COMP 202

More on OO

CONTENTS:
- static revisited
- this reference
- class dependencies
- method parameters
- variable scope
- method overloading

Static member variables

- So far: Member variables were instance variables
  - Every object instance has its own set of these variables
- Member variables can also be static
  - static variables are also called class variables
  - Shared by all objects/instances of a class
  - i.e., there is only one copy of this variable
  - **If one object changes the value of a static variable, all other objects see and access the new value**

```
private static int count = 0;
```

  - Example use:
    - Increase count whenever the Constructor is called
    - Keeps track of the number of objects created;

Static methods

- Static methods can be executed without an object of the class instantiated
- Static methods
  - can not access member variables because they always belong to a specific object/instance
  - can access static variables and local variables
- All methods of Math class are static
  - Math.abs(-5);
- `main` method is static

Calling Methods

- A method calls a method m1 of the same class:
  - Only indicate method name: `m1(…);`
- A method calls a static method m1 of another class AnotherClass:
  - `AnotherClass.m1();`
- A method calls a non-static m1 of another object which is referred to by the variable objVariable
  - `objVariable.m1();`

The calculator as a class with only static methods

```
import java.util.Scanner;

public class CalcMain {
   public static void main(String[] args) {
      int op;
      double v1,v2,result;
      scan = new Scanner(System.in);
      System.out.println("Enter 0 to exit, 1 to add, 2 to subtract");
      op=scan.nextInt();
      if (op==1 || op==2) {
          System.out.print("Enter number: ");
          v1=scan.nextDouble();
          System.out.print("Enter number: ");
          v2=scan.nextDouble();
          if (op==1)
             result = CalcMain.add(v1,v2);
          else
             result = subtract(v1,v2);
          System.out.println("Result: "+result);
      }
   }
```


The calculator as a class with only static methods

```
   public static double add(double d1,double d2) {
      return d1+d2;
   }
   public static double subtract(double d1,double d2) {
      return d1-d2;
   }
   // etc
}
```


Class dependencies

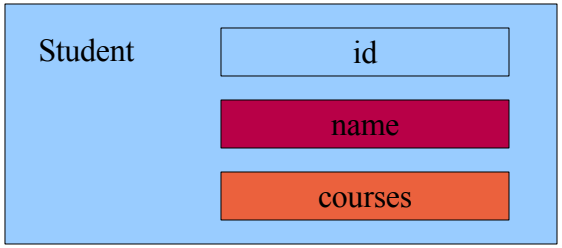
- Stand-alone
  - Classes that do not depend or need other classes to run
- Dependencies
  - When a program calls methods of other classes and/or instantiates objects of other classes.
  - eg. Bank uses Client class
  - eg. use of String and Scanner classes
- Aggregate Classes
  - A class that is composed from multiple other classes

Aggregate Classes

- A class that defines an object as one of its class variables has created a dependency between the class they are building and the object variable
- Aggregate classes are classes that use other classes as variables, these can be:
  - Built-in library classes (e.g. String)
  - Your own classes
  - Purchased / downloaded library classes

Student example

```
public class Student {
   private long id;
   private String name;
   private ArrayList courses;
   public Student(String name,long id) {
      this.name =name;
      this.id = id;
      courses = new ArrayList();
   }
}
```



Parameter Passing

- Parameters in a Java method are *passed by value*
- This means that *a copy* of the actual parameter (the value passed in) is stored into the formal parameter (in the method header) – i.e. not the real/original value
- Changing the local copy does not affect the original
- Not true of objects: when an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other

ParamPassTest.java

```
public class ParamPassTest {
        public static void main (String args[]) {
                int param = 3;
                addOne(param);
                System.out.println("main param is " + param);
        }
        public static void addOne(int p) {
                p = p + 1;
                System.out.println("addOne param is " + p);
        }
}
```

Example of a standard copy pass of an integer parameter:
How does it look in memory while executing?

Assignment Revisited

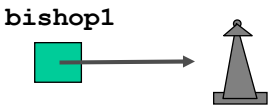
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

```
num2 = num1;
```

Object References

- Recall that an object reference holds the memory address of an object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a "pointer" to an object

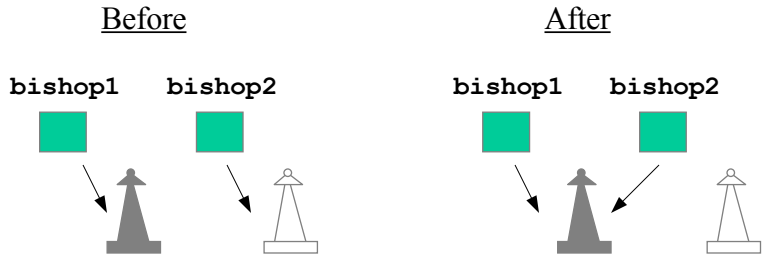
```
ChessPiece bishop1 = new ChessPiece();
```



Reference Assignment

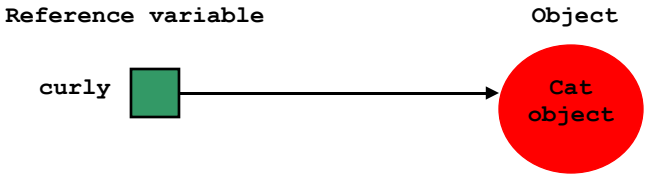
- For object references, assignment copies the memory location:

```
bishop2 = bishop1;
```



Parameter Passing with Objects

- In a Java statement such as `Cat curly = new Cat();` the variable `curly` is not an object, it is simply a reference to an object (hence the term reference variable).



