SENG1110/SENG6110
Object Oriented Programming

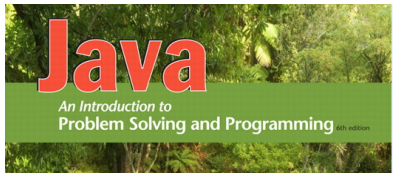
THE UNIVERSITY OF NEWCASTLE AUSTRALIA

FACULTY OF ENGINEERING AND BUILT ENVIRONMENT

www.newcastle.edu.au

Lecture 5

Classes and Methods – Part I



Outline

- Previously…
  - Loop statements
    - **while, for, do…while**
    - **break** and **continue**
  - Testing
- Now…
  - Review – example
  - IDEs - BlueJ
  - Class and method definitions/examples
  - Methods
    - Local variables
    - Parameters of primitive type
    - Information hiding (encapsulation)
    - Pre- and Postcondition comments
    - The `public` and `private` Modifiers
  - UML Class Diagrams
  - Variables of a Class Type
  - Defining an `equals` Method for a Class
  - Parameters of a Class Type

Example – population

- Write a Java code that calculates and prints the population of a country in the beginning of each year between two input years a and b.
- Every year the population increases by x%.
- The inputs are the starting population, the year a, the year b and the value of x
- x must be a number greater than 0. If not, ask the user again.
- Check if a<b. If not swap the values

Example – population


```
input a, b, pop

x = -1;
while (x<0) input x
x/=100;

if (a>b){
   aux=a;
   a=b;
   b=aux;
   }

for(i=a; i<=b; i++) {
   print pop
   pop*=(1+x);
}
```

Example – population


```
import java.util.*;
public class Population
{
  public static void main (String[] args)
  {
   Scanner console =new Scanner(System.in);
   int a, b;
   double x=-1, pop;
   System.out.print("year a: "); a = console.nextInt();
   System.out.print("year b: "); b = console.nextInt();
   System.out.print("initial pop: "); pop = console.nextDouble();

   while(x<0) {
      System.out.print("x>=0: ");
      x = console.nextDouble()/100;
   }
   if (a>b){
        int aux=a;
        a=b;
        b=aux;
        }
   for(int i=a; i<=b; i++) {
        System.out.print("Pop in the year"+i+" = "+Pop);
        pop*=(1+x);
   }
}
```


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

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

IDEs – BlueJ

- IDE = Integrated Development Environment
  - BlueJ (we will use this one the the computer labs)
    - https://www.bluej.org
  - Eclipse
    - https://eclipse.org
  - Netbeans
    - https://netbeans.org
  - Check the first video available in this week's lab

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Class and Method Definitions

- Figure 5.1 A class as a blueprint

Class Name: Automobile

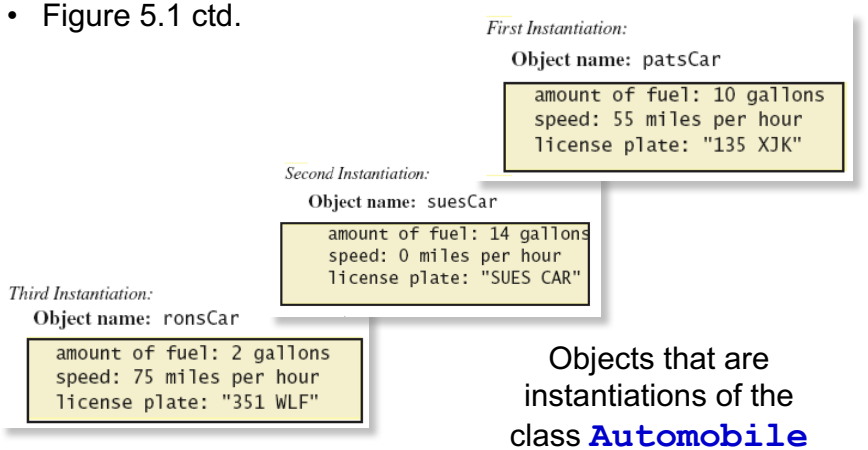
Data:
amount of fuel_____
speed _____
license plate _____

Methods (actions):
accelerate:
How: Press on gas pedal.
decelerate:
How: Press on brake pedal.

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Class and Method Definitions

- Figure 5.1 ctd.



