Murdoch
UNIVERSITY

Topic 3
Basic of OOP

ICT167 Principles of Computer Science

Murdoch
UNIVERSITY

Objectives

- Define and appreciate the use of **information hiding**
- Use **pre-conditions** and **post-conditions** correctly
- Understand the use of **assertions** in programs
- Only use **private** instance variables
- Use **accessor** methods and **mutator** methods correctly

Objectives

- Define Abstract Data Types (ADT), user interfaces, and ADT implementation
- Define data abstraction and encapsulation
- Use javadoc for program document purposes
- Know how to change a class implementation
- Use **new** correctly
- Understand assignment with class type variables
- Explain references and memory addresses

Objectives

- Use **==** with class types
- Define a class **equality** test
- Use parameters of class type
- Understand the **null** reference

Reading

Savitch: Chapters 5.2, 5.3

Information Hiding

- Information hiding:
  - Involves designing a method so that in order to use it, a client does not need to look at the code in the method body
    - A comment at the beginning of the method should tell the client what the method does
  - Allows client to understand the ***what*** (i.e. what the method does) without worrying about the ***how*** (i.e. how the method does what it does)
  - Is related to abstraction

Information Hiding

- Allows a team of implementers to easily divide up their work
- Requires a good clear description of what the method is supposed to do
- Allows implementers to make better, more efficient implementations even after clients have started using the class:
  - The client does not need to change her/his program just because the body of the method is changed
- The same idea of information hiding applies to whole classes as well as methods

Pre- + Post-Condition Comments

- The client (user of a method of a class) will want to know:
  - the name
  - return type
  - Number, type and order of parameters of the method
- They will also want a clear, precise and complete description of what it is supposed to do
  - Eg: see the on-line documentation for the library class methods (eg: those of String class)

Pre- + Post-Condition Comments

- It is very common to present these comments in the form of a **contract** between the creator of a class and the user (client) of the class:
    - The client supplies some arguments satisfying certain conditions, the ***pre-conditions***
    - The pre-condition for a method states the conditions that must be true <u>before</u> the method is invoked
    - The creator promises to bring about some changes to the calling object and/or some properties of the return value, the ***post-conditions***

Pre- + Post-Condition Comments

- The post-condition describes the effect of the method call. That is, it tells what will be true after the method is executed

- Eg: this is all the client needs to know ...

```
/**
Pre-condition: years is a non-negative number
Post-condition: Returns the projected
population of the calling (receiving) object
after the specified number of years.
*/
public int predictPopulation(int years)
```

Pre- + Post-Condition Comments

- According to the contract, if the pre-condition is satisfied, the creator of the class guarantees that the post-condition will be satisfied

Assertions

- An assertion is a statement about the state of the program:
  - It can be true or false
  - It should be true when there are no mistakes in running the program
- Pre-condition and post-condition comments are examples of assertions

Assertions

- Assertions can occur anywhere in programs, such as after a block ({…})
  - A check can be inserted to determine if an assertion is true and, if not, to stop the program and output an error message
- An *assertion check* has the following form:

```
assert Boolean_Expression;
```

- If the `Boolean_Expression` evaluates to false, the program ends and outputs an error message saying that an assertion failed

Assertions

- Eg:

```
assert n == 1;
while (n < limit) {
    n = 2 * n;
}
assert n >= limit;
```

- // n is the smallest power of 2 >= limit
- Assertion checking can be turned on or off
  - They can be turned on during the program testing stage so that a failed assertion will stop the program and display an error message

Assertions

- A class containing assertions would need to be compiled differently. Eg:

```
javac -source 1.8 MyProg.java
```

- To run a program with assertion checking turned on, use:

```
java -enableassertions MyProg
```

- The normal way of running the program has assertion checking turned off – to make it run more efficiently
- Check how to do this in NetBeans?