- Consider a method declared as

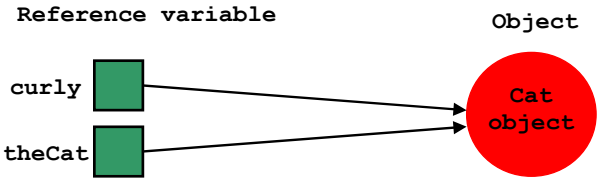
```
public void veterinarian(Cat theCat) {...}
```

- If we call this method passing as a reference to a Cat object, what happens exactly?

```
Cat curly = new Cat();
veterinarian(curly);
```


Parameter Passing with Objects

- The value of the variable `curly` is passed *by value*, and the variable `theCat` within `veterinarian()` receives a copy of this value.
- Variables `curly` and `theCat` now have the same value.
- However, what does it mean to say that two reference variables have the same value ?
- It means that both variables refer to the same object:



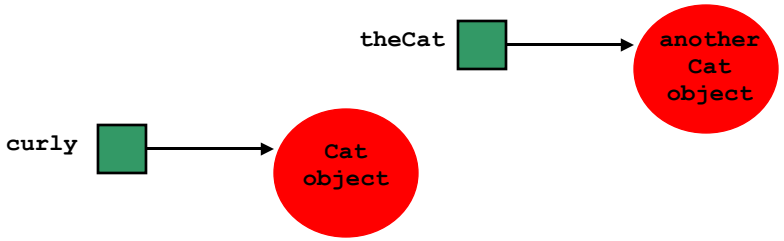
- Within `veterinarian()` you can now update the `Cat` object via variable `theCat`.

Parameter Passing with Objects

- An object can have multiple references to it.
- In this example, we still have just the one object, but it is being referenced by two different variables.
- But if you change the value of the variable `theCat` within `veterinarian()` so that it refers to a different object:

```
theCat = new Cat();
```

- Then the value of variable `curly` within the calling method remains unchanged, and variable `curly` will still refer to the same `Cat` object that it always did:



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- One object (and its data) can be accessed using different variables
- Aliases can be useful, but should be managed carefully
- Changing the object's state (its variables) through one reference changes it for all of its aliases

Passing Objects to Methods Example

- In this example notice the following:
  - What you do to a parameter inside a method may or may not have a permanent effect (outside the method)
  - Note the difference between changing the reference and changing the object that the reference points to

ParameterPassing.java

```
public class ParameterPassing
{
   public static void main (String[] args)
   {
      ParameterTester tester = new ParameterTester();

      int a1 = 111;
      Num a2 = new Num (222);
      Num a3 = new Num (333);

      System.out.println ("Before calling changeValues:");
      System.out.println ("a1\ta2\ta3");
      System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

      tester.changeValues (a1, a2, a3);

      System.out.println ("After calling changeValues:");
      System.out.println ("a1\ta2\ta3");
      System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
   }
}
```

ParameterTester.java

```
public class ParameterTester
{
   public void changeValues (int f1, Num f2, Num f3)
   {
      System.out.println ("Before changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num (777);

      System.out.println ("After changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```


Num.java

```
public class Num
{
   private int value;

   public Num (int update)
   {
      value = update;
   }

   public void setValue (int update)
   {
      value = update;
   }

   public String toString ()
   {
      return value + "";
   }
}
```


Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- It is useless, and therefore called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In some other languages, the programmer has the responsibility for performing garbage collection

Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Member variables (instance variables and static variables) can be used by all methods of the class
  - restriction: static methods can only use static variables
- Data declared within a method can only be used in that method
  - Data declared within a method is called *local data*

Scope

- A variable's scope is the region of a program within which the variable can be referred to by its simple name.
- Secondarily, scope also determines when the system creates and destroys memory for the variable.

Scope

- The location of the variable declaration within your program establishes its scope and places it into one of these 3 categories:
  - member variables
  - method parameter
  - local variable

