mailboxes) and class types objects (back of the mailroom)
- Easier to move a package or a pink card around?
  - Parameter passing—faster to pass an address than a class type object
  - Returning from methods

Allocating Memory for a Reference and an Object

- A declaration such as

  `SpeciesFourthTry s;`

  creates a variable `s` that can hold a memory address (reference).
- A statement such as

  `s = new SpeciesFourthTry();`

  allocates memory for an object of type `SpeciesFourthTry` and assign its memory address to variable `s`.

Issues with Class Type Variables

- Assignment (=)
- Equality (==)
- Parameter passing

Assignment with Variables of a Class Type

```
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth:");
earth.writeOutput();
System.out.println("klingon:");
klingon.writeOutput();
```

What will the output be?

(see the next slide)

Assignment with Variables of a Class Type

```
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth:");
earth.writeOutput();
System.out.println("klingon:");
klingon.writeOutput();
```

What will the output be?

`klingon` and `earth` both print `Elephant`.

Why do they print the same thing?

(see the next slide)

Output:

```
earth:
Name = Elephant
Population = 100
Growth Rate = 12%
klingon:
Name = Elephant
Population = 100
Growth Rate = 12%
```

Assignment with Variables of a Class Type

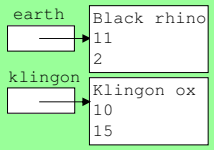
```
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth:");
earth.writeOutput();
System.out.println("klingon:");
klingon.writeOutput();
```

Why do they print the same thing?

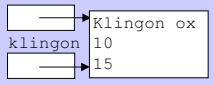
The assignment statement makes `earth` and `klingon` refer to the same object.

When `earth` is changed to "`Elephant`", `klingon` is changed also.

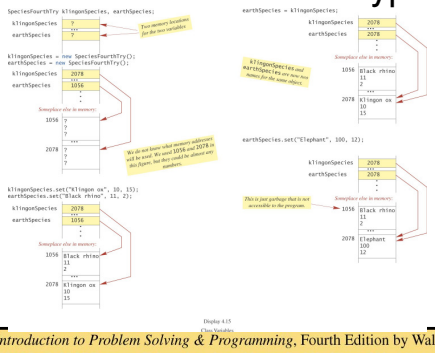
Before the assignment statement, earth and klingon refer to two different objects.



After the assignment statement, `earth` and `klingon` refer to the same object.



Variables of a Class Type



Assignment with Variables of a Class Type

- *Aliases*
  - Multiple class variables that have the same memory address
  - They point to the same object

```
Species mouse = new Species("Mouse", 10, 5);
Species cat = mouse;
Species lion = cat;

// lion and cat are aliases of mouse
```


Comparing Class Variables

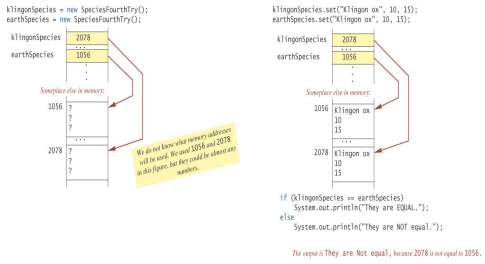
- A class type variable
  - *memory address* of the object
- Equality operator == with two class variables
  - the addresses of the objects are compared!
  - not the content of the objects
  - rarely what you want to do!
- Use the class's equals() method to compare the *content* of objects referenced by class variables

Example: Comparing Class Variables

```
//User enters first string
String firstLine = keyboard.nextLine();

//User enters second string
String secondLine = keyboard.nextLine();

if(firstLine == secondLine)
//this compares their addresses
{
   <body of if statement>
}

if(firstLine.equals(secondLine)
//this compares their values
{
   <body of if statement>
}
```

Use `equals()` method (not the double-equals sign) to compare class variables

== with Class Type Variables



Display 4.16

Dangers of == with Objects

== with Class Type Variables

```
import java.util.*;

public class Species
{
    private String name;
    private int population;
    private double growthRate;

    <The definition of the methods readInput, writeOutput, and projectedPopulation
            go here. They are the same as in Display 4.3 and Display 4.6.>

    <The definition of the methods set, getName, getPopulation,
            and getGrowthRate go here. They are the same as in Display 4.9.>

    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 and is automatically provided as part of the Java language.

Display 4.17
Defining an equals Method

== with Class Type Variables

```
public class SpeciesEqualsDemo
{
    public static void main(String[] args)
    {
        Species s1 = new Species(), s2 = new Species();

        s1.set("Klingon Ox", 10, 15);
        s2.set("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.set("klingon ox", 10, 15);
        if (s1.equals(s2))
            System.out.println("Still 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 we change one Klingon Ox to all lowercase.
Still match with the method equals.
```

Display 4.18
Demonstrating an equals Method

Programming Example

