Introduction To Object-Oriented Programming

Encapsulation

Defining classes and instantiating objects

Attributes and methods

References and parameter passing

Information hiding

Constructors

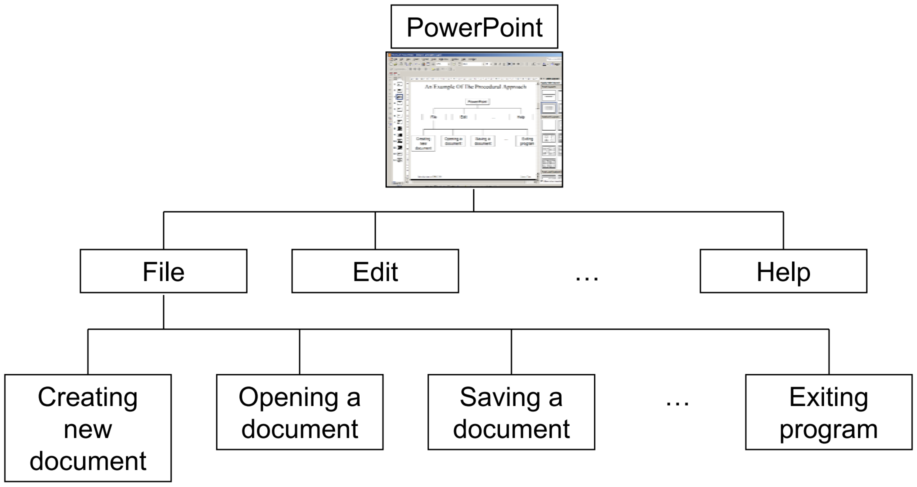
Multiplicity and relationships

Reminder: What You Know

- There are different paradigms (approaches) to implementing computer programs.
- There are several different paradigms but the two you have been introduced to thus far:
  - Procedural
  - Object-Oriented.

An Example Of The Procedural Approach

•Break down the program by what it does (described with *actions/verbs*)



An Example Of The Object-Oriented Approach

•Break down the program into ‘physical’ components (*nouns*)

Dungeon Master

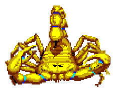
Monsters	
•Scorpion	•Mummy
•Ghost	•Screamer
•Knight	•Dragon

Weapons	
•Broadsword	•Rapier
•Longbow	•Mace

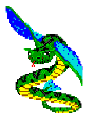
Example Objects: Monsters From Dungeon Master

•Dragon

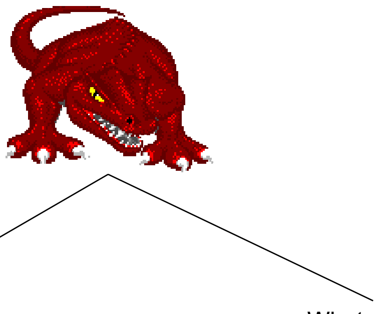
•Scorpion



•Couatl



Ways Of Describing A Monster



What information can be used to describe the dragon? (Attributes)

What can the dragon do? (Behaviors)

Monsters: Attributes

•Represents information about the monster:
- Name
- Damage it inflicts
- Damage it can sustain
- Speed

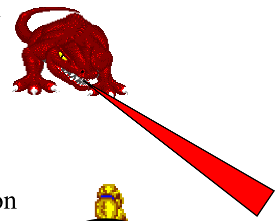
• :

James Tam

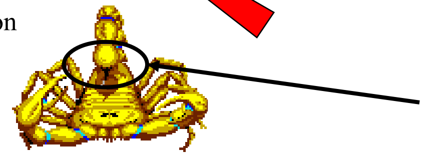
Monsters: Behaviours

• Represents what each monster can do (verb part):

• Dragon



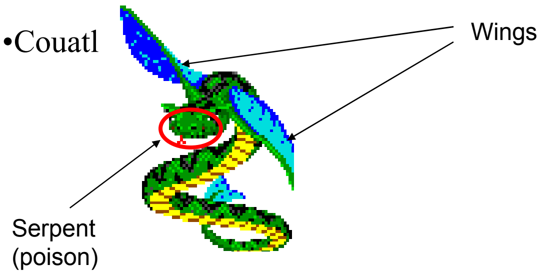
• Scorpion



James Tam

Monsters: Operations

- Couatl

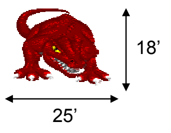


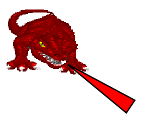
C Structs Vs. Java Objects

Composite type (Structs)

Information (attributes)

- Information about the variable.



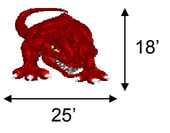


C Structs Vs. Java Objects

Composite type (Objects)

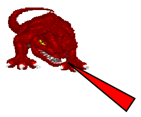
Information (attributes)

- Information about the variable.



Operations (methods[1])

- What the variable “can do”



1 A method is another name for a function in Java

One Benefit Of Bundling Behaviors With Objects

- It can be more logical to bundle into the definition of composite type what each instance can do rather than implementing that function/method elsewhere.

Non-Object-Oriented Approach

```
typedef struct
{
} Dragon;
              :             :
void fly (Dragon a)
{
              :
}
```

Object-Oriented Approach

```
public class Dragon
{
    private int height;
    private int weight;
    public void fly ()
    {
              :
    }
}
```

Working With Objects In Java

I. Define the class

II. Create an instance of the class (instantiate an object)

III. Using the different parts of an object (data and methods)

I) Defining A Java Class

Format:

```
public class <name of class>
{
    instance fields/attributes
    instance methods
}
```

Example:

```
public class Person
{
     // Define instance fields
     // Define instance methods
}
```

Defining A Java Class (2)

Format of instance fields:

- *<access modifier>*[1] *<type of the field> <name of the field>*;

•**Example of defining instance fields**:

```
public class Person
 {
    private int age;
 }
```

1) Can be public or private but typically instance fields are private

2) Valid return types include the simple types (e.g., int, char etc.), predefined classes (e.g., String) or new classes that you have defined in your program. A method that returns nothing has a return type of "void".

James Tam

Defining A Java Class (3)

Format of instance methods:

```
  <access modifier>1 <return type2> <method name> (<p1 type> <p1
    name>…)
    {
          <Body of the method>
    }
```

Example of an instance method:

```
public class Person
{
   public void fun (int num)
   {
       System.out.println (num);
   }
}
```

1) Can be public or private but typically instance methods are public

2) Valid return types include the simple types (e.g., int, char etc.), predefined classes (e.g., String) or new classes that you have defined in your program. A method that returns nothing has return type of "void".

James Tam

Defining A Java Class (4)

Example (complete class definition):

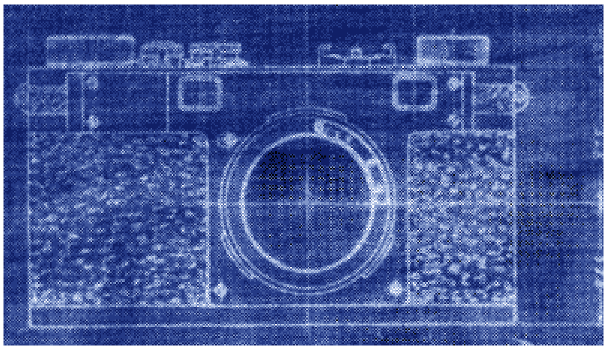
```
public class Person
{
    private int age;
    public void setAge (int anAge)
    {
        age = anAge;
    }
    public int getAge ()
    {
        return age;
    }
}
```

A Class Is Like A Blueprint

•It indicates the format for what an example of the class should look like (methods and attributes).

•Similar to a definition of a struct (except methods can be specified).

•No memory is allocated.



II) Creating/Instantiating Instances Of A Class

Format:

<class name> <instance name> = new *<class name>* ()*;*

Example:

```
Person jim = new Person();
```

- Note: 'jim' is not an object of type 'Person' but a reference to an object of type 'Person'.

An Instance Is An Actual Example Of A Class

- Instantiation is when an actual example/instance of a class is created.