```
class MyClass
{
    ...
    member variable declarations
    ...
    public void aMethod(method parameters)
      {
        ...
        local variable declarations
        ...
    }
}
```


Scope

- Member Variables
  - A member variable is a member of a class.
  - It is declared within a class but outside of any method.
  - A member variable's scope is the entire declaration of the class.
- Local Variables (local data)
  - You declare local variables within a block of code.
  - In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared. Normally defined by the close curly bracket (}).
- Parameter Variables (local data)
  - Parameters are formal arguments to methods and are used to pass values into methods.
  - The scope of a parameter is the entire method for which it is a parameter. It is treated just like a local variable.

Scope Example 1

- Consider the following example:

```
for (int i = 0; i < 100; i++)
{
   ...
}
System.out.println("The value of i = "+i);
```

- The final line won't compile because the local variable `i` is out of scope.
- Either the variable declaration needs to be moved outside of the `if` statement block, or the `println` method call needs to be moved into the `for` loop.

Scope Example 2

```
class ex2 {
  int x;
  String y;

  int sum(int x, int y) {
    int z = x + y;
    return z;
  }

  String toString() {
    return y + x;
  }

  String message(int y){
    int z = x + y;
    return "Value = " + z;
  }
}
```

Who is referencing what?

`this` reference

- Allows an object to refer to itself
- Most common use in Constructor method

```
public Client (double checkingBalance, double savingsBalance){
   balChecking = checkingBalance;
   balSaving = savingsBalance;
}
```

versus

```
public Client (double balChecking, double balSavings){
   this.balChecking = balChecking;
   this.balSavings = balSavings;
}
```

- Allows parameters and/or local variables to have same name as member variables.
- Member variables can always be referred to through this reference.

Encapsulation

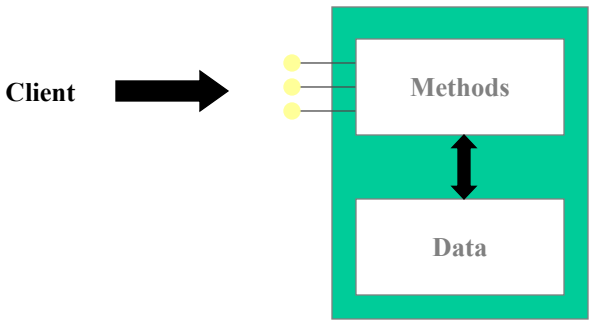
- You can take one of two views of an object:
  - internal - the structure of its data, the algorithms used by its methods
  - external - the interaction of the object with other objects in the program (how to call it)
- From the external view:
  - an object is an *encapsulated* entity, providing a set of specific services
  - These services define the *interface* to the object
  - An object is an *abstraction*, hiding details from the rest of the system

Encapsulation

- An object should be *self-governing*
- Any changes to the object's state (its variables) should be accomplished ONLY by that object's methods
- We should make it difficult, if not impossible, for one object to "reach in" and alter another object's state
- The user of an object can request its services, but it should not have to be aware of how those services are accomplished

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which only invokes the interface methods



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
  - `public` and `private`

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature of a method is built from the method name, number, type, and order of the parameters
- The compiler must be able to determine which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of this signature

Overloading Methods

Version 1

```
float tryMe (int x)
{
   return x+0.375f;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32f)
```

Overloaded Methods

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
      etc.
```

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```


Part 5

Thinking Like A Programmer

(Stepwise refinement & Method decomposition)

Stepwise Refinement

The user wants to be able to input a fixed number of integers into a program and then have it display a menu where the user can select between summing or averaging the values. When the program ends it will write the numbers to the screen together with the sum and average. The numbers are listed vertically down with proper titles for the numbers, sum and average.

How to solve this problem?

Step 1: Identify the Parts

The user wants to be able to input a fixed number of integers into a program and then have it display a menu where the user can select between summing or averaging the values. When the program ends it will write the numbers to the screen together with the sum and average. The numbers are listed vertically down with proper titles for the numbers, sum and average.

The user wants to be able to input a fixed number of integers into a program and then have it display a menu where the user can select between summing or averaging the values. When the program ends it will write the numbers to the screen together with the sum and average. The numbers are listed vertically down with proper titles for the numbers, sum and average.

Step 2: Order the Parts

The user wants to be able to input a fixed number of integers into a program and then have it display a menu where the user can select between summing or averaging the values. When the program ends it will write the numbers to the screen together with the sum and average. The numbers are listed vertically down with proper titles for the numbers, sum and average.

First
Second
Third (choice)
Fourth
Fifth
Sixth

Step 3: Write Code for A Part

From easy to harder…
Draw flowchart if needed…
Write sub-steps in English if needed…

Use:
- Encapsulation, and
- Method decomposition

Notes

- Encapsulation
  - We have already covered this…
    - Objectify things by creating a class that encompasses the desired characteristics
    - Identify state (variables) and activities (methods)
- Method decomposition and parameters
  - As you encapsulate, decisions need to be made concerning the format of your methods and their parameters
  - Which methods are service methods
  - Which methods are interface methods to clients
  - Which should be private and which public

Step 4: Now Assemble into a Program