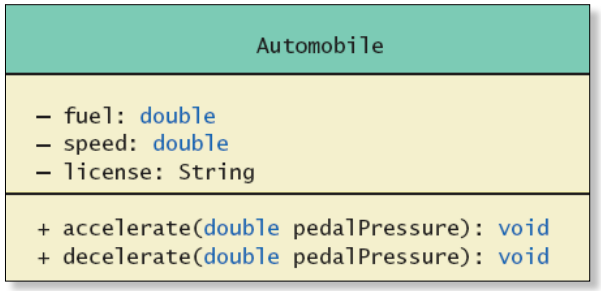
Objects that are instantiations of the class **`Automobile`**

Class Files

- 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

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram



`Dog` class - instance variables

- **`class Dog`**
  - Three pieces of data (instance variables)
    - name
    - breed
    - age
  - Two behaviors
    - writeOutput
    - getAgeInHumanYears

```
public class Dog
{
   public String name;
   public String breed;
   public int age;

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

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

Data
Instances variables

Method
writeOutput

Method
getAgeinHumanYears


```
public class DogDemo
{
   public static void main(String[] args)
   {
         Dog dog1 = new Dog();

         dog1.name = "Balto";
         dog1.age = 8;
         dog1.breed = "Siberian Husky";
         dog1.writeOutput();

         Dog dog2 = new Dog();

         dog2.name = "Scooby";
         dog2.age = 42;
         dog2.breed = "Great Dane";
         System.out.println(dog2.name + " is a " + dog2.breed + ".");
         System.out.print("He is " + dog2.age + " years old, or ");
         int humanYears = dog2.getAgeInHumanYears();
         System.out.println(humanYears + " in human years.");
    }
}
```


Dog class - using a class and its methods

- `class DogDemo`

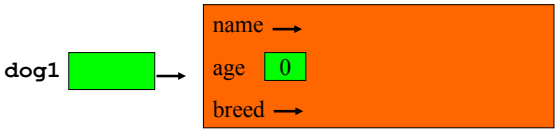
```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

Sample screen output

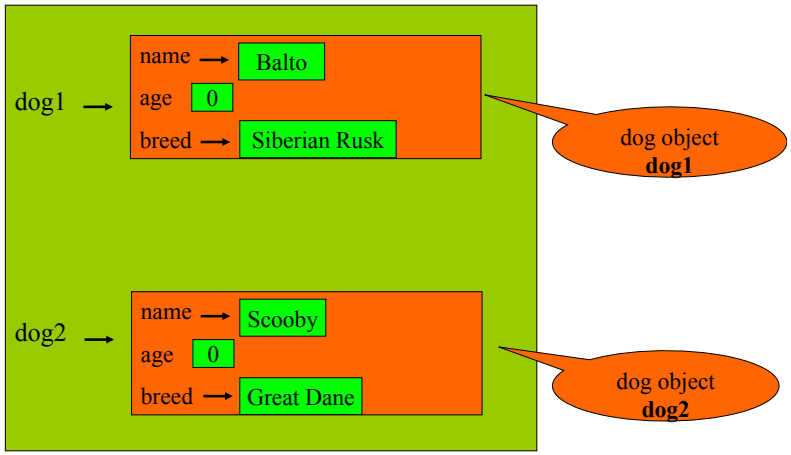
new

- `Dog dog1;`

- `dog1 = new Dog();`



`DogDemo` class



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Defining **`void`** Methods

- Consider the method **`writeOutput`**

```
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

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

Species example

- Class designed to hold records of endangered species (View codeSamplesWeek5)
- class **`SpeciesFirstTry`**
  - Three instance variables:
    - name
    - population
    - growthRate
  - three methods
    - readInput
    - writeOutput
    - getPopulationIn10
- class **`SpeciesFirstTryDemo`**

Methods That Return a Value

- Consider method **`getAgeInHumanYears( )`**

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

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

Local Variables

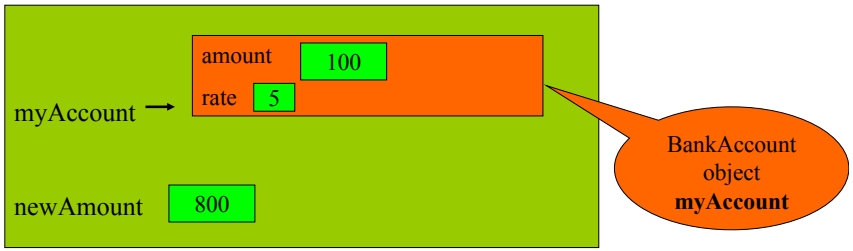
- Variables declared inside a method are called *local* variables
  - May be used only inside the method
  - All variables declared in method **`main`** are local to **`main`**
- Local variables having the same name and declared in different methods are different variables

Local Variables – Example BankAccount