Declaring A Reference Vs. Instantiating An Instance

- Declaring a reference to a 'Person'

```
Person jim;
```

- Instantiating/creating an instance of a 'Person'

```
jim = new Person ();
```

III) Using The Parts Of A Class

Format:

<instance name>.<attribute name>;

<instance name>.<method name>(*<p1 name>*, *<p2 name>…*);

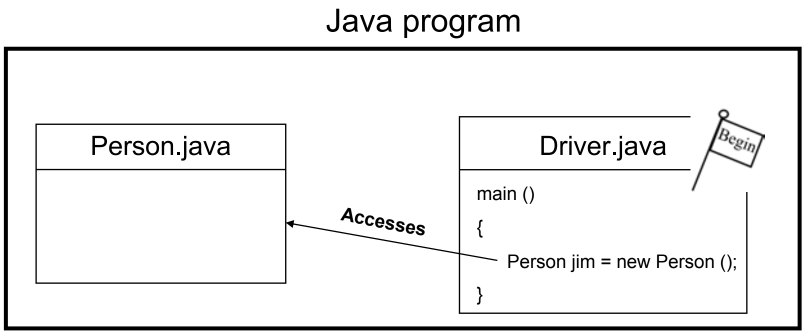
Example:

```
int anAge = 27;
Person jim = new Person ();
jim.setAge(anAge);
System.out.println(jim.getAge());
```

Note: In order to use the dot-operator "." the instance field or method cannot have a private level of access

Laying Out Your Program

- The program must contain a 'Driver' class.
- The driver class is the place where the program starts running (**it contains the main method**).
- Instances of other classes can be created and used here.
- For now you should have all the classes for a particular program reside in the same directory or folder.



```
main ()
{
    Person jim = new Person ();
}
```

Laying Out Your Program

- The code for each class should reside in it's own separate file.

Person.java

```
class Person
{
    :    :
}
```

Driver.java

```
class Driver
{
    :    :
}
```

Putting It Altogether: First Object-Oriented Example

•Example (The complete example can be found in the directory /home/courses/219/examples/introductionOO/firstExample

```
public class Driver
{
    public static void main (String [] args)
    {
        int anAge = 27;
        Person jim = new Person ();
       jim.setAge(anAge);
        System.out.println("Jim's current age is..." + jim.getAge());
    }
}
```

Putting It Altogether: First Object-Oriented Example (2)

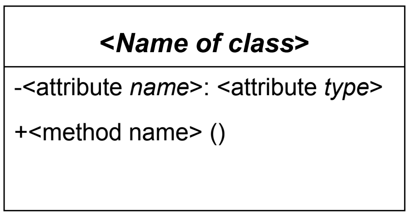
```
public class Person
{
    private int age;
    public void setAge (int anAge)
    {
        age = anAge;
    }
    public int getAge ()
    {
        return age;
    }
}
```

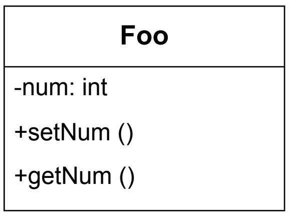
Points To Keep In Mind About The Driver Class

- Contains the only main method of the whole program (where execution begins)
- Do not instantiate instances of the Driver[1]
- For now avoid:
  - Defining instance fields / attributes for the Driver[1]
  - Defining methods for the Driver (other than the main method)[1]

1 Details will be provided later in this course

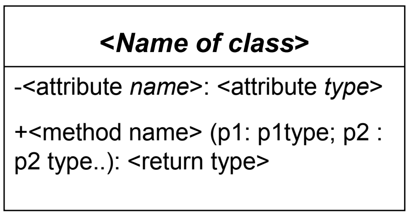
UML[1] Representation Of A Class

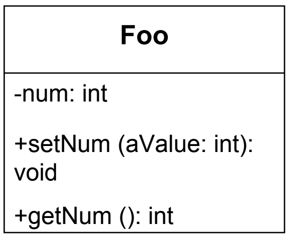




1 UML = Unified Modeling Language

Class Diagrams With Increased Details


<Name of class>
-<attribute *name*>: <attribute *type*>
+<method name> (p1: p1type; p2 : p2 type..): <return type>



2 UML = Unified Modeling Language

Attributes Vs. Local Variables

- Class attributes (variables or constants)
  - Declared inside the body of a class definition but outside the body of any class methods.

```
class Foo
{
    private int num;
}
```

  - Typically there is a separate attribute for each instance of a class and it lasts for the life of the object.

- Local variables and constants
  - Declared within the body of a class' method.
  - Last for the life of the method

```
class Foo
{
    public void aMethod () { char ch; }
}
```

Examples Of An Attribute

```
public class Person
{
    private int age;
    public void setAge (int newAge)
    {
        int aLocal;
        age = newAge;
    }
        :
}
            :
 main (String [] args)
 {
       Person jim = new Person ();
       Person joe = new Person ();
 }
```

"age": Declared within the definition of a class

Examples Of An Attribute

```
public class Person
{
    private int age;
    public void setAge (int anAge)
    {
        int aLocal;
        age = anAge;
    }
        :
}
            :
 main (String [] args)
 {
       Person jim = new Person ();
       Person joe = new Person ();
 }
```

But declared outside of the body of a method

Example Of A Local Variable

```
public class Person
{
   private int age;
   public void setAge (int anAge)
   {
       int aLocal;
       age = anAge;
    }
            :
}
            :
 main (String [] args)
 {
      Person jim = new Person ();
      Person joe = new Person ();
      jim.setAge (5);
      joe.setAge (10);
 }
```

"aLocal": Declared inside the body of a method

Scope Of Local Variables

- Enter into scope
  - Just after declaration
- Exit out of scope
  - When the corresponding enclosing brace is encountered

```
public class Bar
{
                   public void aMethod ()
                   {
                       int num1 = 2;
                       if (num1 % 2 == 0)
                       {
                            int num2;
                            num2 = 2;
                        }
                   }
```

Scope of num1

Scope Of Local Variables

- Enter into scope
  - Just after declaration
- Exit out of scope
  - When the proper enclosing brace is encountered

```
public class Bar
{
                 public void aMethod ()
                 {
                     int num1 = 2;
                     if (num1 % 2 == 0)
                     {
                          int num2;
                          num2 = 2;
                     }
                 }
```

Scope of num2

James Tam

Scope Of Attributes

```
public class Bar
{
           private int num1;
                      :          :
           public void methodOne ()
           {
               num1 = 1;
               num2 = 2;
           }
           public void methodTwo ()
           {
               num1 = 10;
               num2 = 20;
               methodOne ();
           }
                      :          :
           private int num2;
}
```

Scope of num1 & num2

James Tam

Scope Of Attributes

```
public class Bar
{
            private int num1;
                        :           :
            public void methodOne ()
            {
                num1 = 1;
                num2 = 2;
            }
            public void methodTwo ()
            {
                num1 = 10;
                num2 = 20;
                methodOne ();
            }
                        :           :
            private int num2;
}
```

Scope of methodOne and methodTwo

Referring To Attributes And Methods Outside Of A Class: An Example

```
public class Bar
{
    public void aMethod ()
    {
        System.out.println("Calling aMethod of class Bar");
    }
}
```

Scope of aMethod

Referring To Attributes And Methods Outside Of A Class: An Example

```
public class Bar
{
    public void aMethod ()
    {
        System.out.println("Calling aMethod of class Bar");
    }
}

public class Driver
{
    public static void main (String [] args)
    {
        Bar b1 = new Bar ();
        Bar b2 = new Bar ();
        b1.aMethod();
    }
}
```

Outside the scope (dot operator is needed)

Referring To Attributes And Methods Inside Of A Class: An Example

```
public class Foo
{
    private int num;
    public Foo () { num = 0; }
    public void methodOne () { methodTwo(); }
    public void methodTwo () { .. }
        :        :        :
}
  :        :
  main ()
  {
     Foo f1 = new Foo ();
     Foo f2 = new Foo ();
     f1.methodOne();
  }
```

Call is inside the scope (no instance name or 'dot' needed

Call is outside the scope (instance name and 'dot' IS needed

Referring To The Attributes And Methods Of A Class: Recap

1. Outside of the methods of the class you must use the dot-operator as well as indicating what instance that you are referring to.

```
e.g., f1.method();
```

2. Inside the methods of the class there is no need to use the dot-operator nor is there a need for an instance name.

```
e.g.,
public class Foo
{
   public void m1 () { m2(); }
   public void m2 () { .. }
}
```

Shadowing

One form of shadowing occurs when a variable local to the method of a class has the same name as an attribute of that class.