Making Instance Variables Private

- Recall that if instance variables are declared public in a class, such as:

```
public String name;
```

- in the class `SpeciesFirstTry`, then the client can directly access the instance variable. Eg:

```
speciesOfTheMonth.name = ”Klingon ox”;
```

- The modifier ***public*** means that any other class/program can directly access/change the instance variable

Making Instance Variables Private

- Allowing direct access to instance variables is bad for information hiding
  - The instance variables are the real substance of a class and once clients are using them, they can not be changed by the implementer
- Therefore, HIDE THEM
- This is achieved by declaring instance variables using the **private** modifier
- An example soon, but first …

Making Instance Variables Private

- An analogy:
  - An ATM permits deposits and withdrawals, both of which affect the account balance however, it does not permit an account balance to be accessed and changed **directly**
  - If an account balance could be accessed and changed directly, a bank would be at the mercy of ignorant and unscrupulous users

Making Instance Variables Private

- For example, in `SpeciesThirdTry` class:

```
private String name;
private int population;
private double growthRate;
```

- Then, they can still be used inside the class `SpeciesThirdTry` but not by its clients, like `SpeciesThirdTryDemo`

Making Instance Variables Private

- If an instance variable (or method) is declared to be ***private*** inside a class then it can not be directly referred to by name outside its class definition
- Normally:
  - All instance variables are marked **private**
  - All methods are marked public (except for helper methods - see later)

Accessor and Mutator Methods

- An ***accessor*** method is a method that accesses an object and returns some information about it, without changing the state of that object (its instance variables)
- Eg: the following methods of the class `SpeciesFourthTry` are accessor methods:
  - `writeOutput()`, `predictPopulation(int years)`, `getName()`, `getPopulation()`, `getGrowthRate()`
- Accessor methods are also called ***get methods*** or ***getters***

Accessor and Mutator Methods

- A ***mutator*** method is a method that modifies the state of an object
- Eg: the `SpeciesFourthTry` class methods

```
void readInput() and

void setSpecies(String newName,
       int newPopulation, double
                  newGrowthRate)
```

- are mutator methods

Accessor and Mutator Methods

- As a rule of thumb, it is best to separate accessors and mutators:
    - If a method returns a value to the client program then it should not modify the object
    - Conversely, mutators should have a return type of void

Accessor and Mutator Methods

- Since instance variables (i.e. the state of an object) do need to be looked at, initialized or changed, this can be achieved as follows:
  1. If the value of an instance variable may be needed by a client then supply a **public** "get" method (accessor method) for the variable
  2. If the value of an instance variable might need to be initialized or changed by a client then supply a **public** "set" method (mutator method) for the variable, or if convenient, for a whole lot of instance variables at once

Accessor and Mutator Methods

- Advantages of this design:
  - Inside a set method you can check whether the new value is legitimate
  - Some variables may not be allowed to be set by clients
  - Some groups of variables may have to be updated together
  - The implementer can even change the real instance variables and keep the old set and get methods to make the class work the same for clients

Immutable Classes

- Some classes have been designed to have only accessor methods and no mutator methods at all
- These are called ***immutable*** classes
- Eg: the String class:
  - once a string object has been constructed, its contents (state) never change
  - no method in the String class can modify the contents of a string
  - The **StringBuffer** class is available for modification

Example Class

```
import java.util.Scanner;
public class SpeciesFourthTry {
  private String name;
  private int population;
  private double growthRate;
  public void writeOutput() {
     System.out.println("Name = " + name);
     System.out.println("Population = " +
                                       population);
     System.out.println("Growth rate = " +
                                   growthRate + "%");
  }
```

Example Class

```
public void readInput() {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("What species' name?");
    name = keyboard.nextLine( );
    System.out.println("Population of species?");
    population = keyboard.nextInt( );
    while (population < 0) {
        System.out.println("No negative population");
        System.out.println("Re-enter population:");
        population = keyboard.nextInt( );
    }
    System.out.println("Growth rate (%/yr):");
    growthRate = keyboard.nextDouble( );
}
```

Example Class

```
/** Pre-condition: years is a nonnegative number.
Post-condition: Returns projected population of
calling object after specified number of years.   */
public int predictPopulation(int years) {
   int result = 0;
   double populationAmount = population;
   int count = years;
   while ((count >0)&&(populationAmount >0)) {
      populationAmount = (populationAmount +
                  (growthRate/100) * populationAmount);
      count--;
   }
   if (populationAmount > 0)
      result = (int)populationAmount;
   return result;
}
```