```
public class BankAccount
{
   public double amount;
   public double rate;

   public void showNewBalance ()
   {
        double newAmount = amount + (rate / 100.0) * amount;
        System.out.println ("With interest added, the new
                             amount is $" + newAmount);
   }
}
```

Local variable

Local Variables – Example BankAccount



Local Variables – Example BankAccount


```
public class LocalVariablesDemoProgram
{
   public static void main (String [] args)
   {
        BankAccount myAccount = new BankAccount ();

        myAccount.amount = 100.00;
        myAccount.rate = 5;
        double newAmount = 800.00;

        myAccount.showNewBalance ();

        System.out.println("I wish my new amount were $" +newAmount);
   }
}
```


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

- Recall method in CodeSamplesWeek5 - **SpeciesFirstTry**
  - Note it only works for 10 years
  - We can make it more versatile by giving the method a parameter to specify how many years

```
public int getPopulationIn10 ()
{
   int result = 0;
   double populationAmount = population;
   int count = 10;
   while ((count > 0) && (populationAmount > 0))
   {
     populationAmount = populationAmount +(growthRate/100)*populationAmount;
     count - - ;
   }
   if (populationAmount > 0)
     result = (int) populationAmount;
   return result;
}
```


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
- See CodeSamplesWeek5 – **SpeciesSecondClassDemo**

Parameters of Primitive Type

- See CodeSamplesWeek5 - **SpeciesSecondTry**

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

Information Hiding - encapsulation

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

Pre- and Postcondition Comments

- 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)
```


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()
```


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`**
- View CodeSamplesWeek5 – **`SpeciesThirdTry`**

Private – Example Rectagle


```
public class Rectangle
{
   private int width;
   private int height;

   public void setDimensions (int newWidth, int newHeight)
   {
         width = newWidth;
         height = newHeight;
   }

   public int getArea ()
   {
         return width * height;
   }
}
```


Accessor and Mutator Methods

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

Private – Example Rectagle

- Suppose we declare

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

box.height = 5        is illegal
```

- The only way the **`width`** and **`height`** may be altered outside the class is using the method **`setDimensions()`**

Accessor and Mutator Methods

- Consider an example class with accessor and mutator methods
- View CodeSamplesWeek5 – **`SpeciesFourthTry`**
- Note the mutator method
  - **`setSpecies`**
- Note accessor methods
  - **`getName`**, **`getPopulation`**, **`getGrowthRate`**

Accessor and Mutator Methods

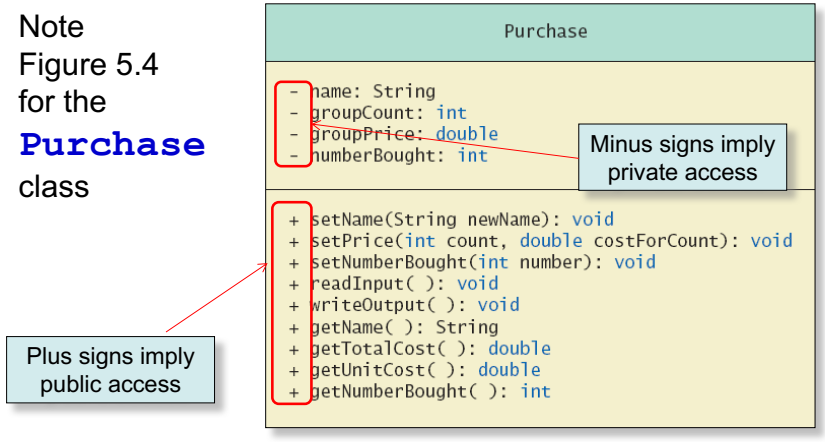
- Using a mutator method
- View CodeSamplesWeek5 – **`SpeciesFourthTryDemo`**

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

Sample screen output

UML Class Diagrams

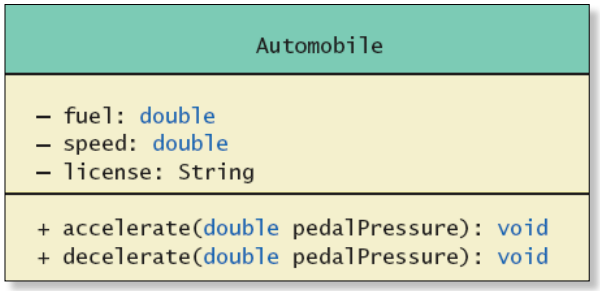
- Note Figure 5.4 for the **`Purchase`** class