- Be careful of accidentally doing this because the wrong identifier could be accessed.

NO!

```
public class Sheep
{
    private String name;
    public Sheep (String aName)
   {
       String name;
       name = aName;
   }
```

Shadowing

Scope Rules:

1. Look for a local identifier (variable or constant)
2. Look for an attribute

```
public class Foo
{
    // Attributes
    public void method ()
   {
        // Local variables

        num = 1;
    }
}
```

Second: Look for an attribute by that name

First: Look for a local identifier by that name

A reference to an identifier

Encapsulation

- The ability bundle information (attributes) and behavior (methods) into a single entity.
- In Java this is done through a class definition.

Information Hiding

•An important part of Object-Oriented programming and takes advantage of encapsulation.

•Protects the inner-workings (data) of a class.

•Only allow access to the core of an object in a controlled fashion (use the *public* parts to access the *private* sections).

Illustrating The Need For Information Hiding: An Example

•Creating a new monster: “The Critter”

•Attribute: Height (must be 60” – 72”)

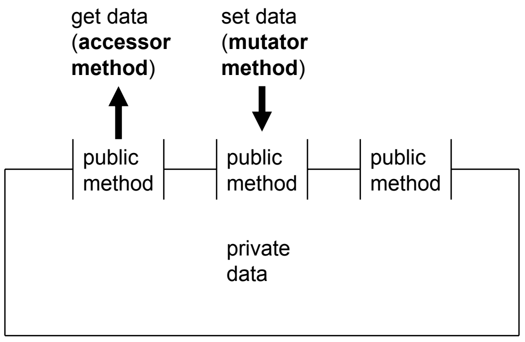
Illustrating The Need For Information Hiding: An Example

•Creating a new monster: “The Critter”

•Attribute: Height (must be 60” – 72”)

Public And Private Parts Of A Class

• The public methods can be used to do things such as access or change the instance fields of the class



Public And Private Parts Of A Class (2)

- Types of methods that utilize the instance fields:

1) **Accessor methods**: a ‘get’ method
   - Used to determine the current value of a field
   - Example:

```
public int getNum ()
{
    return num;
}
```

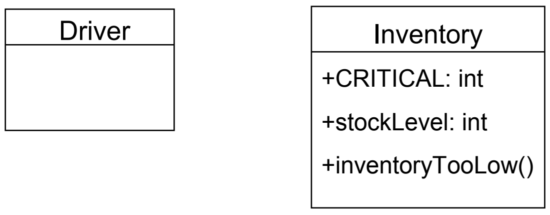
2) **Mutator methods**: a ‘set’ method
   - Used to set a field to a new value
   - Example:

```
public void setNum (int aValue)
{
    num = aValue;
}
```

How Does Hiding Information Protect The Class?

- Protects the inner-workings (data) of a class
  - e.g., range checking for inventory levels (0 – 100)

- The complete example can be found in the directory /home/courses/219/examples/introductionOO/secondExample



The Inventory Class

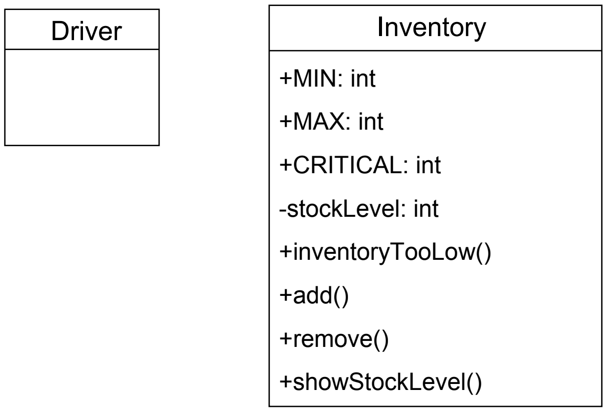
```
public class Inventory
{

   public final int CRITICAL = 10;
   public int stockLevel;
   public boolean inventoryTooLow ()
   {
       if (stockLevel < CRITICAL)
           return true;
      else
           return false;
   }
}
```

The Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
      Inventory chinook = new Inventory ();
      chinook.stockLevel = 10;
      System.out.println ("Stock: " + chinook.stockLevel);
      chinook.stockLevel = chinook.stockLevel + 10;
      System.out.println ("Stock: " + chinook.stockLevel);
      chinook.stockLevel = chinook.stockLevel + 100;
      System.out.println ("Stock: " + chinook.stockLevel);
      chinook.stockLevel = chinook.stockLevel - 1000;
      System.out.println ("Stock: " + chinook.stockLevel);
   }
}
```

Utilizing Information Hiding: An Example

•The complete example can be found in the directory
/home/courses/219/examples/introductionOO/thirdExample



The Inventory Class

```
public class Inventory
{
   public final int CRITICAL = 10;
   public final int MIN = 0;
   public final int MAX = 100;
   private int stockLevel = 0;

   // Method definitions
   public boolean inventoryTooLow ()
   {
      if (stockLevel < CRITICAL)
         return true;
      else
         return false;
   }
```

The Inventory Class (2)

```
public void add (int amount)
{
   int temp;
   temp = stockLevel + amount;
   if (temp > MAX)
   {
      System.out.println();
      System.out.print("Adding " + amount + " item will cause stock ");
      System.out.println("to become greater than " + MAX + " units
        (overstock)");
   }
   else
   {
      stockLevel = temp;
   }
} // End of method add
```

The Inventory Class (3)

```
public void remove (int amount)
{
   int temp;
   temp = stockLevel - amount;
   if (temp < MIN)
   {
        System.out.print("Removing " + amount + " item will cause stock ");
        System.out.println("to become less than " + MIN + " units
          (understock)");
   }
   else
   {
      stockLevel = temp;
   }
}

public String showStockLevel () { return("Inventory: " + stockLevel); }
}
```

The Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Inventory chinook = new Inventory ();
        chinook.add (10);
        System.out.println(chinook.showStockLevel ());
        chinook.add (10);
        System.out.println(chinook.showStockLevel ());
        chinook.add (100);
        System.out.println(chinook.showStockLevel ());
        chinook.remove (21);
        System.out.println(chinook.showStockLevel ());
        // JT: The statement below won't work and for good reason!
        // chinook.stockLevel = -999;
    }
}
```

Information Hiding

VERSION I: BAD!!! ☹

```
public class Inventory
{

    public final int CRITICAL = 10;
    public int stockLevel;
            :           :

}

:           :
chinook.stockLevel = <value!!!>
```

Allowing direct access to the attributes of an object by other programmers is dangerous!!!

VERSION II: BETTER! :D

```
public class Inventory
{
    public final int CRITICAL = 10;
    public final int MIN = 0;
    public final int MAX = 100;
    private int stockLevel = 0;
            :           :
     // mutator and accessors
}
:           :
chinook.add (<value>);
```

Only allow access to privates attributes via public mutators and accessors

Method Overloading

- Same method name but the type, number or order of the parameters is different (method signature).
- Used for methods that implement similar but not identical tasks.
- Method overloading is regarded as good coding style.
- Example:

  System.out.println(int)
  System.out.println(double)
  etc.

  For more details on class System see:
  - http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintStream.html

Method Overloading (2)

- Things to avoid when overloading methods
  1. Distinguishing methods solely by the order of the parameters.
  2. Overloading methods but having an identical implementation.

Method Signatures And Program Design

- Unless there is a compelling reason do not change the signature of your methods!