Example Class

```
public void setSpecies(String newName,
   int newPopulation, double newGrowthRate) {
   name = newName;
   if (newPopulation >= 0)
      population = newPopulation;
   else
   {
      System.out.println("ERROR: negative
                              population.");
      System.exit(0);
   }
   growthRate = newGrowthRate;
}
```

Example Class

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

Example Client

```
import java.util.*;
/** Demonstrates the use of the mutator method
   setSpecies*/
public class SpeciesFourthTryDemo {
  public static void main(String[] args) {
     SpeciesFourthTry speciesOfTheMonth =
                                new SpeciesFourthTry();
     int numYears, futurePopulation;
     System.out.println("No. of years to project:");
     Scanner keyboard = new Scanner(System.in);
     numYears = keyboard.nextInt( );
     System.out.println("Enter data on the Species
                       of the Month:");
     speciesOfTheMonth.readInput();
```

Murdoch
UNIVERSITY

Example Client

```
        speciesOfTheMonth.writeOutput();
        futurePopulation =
        speciesOfTheMonth.predictPopulation(numYears);
        System.out.println("In " + numYears + " years
                                 the population will be " +
                                 futurePopulation);
        speciesOfTheMonth.setSpecies("Klingon ox",
                              10,15);
        System.out.println("The new Species of the
                              Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In " + numYears + " years
                        the   population will be " +
      speciesOfTheMonth.predictPopulation(numberOfYears));
       }//end main
   }//end class
```

Abstract Data Types (ADTs)

- A **data type** is a set of values together with a collection of operations that can be performed on these values
- An **abstract data type** (ADT) is a data type defined so that the clients who use the type do not have access to the details of how the values and operations are implemented

Abstract Data Types (ADTs)

- An ADT or a class can be divided into two parts:
  - **Class interface:**
    - Defines what the client of a class needs to know in order to use the class
    - A client of a class just needs to know its name and about what public methods it has available
    - This is (loosely) called the user ***interface*** of the class

Abstract Data Types (ADTs)

- An ADT or a class can be divided into two parts:
  - **Class implementation:**
    - Tells how the class interface is realised as Java code
    - All the details like (private) instance variables, private methods and all method bodies can be kept hidden (to avoid confusion and allow for changes etc)
    - This is the ***implementation*** of the class

Abstract Data Types (ADTs)

- For example, you have been clients of the String class without knowing its implementation details
- Rival teams of developers can supply several versions of common and useful classes (eg: String), and, provided the same methods are available and do the same things, then we think of all of these as the same class

Abstract Data Types (ADTs)

- To be more correct, we use the term Abstract Data Type to refer to a class of Objects with certain standard public methods available
- Eg: we might say that the String ADT is implemented in many ways, some more efficient than others

Guidelines for Making Class Definitions into ADTs

- Place a comment before the class definition that describes what the class does, and how the client should think about the class data and methods
- Declare the instance variables in the class as ***private***
- Provide ***public*** accessor and/or mutator methods to read data and output data/results (eg: getter and setter methods)

Guidelines for Making Class Definitions into ADTs

- Also provide basic methods that a client needs to manipulate data in an object
- Specify how to use each public method with a comment placed before the method heading
- Make any helper methods ***private***

Encapsulation + Data Abstraction

- **Data abstraction** = lumping a bunch of related values together and calling it by one name
  - Eg: three related values to do with Species are put together as one `SpeciesFourthTry` object
- **Encapsulation** = lumping related data values and actions (i.e. methods) together in one item
  - Eg: making a class to deal with the three Species data values and all related methods

Murdoch UNIVERSITY

Encapsulation + Data Abstraction

- If implementers follow these ideas then one programmer can easily manage whole lists of species without troubling themselves with the details within an individual species
- Another team can deal with details at the species level

Encapsulation + Data Abstraction

- Another example:
  - One team implements methods to do with enrolment and personal details of individual *students*
  - Another team implements methods to do with enrolment and results etc., of all students in a particular *unit*
  - Another team implements methods to do with organizing the rooms and exams for all the units on *campus*

Javadoc

- The **javadoc** program (supplied with the JDK) automatically produces an HTML document that describes your class in a form extremely useful for client users of the class (i.e. with private implementation details hidden)