```
import java.util.*;

/**
 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 (percent 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 calling object
    after the specified number of years.
    */
    public int projectedPopulation(int years)
    {
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                        (growthRate/100) * populationAmount);
            count--;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }
}
```

This is the same class definition as in Display 4.17, but with all the details shown.

Display 4.19
A Complete Special Class

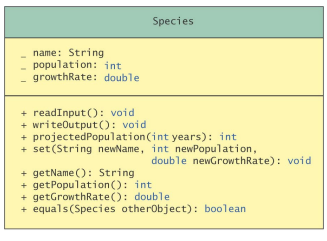
Programming Example, contd.

```
    public void set(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 Display 4.17. In Display 4.17, we explicitly used the this parameter. This version omits underneath to be there explicitly.

Display 4.19
A Complete Special Class

Programming Example, cont.



Display 4.20
Class Diagram for the Class Species in Display 4.19

Class Types as Method Parameters

- class variable names used as parameters in a method call
  - copy the ***address*** in the argument to the formal parameter
  - formal parameter name contains the address of the argument
  - the formal parameter name is an **alias** for the argument name

  Any action taken on the formal parameter is actually taken on the original argument!
- Different for parameters of primitive types
  - the original argument is not protected for class types!

Class Parameters, cont.

- Example

```
if (s1.equals(s2))
…
public boolean equals(Species otherObject)
```

causes `otherObject` to become an alias of `s2`, referring to the same memory location, which is equivalent to

```
otherObject = s2;
```


Example: Class Type as a Method Parameter

```
//Method definition with a DemoSpecies class parameter
public void makeEqual(DemoSpecies otherObject)
{
   otherObject.name = this.name;
   otherObject.population = this.population;
   otherObject.growthRate = this.growthRate;
}

//Method invocation
DemoSpecies s1 = new DemoSpecies("Crepek", 10, 20);
DemoSpecies s2 = new DemoSpecies();
s1.makeEqual(s2);  // s2 is changed!
```

- The method call makes `otherObject` an alias for `s2`, therefore the method acts on `s2`, the `DemoSpecies` object passed to the method!
- This is *unlike* primitive types, where the passed variable cannot be changed.

Comparing Class Parameters and Primitive-Type Parameters, cont.

```
import java.util.*;

/**
 This is a version of the class Species, but 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;

    /**
     Precondition: Calling object has been given values.
     Postcondition: otherObject has the same data as the
     calling object. The calling object is unchanged.
    */
    public void makeEqual(DemoSpecies otherObject)
    {
        otherObject.name = this.name;
        otherObject.population = this.population;
        otherObject.growthRate = this.growthRate;
    }

    /**
     Tries to set intVariable equal to the population of
     the calling object. But it cannot succeed, because
     arguments of a primitive type cannot be changed.
    */
    public void tryToMakeEqual(int intVariable)
    {
        intVariable = this.population;
    }

    public boolean equals(DemoSpecies otherObject)
    <The rest of the class definition of the method equals is the same as in Display 4.19.>

    <The rest of the class definition is the same as that of the class Species in Display 4.19.>
}
```

Display 4.21
Just a Demonstration Class

Comparing Class Parameters and Primitive-Type Parameters, cont.

```
public class ParametersDemo
{
    public static void main(String[] args)
    {
        DemoSpecies s1 = new DemoSpecies(),
                    s2 = new DemoSpecies();

        s1.set("Klingon Ox", 10, 15);
        s2.set("Ferengie Fur Ball", 90, 56);
        System.out.println("Value of s2 before call to method:");
        s2.writeOutput();
        s1.makeEqual(s2);
        System.out.println("Value of s2 after call to method:");
        s2.writeOutput();

        int aPopulation = 42;
        System.out.println(
                    "Value of aPopulation before call to method: "
                    + aPopulation);
        s1.tryToMakeEqual(aPopulation);
        System.out.println(
                    "Value of aPopulation after call to method: "
                    + aPopulation);
    }
}
```

Screen Output

```
Value of s2 before call to method:
Name = Ferengie Fur Ball
Population = 90
Growth Rate = 56.0%
Value of s2 after call to method:
Name = Klingon Ox
Population = 10
Growth Rate = 15.0%
Value of aPopulation before call to method: 42
Value of aPopulation after call to method: 42
```

An argument of a class type can change.

An argument of a primitive type cannot change.

Display 4.22
Comparing Parameters of a Class Type and a Primitive Type

Summary

- Classes and objects
- Attributes (instance variables) and methods
- Private attributes (instance variables), public methods
- Information hiding and encapsulation
- Class type variables have addresses of objects
- Issues with assignment, equality, and parameters