Before:
```
Class Foo
{
    void fun ()
    {

    }
}
```

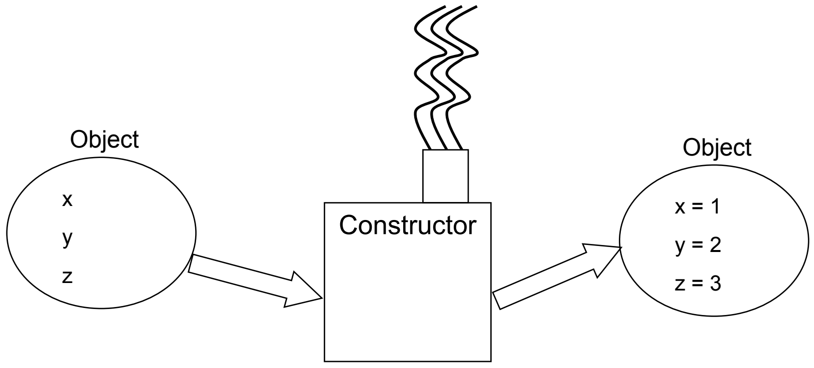
After:
```
Class Foo
{
    void fun (int num)
    {

    }
}
```

```
public static void main ()
{
    Foo f = new Foo ();
    f.fun ()
}
```

This change has broken me! ☹

Creating Objects With The Constructor

- A method that is used to initialize the attributes of an object as the objects are instantiated (created).
- The constructor is automatically invoked whenever an instance of the class is created.



Creating Objects With The Constructor (2)

•If no constructor is specified then the **default constructor** is called

-e.g., Sheep jim = new Sheep();

The call to 'new' calls the default constructor (if no constructor method has been explicitly defined in the class) as an instance of the class is instantiated.

Writing Your Own Constructor

Format (Note: *Constructors have no return type*):

```
public <class name> (<parameters>)
{
    // Statements to initialize the fields of the object
}
```

Example:

```
public Sheep ()
{
    System.out.println("Creating \"No name\" sheep");
    name = "No name";
}
```

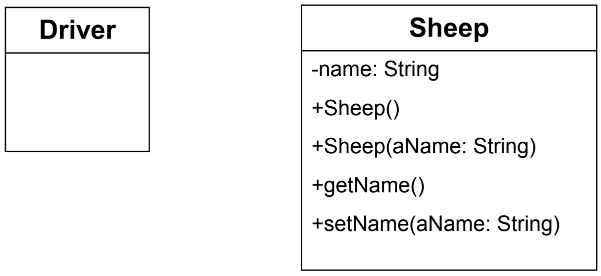
Overloading The Constructor

- Similar to other methods, constructors can also be overloaded
- Each version is distinguished by the number, type and order of the parameters

```
public Sheep ()
public Sheep (String aName)
```

Constructors: An Example

- The complete example can be found in the directory /home/courses/219/examples/introductionOO/fourthExample



The Sheep Class

```
public class Sheep
{
    private String name;

    public Sheep ()
    {
        System.out.println("Creating \"No name\" sheep");
        setName("No name");
    }

    public Sheep (String aName)
    {
        System.out.println("Creating the sheep called " + aName);
        setName(aName);
    }
```

The Sheep Class (2)

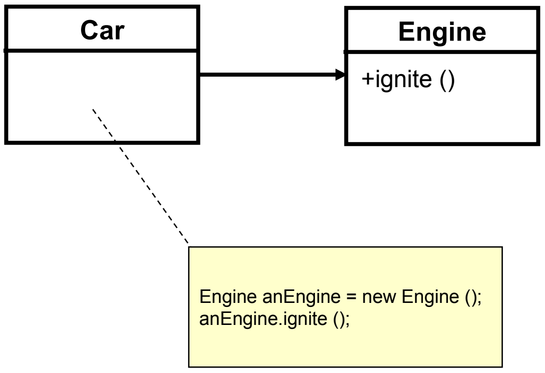
```
    public String getName ()
    {
        return name;
    }

    public void setName (String aName)
    {
        name = aName;
    }
}
```

The Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Sheep nellie;
        Sheep jim;
        System.out.println();
        System.out.println("Creating flock...");
        nellie = new Sheep ("Nellie");
        jim = new Sheep();
        jim.setName("Jim");
        System.out.println("Displaying updated flock");
        System.out.println("   " + nellie.getName());
        System.out.println("   " + jim.getName());
        System.out.println();
    }
}
```

Association Relations Between Classes

•A relation between classes allows messages to be sent (objects of one class can call the methods of another class).



```
Engine anEngine = new Engine ();
anEngine.ignite ();
```

Associations Between Classes

•One type of association relationship is a 'has-a' relation (also known as "aggregation").

- E.g. 1, A car <has-a> engine.
- E.g. 2, A lecture <has-a> student.

•Typically this type of relationship exists between classes when a class is an attribute of another class.

```
public class Car
{
        private Engine anEngine;
        private Lights carLights;
        public start ()
        {
           anEngine.ignite ();
           carLight.turnOn ();
         }
}
```

```
public class Engine
{
        public boolean ignite () { .. }
}

public class Lights
{
        private boolean isOn;
        public void turnOn () { isOn =
        true;}
}
```

Directed Associations

•Unidirectional

- The association only goes in one direction
- You can only navigate from one class to the other (but not the other way around).
- e.g., You can go from an instance of Car to Lights but not from Lights to Car, or you can go from an instance of Car to Engine but not from Engine to Car (previous slide).

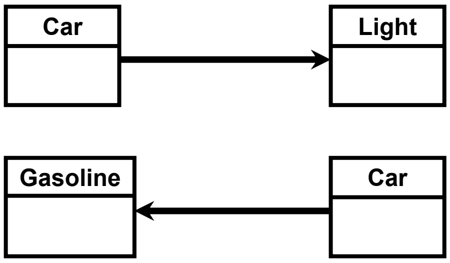
Directed Associations (2)

- Bidirectional
  - The association goes in both directions
  - You can navigate from either class to the other
  - e.g.,

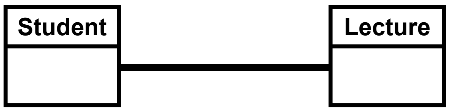
```
public class Student
{
    private Lecture [] lectureList = new Lecture [5];
                              :
}

public class Lecture
{
       private Student [] classList = new Student [250];
                              :
 }
```

UML Representation Of Associations

Unidirectional associations



Bidirectional associations

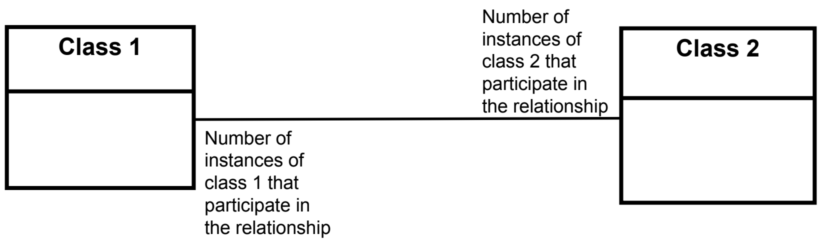


Multiplicity

- It indicates the number of instances that participate in a relationship
- Also known as cardinality

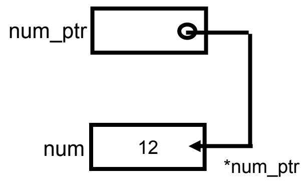
Multiplicity	Description
1	Exactly one instance
n	Exactly "n" instances
n..m	Any number of instances in the inclusive range from "n" to "m"
*	Any number of instances possible

Multiplicity In UML Class Diagrams


Class 1
Number of instances of class 2 that participate in the relationship
Class 2
Number of instances of class 1 that participate in the relationship

Review: C Pointers

```
main ()
{
    int *num_ptr;
    int num = 12;
    num_ptr = &num;
}
```



Java References

- It's a pointer that cannot be explicitly de-referenced by the programmer.
- Dynamic memory: automatically garbage collected when no longer needed.

De-Referencing: Java Example