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

UML Class Diagrams

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



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

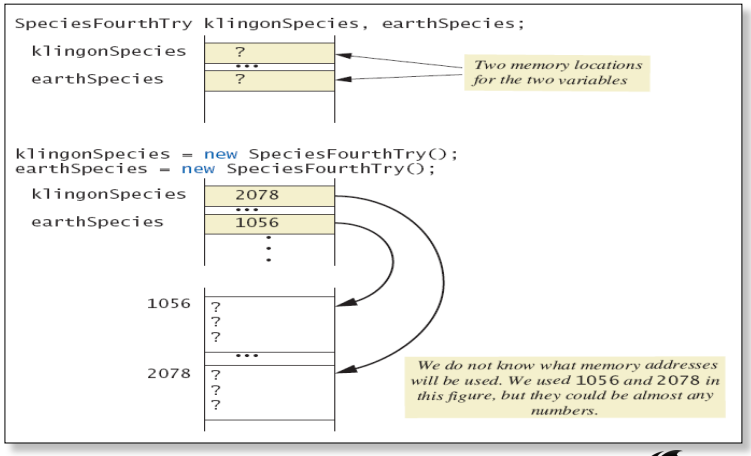
THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

- Figure 5.5a – Behavior of class variables

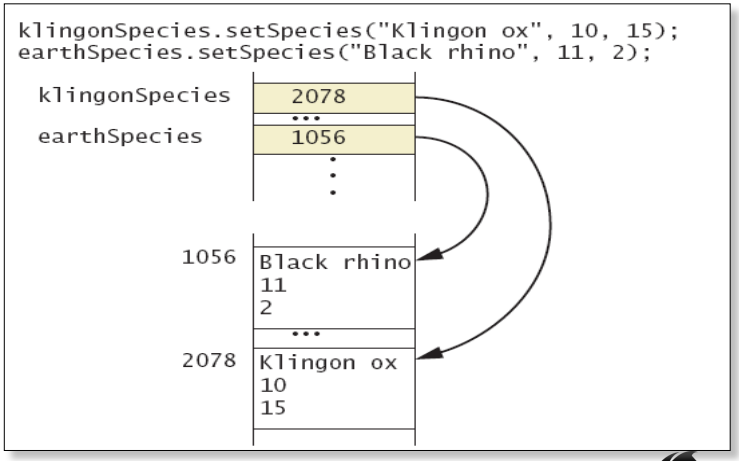


Variables of a Class Type

- 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

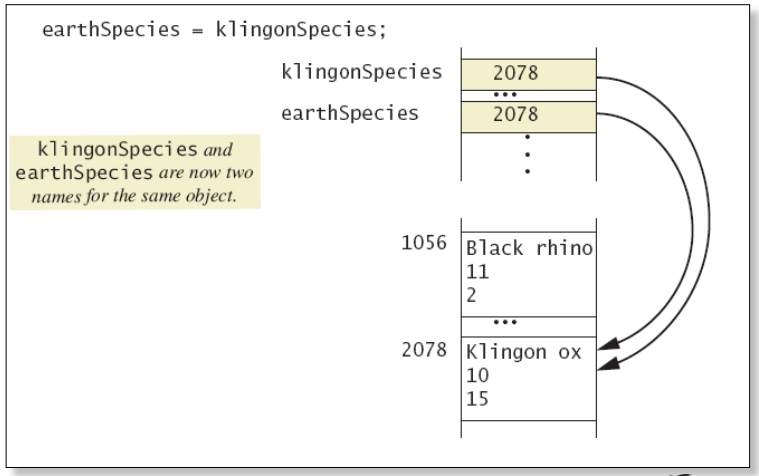
Variables of a Class Type

- Figure 5.5b – Behavior of class variables



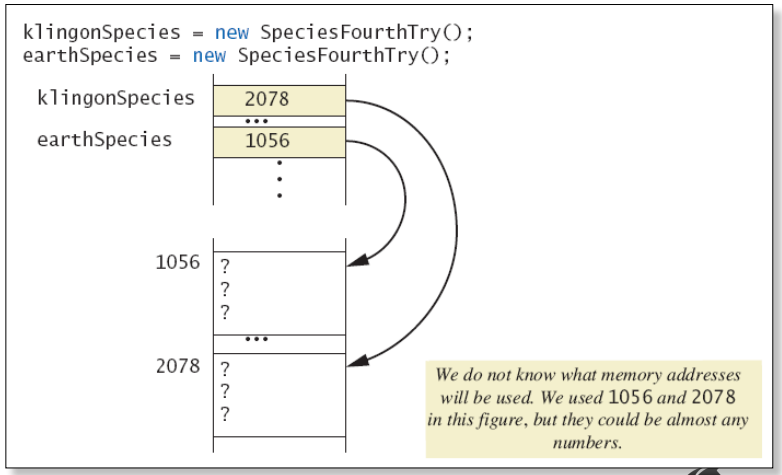
Variables of a Class Type

- Figure 5.5c – Behavior of class variables



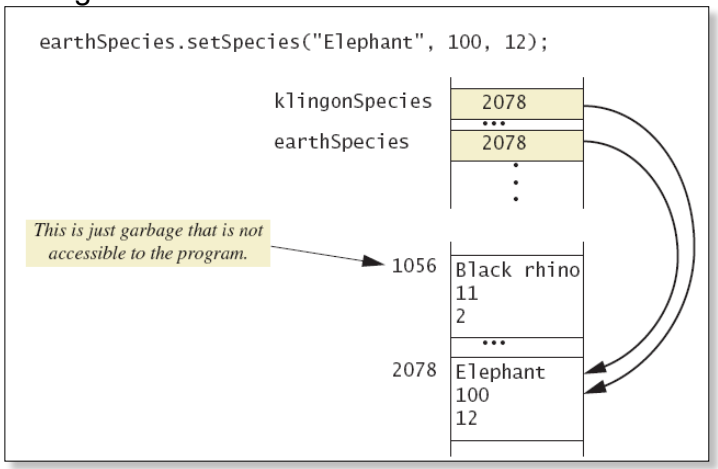
Variables of a Class Type

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



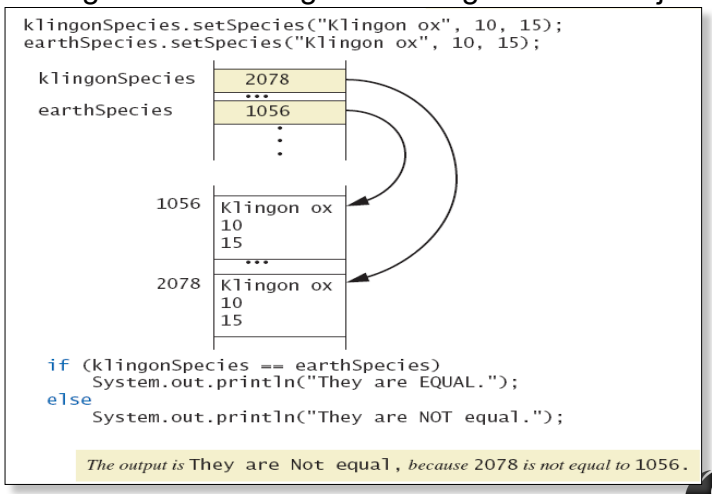
Variables of a Class Type

- Figure 5.5d – Behavior of class variables



Variables of a Class Type

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



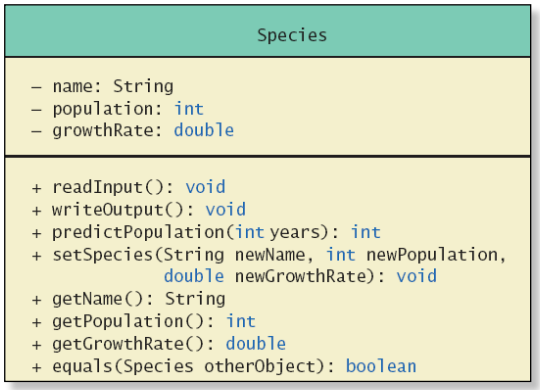
Defining an `equals` Method

- 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

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


Complete Programming Example

- View CodeSamplesWeek5 – `Species`
- Figure 5.7 - Class Diagram for the class `Species`



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Using an `equals` Method


```
public class SpeciesEqualsDemo
{
   public static void main (String [] args)
   {
     Species s1 = new Species (), s2 = new Species ();
     s1.setSpecies ("Klingon ox", 10, 15);
     s2.setSpecies ("Klingon ox", 10, 15);

     if (s1 == s2) System.out.println ("Match with ==.");
     else          System.out.println ("Do Not match with ==.");

     if (s1.equals (s2)) System.out.println("Match with the method equals.");
     else  System.out.println ("Do Not match with the method equals.");

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


THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Boolean-Valued Methods

- Methods can return a value of type `boolean`
- Use a `boolean` value in the `return` statement

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

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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 CodeSamplesWeek5 – `SpeciesTest`

Your task

- Read
  - Chapter 5 of the text book

- Exercises
  - MyProgrammingLab
  - Implement/compile/run the examples from lecture slides (copy from codeSamplesWeek5 – available in Blackboard)
  - Complete the lab exercises

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