Javadoc

- The implementer of the class just needs to follow two simple rules about the internal comments which they want to be picked up by javadoc:
  - Put the comment immediately before a public class definition or a public method definition (or other public item)
  - start the comment with /** and end with */

Javadoc

- Then type:

  ```
  javadoc MyClassName.java
  ```

  in the directory in which `MyClassName.java` resides; this will create a new file `MyClassName.html`
- Try it with `SpeciesFourthTry` class or your own class

Javadoc

- You can also javadoc all classes in a NetBeans project by selecting Run|Generate Javadoc from the NetBeans tool bar
- There are plenty of features for more sophisticated use:
  - See Appendix 5 of Savitch for further details
- You should use javadoc for your assignment classes

Changing the Implementation

- One of the most important reasons for information hiding is to allow implementers to improve their implementation even after clients have started using it
- See the case study on pages 350-354 (*page numbers could be different depending of the edition*) of the textbook – A **Purchase class**

Objects and References

- The way Java handles class type values, as opposed to primitive values, affects:
  - The use of **new**
  - Assignment of class type variables
  - References and memory addresses
  - Testing for equality
  - Class type parameters

Objects and References

- We already know that to create an object of class type we need to use the keyword **new.** Eg:

```
SpeciesFourthTry speciesOfTheMonth = new
                                  SpeciesFourthTry();
```

- This sets aside just enough memory (in a special area of memory called the heap) to store a `SpeciesFourthTry` object with all its data values (the values of its instance variables)
- It also stores the memory address of this area in the memory location used by the variable `speciesOfTheMonth`

Objects and References

- We say `speciesOfTheMonth` is a *reference* to this new object
- Thus an ***object reference*** is information on how to find a particular object
  - The object is a chunk of main memory; a reference to the object is a way to get to that chunk of memory
  - The variable `speciesOfTheMonth` does not actually contain the object, but contains information about where the object is

Assignment with Class Type Variables

- Compare the output from the two experiments in the following program. They each involve an assignment:
  - one of primitive type
  - one of class type variables
- The difference in output is explained by the difference in behaviour of the assignment operator

Assignment with Class Type Variables

```
n = m;
```
// n and m are primitive variables

- This just means that the value currently stored in m is also put in n
  - There is no lasting association between the two memory locations
- However, both of these are class variables

```
EarthSpecies = klingonSpecies;
```

- This results in the memory address currently being stored in `klingonSpecies` being also stored in `EarthSpecies`

Assignment with Class Type Variables

- So now the two variables refer to the same Object
- If we then make a change to that one Object via the `setSpecies` method, it should not be surprising that the two output calls print out exactly the same information

Example

```
public class AssignDemo {
  public static void main(String[] args){
     System.out.println("Experiment 1: primitives:");
     int n, m;
     n = 42;
     m = n;
     n = 99;
     System.out.println( n + " and " + m );
```

Example

```
     System.out.println("Experiment 2: class
                       types:");
     SpeciesFourthTry klingonSpecies, earthSpecies;
     klingonSpecies = new SpeciesFourthTry();
     earthSpecies = new SpeciesFourthTry();
     klingonSpecies.setSpecies("Klingon ox",10,15);
     earthSpecies = klingonSpecies;
     earthSpecies.setSpecies("Elephant",100,12);
     System.out.println("earthSpecies:");
     earthSpecies.writeOutput();
     System.out.println("klingonSpecies:");
     klingonSpecies.writeOutput();
  }//end of main
}//end of class
```

Example

```
/* OUTPUT
Experiment 1: primitives
99 and 42
Experiment 2: class types
earthSpecies:
Name = Elephant
Population = 100
Growth rate = 12.0%
klingonSpecies:
Name = Elephant
Population = 100
Growth rate = 12.0%
*/
```

New Values in Reference Variables

- Consider the following example:

```
// File: referenceString.java
class referenceString {
  public static void main ( String[] args ) {
     String myStr;
     myStr = new String("Computer Science");
     System.out.println(myStr);
     myStr = new String("Games Technology");
     System.out.println(myStr);
  }// end of main
}// end of class referenceString
```

New Values in Reference Variables

- The above program will, as expected, write out

```
Computer Science
Games Technology
```

- However, consider some of the details involved:
- The statement

```
myStr = new String("Computer Science");
```