```
Foo f1 = new Foo ();
Foo f2 = new Foo ();

f1 = f2;
```

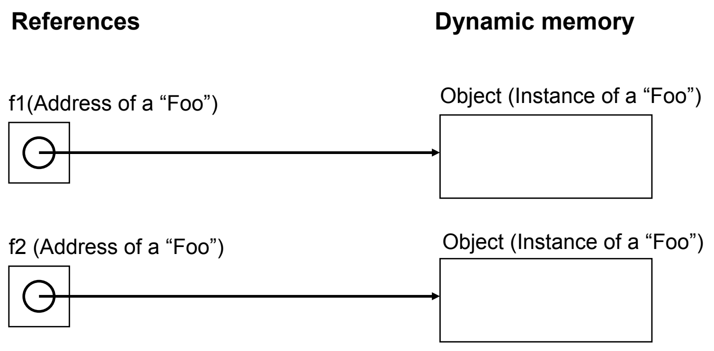
$\leftarrow$ **Exactly what is being copied here?**

Java References

- It's a pointer that cannot be explicitly de-referenced by the programmer.
- Dynamic memory: automatically garbage collected when no longer needed.

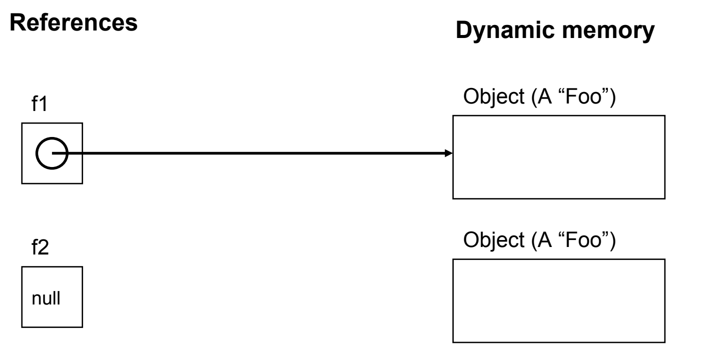
Automatic Garbage Collection Of Java References

•Dynamically allocated memory is automatically freed up when it is no longer referenced



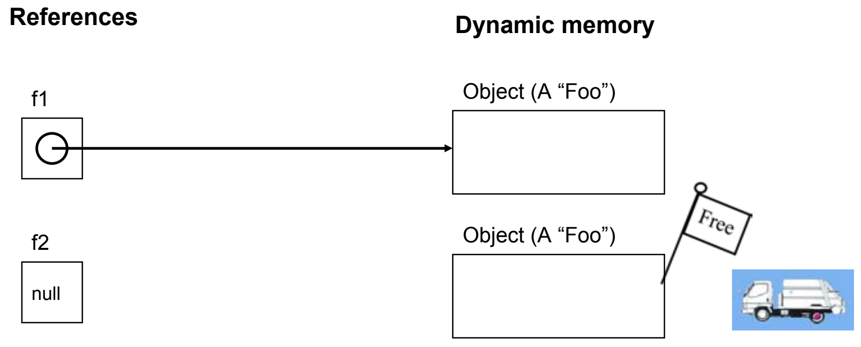
Automatic Garbage Collection Of Java References (2)

•Dynamically allocated memory is automatically freed up when it is no longer referenced e.g., f2 = null;



Automatic Garbage Collection Of Java References (2)

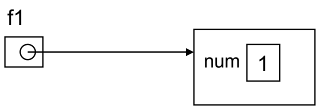
•Dynamically allocated memory is automatically freed up when it is no longer referenced e.g., f2 = null;



The Finalize Method

•Example sequence:

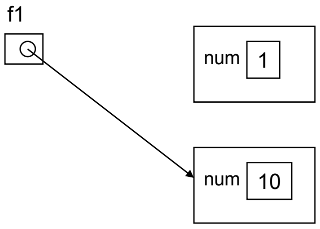
```
public class Foo
{
        int num;
        public Foo () { num = 1; }
        public Foo (int newValue) { num = newValue; }
          :      :          :
}
        :        :
Foo f1 = new Foo ();
```



The Finalize Method

•Example sequence:

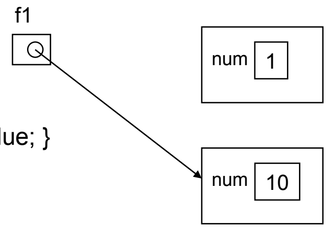
```
public class Foo
{
        int num;
        public Foo () { num = 1; }
        public Foo (int aValue) { num = aValue; }
          :      :        :
}
        :        :
Foo f1 = new Foo ();
f1 = new Foo (10);
```



The Finalize Method

•Example sequence:

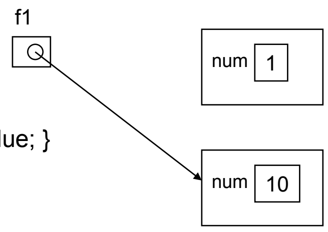
```
public class Foo
{
        int num;
        public Foo () { num = 1; }
        public Foo (int newValue) { num = newValue; }
          :      :        :
}
        :        :
Foo f1 = new Foo ();
f1 = new Foo (10);
```



The Finalize Method

- Example sequence:

```
public class Foo
{
        int num;
        public Foo () { num = 1; }
        public Foo (int newValue) { num = newValue; }
          :       :          :
}
        :         :
Foo f1 = new Foo ();
f1 = new Foo (10);
```

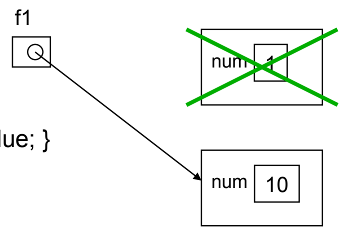


f1.finalize()

The Finalize Method

- Example sequence:

```
public class Foo
{
        int num;
        public Foo () { num = 1; }
        public Foo (int newValue) { num = newValue; }
          :       :          :
}
        :         :
Foo f1 = new Foo ();
f1 = new Foo (10);
```



f1.finalize()

The Finalize Method

- The Java interpreter tracks what memory has been dynamically allocated.
- It also tracks when memory is no longer referenced.
- When the system isn't busy, the Automatic Garbage Collector is invoked.
- If an object has a finalize method then it is invoked:
  - The finalize is a method written by the programmer to free up non-memory resources e.g., closing and deleting temporary files created by the program, closing network connections.
  - This method takes no arguments and returns no values.
  - Dynamic memory is **NOT** freed up by this method.
- After the finalize method finishes execution, the dynamic memory is freed up by the Automatic Garbage Collector.

Common Errors When Using References

- Forgetting to initialize the reference
- Using a null reference

Error: Forgetting To Initialize The Reference

```
Foo f;
f.setNum(10);
```

Compilation error!

```
> javac Driver.java
Driver.java:14: variable f might not have been initialized
        f.setNum(10);
        ^
1 error
```

Error: Using Null References

```
Foo f = null;
f.setNum(10);
```

Run-time error!

```
> java Driver
Exception in thread "main" java.lang.NullPointerException
        at Driver.main(Driver.java:14)
```

Arrays And References

•(Reminder): Arrays involve dynamic memory allocation.

•Arrays are actually references to arrays

Format:

```
<array name> = new <array type> [<no elements>];
```

Example:

```
int [] arr = new int [4];
```

Arrays Of Objects (References)

•An array of objects is actually an array of references to objects

```
class Foo
{
    private int num;
    public void setNum (int aNum) { num = aNum; }
    public int getNum () { return num; }
}

e.g., Foo [] arr = new Foo [4];
```

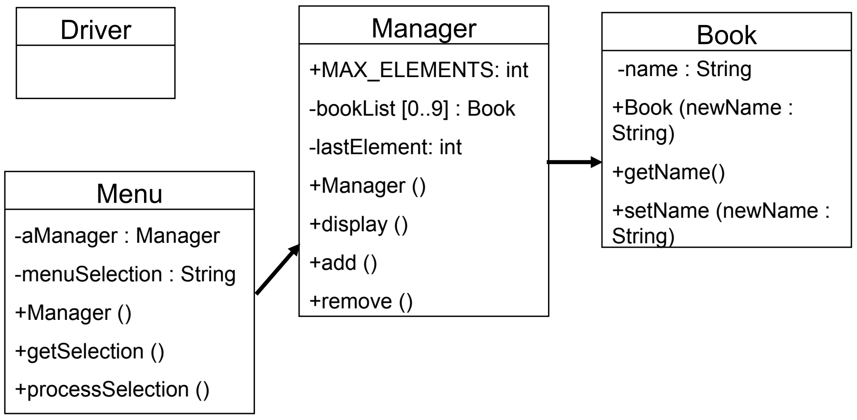
•The elements are initialized to null by default

```
arr[0].setNum(1);
```

NullPointerException

Arrays Of References To Objects: An Example

•The complete example can be found in the directory
/home/courses/219/examples/introductionOO/fifthExample



The Book Class

```
public class Book
{
   private String name;
   public Book (String aName)
   {
      setName(aName);
   }
   public void setName (String aName)
   {
      name = aName;
   }
   public String getName ()
   {
       return name;
   }
}
```

The Manager Class

```
public class Manager
{
    public final int MAX_ELEMENTS = 10;
    private Book [] bookList;
    private int lastElement;

    public Manager ()
    {
        bookList = new Book[MAX_ELEMENTS];
        int i;
        for (i = 0; i < MAX_ELEMENTS; i++)
        {
            bookList[i] = null;
        }
        lastElement = -1;
    }
```

The Manager Class (2)

```
    public void display()
    {
        int i;
        System.out.println("Displaying list");
        if (lastElement == -1)
            System.out.println("\tList is empty");
        for (i = 0; i <= lastElement; i++)
        {
            System.out.println("\tTitle No. " + (i+1) + ": "+ bookList[i].getName());
        }
    }
```

The Manager Class (3)

```
public void add ()
{
    String newName;
    Scanner in;
    if ((lastElement+1) < MAX_ELEMENTS)
    {
        System.out.print("Enter a title for the book: ");
        in = new Scanner (System.in);
        newName = in.nextLine ();
        lastElement++;
        bookList[lastElement] = new Book(newName);
    }
    else
    {
        System.out.print("Cannot add new element: ");
        System.out.println("List already has " + MAX_ELEMENTS + "
        elements.");
    }
}
```

The Manager Class (4)

```
public void remove ()
{
    if (lastElement != -1)
    {
        bookList[lastElement] = null;
        lastElement--;
        System.out.println("Last element removed from list.");
    }
    else
    {
        System.out.println("List is already empty: Nothing to remove");
    }
}
}
```

The Menu Class

```
public class Menu
{
   private Manager aManager;
   private String menuSelection;

   public Menu ()
   {
       aManager = new Manager ();
       menuSelection = null;
   }
```

The Menu Class (2)

```
   public void display ()
   {
        System.out.println("\n\nLIST MANAGEMENT PROGRAM: OPTIONS");
        System.out.println("\t(d)isplay list");
        System.out.println("\t(a)dd new element to end of list");
        System.out.println("\t(r)emove last element from the list");
        System.out.println("\t(q)uit program");
        System.out.print("Selection: ");
   }

   public void getSelection ()
   {
        String newName;
        Scanner in = new Scanner (System.in);
        menuSelection = in.nextLine ();
   }
```

The Menu Class (3)

```
   public void processSelection ()
   {
     do
     {
         display();
         getSelection();
         if (menuSelection.equals("d"))
              aManager.display ();
         else if (menuSelection.equals("a"))
              aManager.add ();
         else if (menuSelection.equals("r"))
              aManager.remove ();
         else if (menuSelection.equals("q"))
              System.out.println ("Quitting program.");
         else
             System.out.println("Please enter one of 'd','a','r' or 'q'");
      } while (!(menuSelection.equals("q")));
   }
}
```

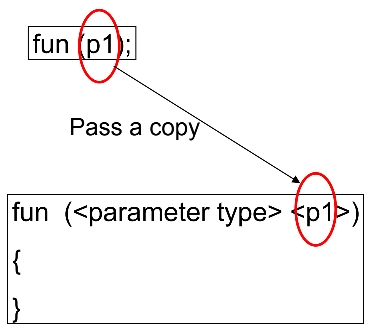
The Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
        Menu aMenu = new Menu ();
        aMenu.processSelection();
   }  // End of main.
}  // End of class Driver.
```

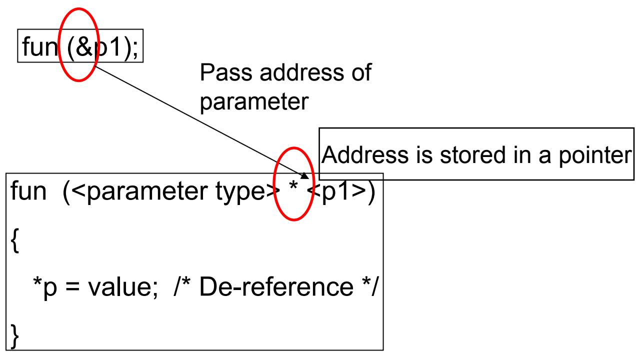
Methods Of Parameter Passing

- Passing parameters as value parameters (pass by value)
- Passing parameters as variable parameters (pass by reference)

Passing Parameters As Value Parameters



Passing Parameters As Reference Parameters


fun (&p1);
Pass address of parameter
Address is stored in a pointer
fun (<parameter type> * <p1>)
{
p = value; / De-reference */
}

Parameter Passing In Java: Simple Types

•All simple types are always passed by value in Java.

Type	Description
byte	8 bit signed integer
short	16 but signed integer
int	32 bit signed integer
long	64 bit signed integer
float	32 bit signed real number
double	64 bit signed real number
char	16 bit Unicode character
boolean	1 bit true or false value

Parameter Passing In Java: Simple Types (2)

Example:

```
 public static void main (String [] args)
{
      int num1;
      int num2;
      Swapper s = new Swapper ();
      num1 = 1;
      num2 = 2;
      System.out.println("num1=" + num1 + "\tnum2=" + num2);
      s.swap(num1, num2);
      System.out.println("num1=" + num1 + "\tnum2=" + num2);
}
```

Passing Simple Types In Java (2)

```
public class Swapper
{
    public void swap (int num1, int num2)
    {
        int temp;
        temp = num1;
        num1 = num2;
        num2 = temp;
        System.out.println("num1=" + num1 + "\tnum2=" + num2);
    }
}
```

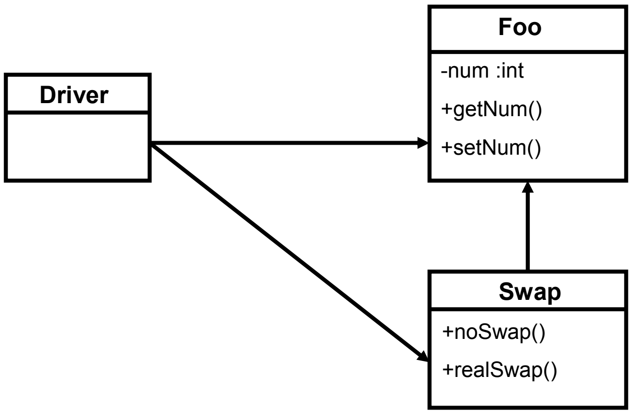
Passing References In Java

- (Reminder: References are required for variables that are arrays or objects)
- Question:
  - If a reference (object or array) is passed as a parameter to a method do changes made in the method continue on after the method is finished?

  Hint: If a reference is passed as a parameter into a method then a copy of the reference is what is being manipulated in the method.

An Example Of Passing References In Java: UML Diagram

•Example (The complete example can be found in the directory /home/courses/219/examples/introductionOO/sixthExample



An Example Of Passing References In Java: The Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Foo f1;
        Foo f2;
        Swap s1;
        f1 = new Foo ();
        f2 = new Foo ();
        s1 = new Swap ();
        f1.setNum(1);
        f2.setNum(2);
```

An Example Of Passing References In Java: The Driver Class (2)

```
      System.out.println("Before swap:\t f1=" + f1.getNum() +"\tf2=" +
                  f2.getNum());
        s1.noSwap (f1, f2);
        System.out.println("After noSwap\t f1=" + f1.getNum() +"\tf2=" +
                  f2.getNum());
        s1.realSwap (f1, f2);
        System.out.println("After realSwap\t f1=" + f1.getNum() +"\tf2=" +
                  f2.getNum());
    }
}
```

An Example Of Passing References In Java: Class Foo

```
public class Foo
{
    private int num;
    public void setNum (int newNum)
    {
        num = newNum;
    }
    public int getNum ()
    {
        return num;
    }
}
```

An Example Of Passing References In Java: Class Swap

```
public class Swap
{
  public void noSwap (Foo f1, Foo f2)
    {
        Foo temp;
        temp = f1;
        f1 = f2;
        f2 = temp;
        System.out.println("In noSwap\t f1=" + f1.getNum () + "\tf2=" +
                   f2.getNum());
    }
```

An Example Of Passing References In Java: Class Swap (2)