  - creates the **first** object, and
  - Puts a reference to this object into `myStr`

New Values in Reference Variables

- The statement

  ```
  System.out.println(myStr);
  ```

  - Follows the reference in `myStr` to the **first** object
  - Gets the data in the **first** object and prints it
- The statement

  ```
  myStr = new String("Games Technology");
  ```

  - Creates a **second** object
  - Puts a reference to the **second** object into `myStr`

New Values in Reference Variables

- **At this point there is no reference to the first object - it is now "garbage "**
  - This is a commonly occurring situation in Java, and not a mistake
- As the program runs, a part of the Java system called the "garbage collector" reclaims the lost objects (the "garbage") so that their memory can be used again

New Values in Reference Variables

- The statement

  `System.out.println(mySTr);`

  - Follows the reference in `myStr` to the **second** object
  - Gets the data in the **second** object and prints it

New Values in Reference Variables

- Thus:
  - Each time the **new** operator is used, a new object is created
  - Each time an object is created, a reference to it is saved in a variable
  - The reference in the variable is later used to find the object
  - If another reference is saved in that variable, it replaces the previous reference
  - If no variable holds a reference to an object, the object becomes "garbage"

Testing for Equality of Class Variables

- Be careful of using the comparison operator == to test for equality between class type variables
- The test will only return true if the two variables both refer to exactly the same Object
- It is possible to have a **different** Object with the same values

Testing for Equality of Class Variables

- Eg:

```
SpeciesFourthTry eS= new SpeciesFourthTry();
SpeciesFourthTry kS= new SpeciesFourthTry();
kS.setSpecies(“Klingon ox”, 10, 15);
eS.setSpecies(“Klingon ox”, 10, 15);
if (eS == kS) System.out.println(“EQUAL”);
else System.out.println(“Not EQUAL”);
```

Testing for Equality of Class Variables

- Here the output will be “Not EQUAL” and thus the test of equality will fail
- Would you want to count these two objects as being equal? Probably.
- What about:

```
“Klingon ox”, 10, 15 and
“klingon ox”, 10, 15
```

- What about:

```
“Klingon ox”, 10, 15 and
“Klingon ox”, 12, 15
```

Defining Your Own “Equals”

- Many classes usefully need a test of equality
- Exactly what counts as equal should be defined by the implementer of the class
- Often they supply an equals method

Defining Your Own “Equals”

- Eg:

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

- This allows for differences in (upper/lower) case in the species name
- This might be used by a client in a test such as:

```
if (klingonSpecies.equals(earthSpecies))
     System.out.println(“EQUAL”);
else System.out.println(“Not EQUAL”);
```

Class Parameters

- Passing a class type argument to a method may change the argument (cf primitive types)
- In general it is not good design to allow this to happen as it may surprise the client but sometimes it has a use

Class Parameters

- Eg:

```
public void makeEqual
(SpeciesFourthTry otherObject){
  otherObject.name=name;
  otherObject.population=population;
  otherObject.growthrate=growthRate;
}
```

- Call by:

```
klingonSpecies.makeEqual(earthSpecies);
```

Class Parameters

- This still allows `earthSpecies` to refer to a different Object in memory but it changes all the data values of `earthSpecies` to be the same as those for `klingonSpecies`
- Basically the formal parameter `otherObject` is given (by the call) a ***reference*** to the `earthSpecies` Object and so is able to change it
- You may hear that Java uses ***call-by-reference*** for parameter passing of class type variables

The `null` references

- Note that sometimes a class type variable will refer to no object, especially if it has just been declared and not made to refer to a new object
- You can use `null` to initialize any class type variable to refer to nothing if you don't need particular object. Eg:

```
String line = null;
```

- You might get a `NullPointerException` if you try to call a method on a variable which refers to no Object

The **`null`** references

- You can test for `null`-ness via

```
if (line == null) ...
```

- You can test for non-`null`-ness via

```
if (line != null) ...
```

The `null` references

- **Another Example:**

```
String month = "August";
String year = "";          // empty string
// refers to no string at all
String message = null;
int len1 = month.length();   // returns 6
int len2 = year.length();    // returns 0
// runtime error
int len3 = message.length();
```

- Note that empty string and a `null` reference are different

End of Topic 3