```
  public void realSwap (Foo f1, Foo f2)
   {
      Foo temp = new Foo ();
      temp.setNum(f1.getNum());
      f1.setNum(f2.getNum());
      f2.setNum(temp.getNum());
      System.out.println("In realSwap\t f1=" + f1.getNum () + "\tf2=" +
              f2.getNum());
   }
}  // End of class Swap
```

James Tam

References: Things To Keep In Mind

- You can't explicitly de-reference a reference
- But...
  - If you refer to just the name of the reference then you are dealing with the reference (to an object, to an array).
    - E.g., f1 = f2;
    - This copies an address from one reference into another reference, the original objects don't change.
  - If you use the dot operator then you are dealing with the actual object.
    - E.g.,
    - temp = f2;
    - temp.setNum (f1.getNum());
    - temp and f2 refer to the same object and using the dot operator changes the same object.
  - Other times that this may be an issue
    - Assignment
    - Comparisons

James Tam

Shallow Copy Vs. Deep Copies

- Shallow copy
  - Copy the address from one reference into another reference
  - Both references point to the same dynamically allocated memory location
  - e.g.,

```
Foo f1;
Foo f2;
f1 = new Foo ();
f2 = new Foo ();
f1 = f2;
```

Shallow Vs. Deep Copies (2)

- Deep copy
  - Copy the contents of the memory location pointed to by the reference
  - The references still point to separate locations in memory.
  - e.g.,

```
f1 = new Foo ();
f2 = new Foo ();
f1.setNum(1);
f2.setNum(f1.getNum());
System.out.println("f1=" + f1.getNum() + "\tf2=" + f2.getNum());
f1.setNum(10);
f2.setNum(20);
System.out.println("f1=" + f1.getNum() + "\tf2=" + f2.getNum());
```

Comparison Of The References

```
f1 = new Foo ();
f2 = new Foo ();
f1.setNum(1);
f2.setNum(f1.getNum());
if (f1 == f2)
    System.out.println("References point to same location");
else
    System.out.println("References point to different locations");
```

Comparison Of The Data

```
f1 = new Foo2 ();
f2 = new Foo2 ();
f1.setNum(1);
f2.setNum(f1.getNum());
if (f1.getNum() == f2.getNum())
   System.out.println(“Same data");
else
    System.out.println(“Different data");
```

Self Reference: This Reference

•From every (non-static) method of an object there exists a reference to the object (called the "this" reference)

```
e.g.,
Foo f1 = new Foo ();
Foo f2 = new Foo ();
f1.setNum(10);

public class Foo
{
    private int num;
    public void setNum (int num)
   {
       num = num;
   }
         :                  :
 }
```

Self Reference: This Reference

•From every (non-static) method of an object there exists a reference to the object (called the "this" reference)

```
e.g.,
Foo f1 = new Foo ();
Foo f2 = new Foo ();
f1.setNum(10);

public class Foo
{
    private int num;
    public void setNum (int num)
    {
       this.num = num;
    }
    :    :
}
```

Because of the 'this' reference, attributes of an object are always in scope when executing that object's methods.

Implementation Hiding

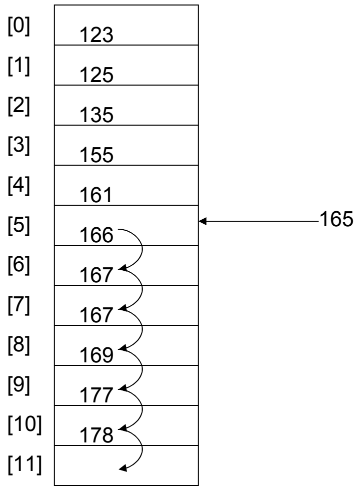
- As long as the signature of a method doesn't change the specific way in which that method implements a task can change as needed.
  - (Worded differently: if you are using a method of another class you won't necessarily care how that method has been implemented as long as it does what you need it to do).
- This hiding of the details of how part of a program has been written (implemented) is referred to as implementation hiding.

Implementation Hiding (2)

- Allows you to use a program module/method without knowing how the code in the module was written (i.e., you don't care about the implementation).
- For example, a list can be implemented as either an array or as a linked list.

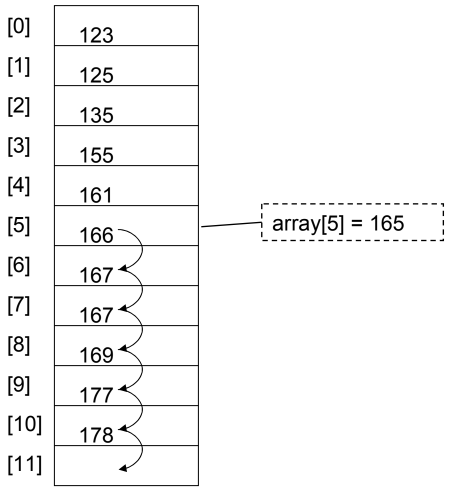
Implementation Hiding (3)

List implemented as an array (add element)



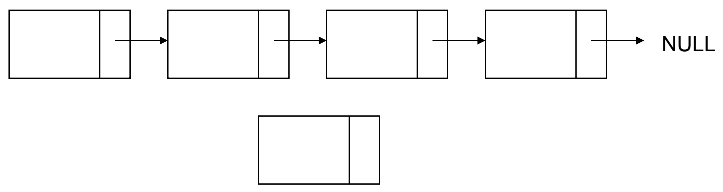
Implementation Hiding (4)

List implemented as an array (add element)



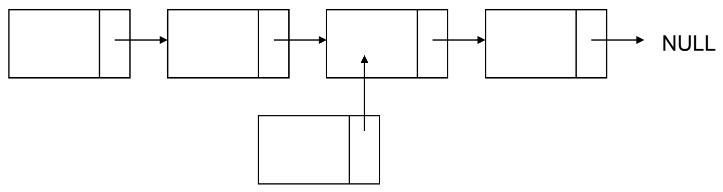
Implementation Hiding (5)

•List implemented as a linked list (add element)



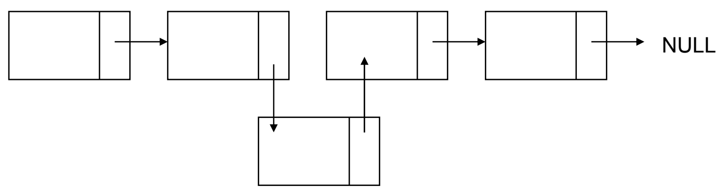
Implementation Hiding (6)

•List implemented as a linked list (add element)



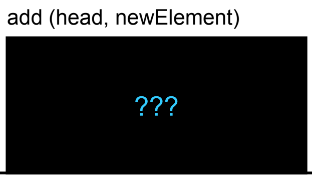
Implementation Hiding (7)

•List implemented as a linked list (add element)



Implementation Hiding (8)

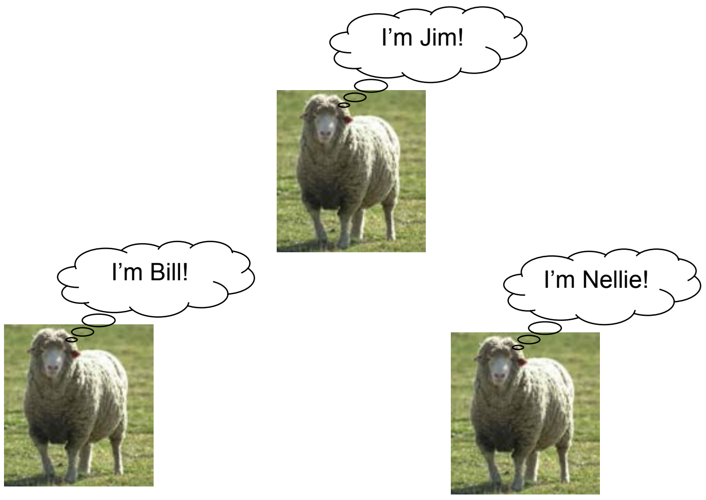
- Changing the implementation of the list should have a minimal (or no) impact on the rest of the program.
- Changing the implementation of a method is separate from changing the signature of a method.
  - This allows the program to modified (and improved) without having side effects on the rest of the program or other programs that use the modified code.
- For example:
  - The "add" method is a black box.
  - We know how to use it without being effected by the details of how it works.



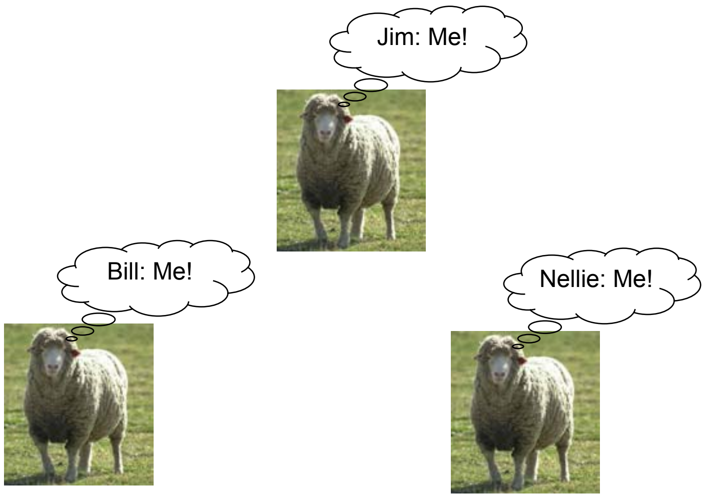
A Previous Example Revisited: Class Sheep

```
public class Sheep
{
   private String name;

   public Sheep ()
   {
      System.out.println("Creating \"No name\" sheep");
      name = "No name";
   }
   public Sheep (String aName)
   {
      System.out.println("Creating the sheep called " + n);
      name = aName;
   }
  public String getName () { return name;}

  public void setName (String newName) { name = newName; }
}
```

We Now Have Several Sheep



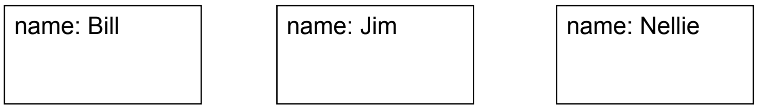
Question: Who Tracks The Size Of The Herd?



Answer: None Of The Above!

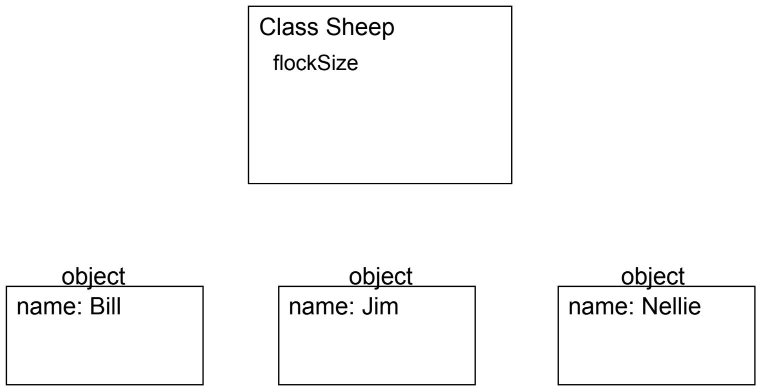
- Information about all instances of a class should not be tracked by an individual object.
- So far we have used instance fields.
- Each *instance* of an object contains *it's own set of instance fields* which can contain information unique to the instance.

```
public class Sheep
{
        private String name;
        :        :        :
}
```



The Need For Static (Class Fields)

- Static fields: One instance of the field exists *for the class* (not for the instances of the class)

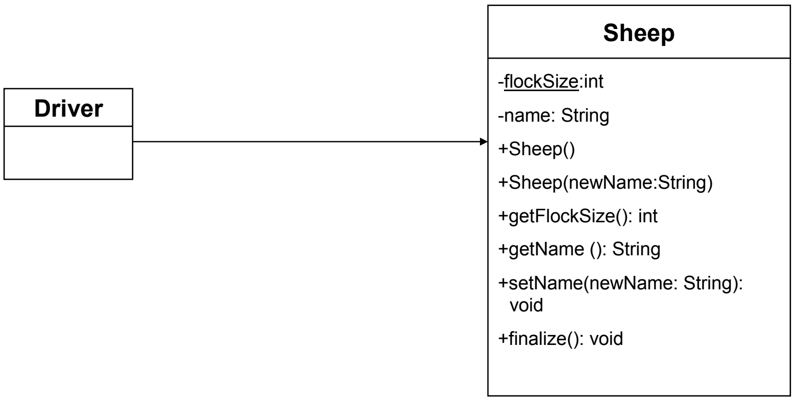


Static (Class) Methods

- Are associated with the class as a whole and not individual instances of the class.
- Typically implemented for classes that are never instantiated e.g., Math.
- May also be used act on the class fields.

Static Data And Methods: UML Diagram

•Example (The complete example can be found in the directory /home/233/examples/advancedOO/seventhExample



Static Data And Methods: The Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
      System.out.println();
      System.out.println("You start out with " + Sheep.getFlockSize() + "
      sheep");
      System.out.println("Creating flock...");
      Sheep nellie = new Sheep ("Nellie");
      Sheep bill = new Sheep("Bill");
      Sheep jim = new Sheep();
```

Static Data And Methods: The Driver Class (2)

```
        System.out.print("You now have " + Sheep.getFlockSize() + " sheep:");
        jim.setName("Jim");
        System.out.print("\t"+ nellie.getName());
        System.out.print(", "+ bill.getName());
        System.out.println(", "+ jim.getName());
        System.out.println();
    }
}  // End of Driver class
```

Static Data And Methods: The Sheep Class

```
public class Sheep
{
  private static int flockSize = 0;
  private String name;

   public Sheep ()
    {
        flockSize++;
        System.out.println("Creating \"No name\" sheep");
        name = "No name";
    }

    public Sheep (String aName)
    {
        flockSize++;
        System.out.println("Creating the sheep called " + newName);
        name = aName;
    }
```

Static Data And Methods: The Sheep Class (2)

```
   public static int getFlockSize () { return flockSize; }

   public String getName () {  return name; }

   public void setName (String newName)  { name = newName; }

   public void finalize ()
   {
      System.out.print("Automatic garbage collector about to be called for ");
      System.out.println(this.name);
      flockSize--;
   }
}// End of definition for class Sheep
```

Rules Of Thumb: Instance Vs. Class Fields

- If a attribute field can differ between instances of a class:
    - The field probably should be an instance field (non-static)
- If the attribute field relates to the class (rather to an instance) or to all instances of the class
    - The field probably should be a static field of the class

Rule Of Thumb: Instance Vs. Class Methods

- If a method should be invoked regardless of the number of instances that exist then it probably should be a static method.
- If it never makes sense to instantiate an instance of a class then the method should probably be a static method.
- Otherwise the method should likely be an instance method.

Static Vs. Final

- **Static**: Means there's one instance of the field for the class (not individual instances of the field for each instance of the class)
- **Final**: Means that the field cannot change (it is a constant)

```
public class Foo
{
    public static final int num1= 1;
    private static int num2;            /*  Rare */
    public final int num3 = 1;          /* Why bother? */
    private int num4;
            :           :
}
```

An Example Class With A Static Implementation

```
public class Math
{
 // Public constants
 public static final double E = 2.71…
 public static final double PI = 3.14…

 // Public methods
 public static int abs (int a);
 public static long abs (long a);
             :              :
}
```

- For more information about this class go to:
- http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html

Should A Class Be Entirely Static?

- Generally it should be avoided if possible because it often bypasses many of the benefits of the Object-Oriented approach.
- Usually purely static classes (cannot be instantiated) have only methods and no data (maybe some constants).
- When in doubt do not make attributes and methods static.

A Common Error With Static Methods

•Recall: The “this” reference is an implicit parameter that is automatically passed into the method calls (you’ve seen so far).

•e.g.,

•Foo f = new Foo ();

•f.setNum(10);

Explicit parameter

Implicit parameter “this”

A Common Error With Static Methods

•Static methods have no “this” reference as an implicit parameter (because they are not associated with any instances).

```
public class Driver
{
    private int num;
    public static void main (String [] args)
    {
        num = 10;
    }
}
```

Compilation error:

```
Driver3.java:6: non-static variable num cannot be referenced from a static context
        num = 10;
        ^
error
```

Common Methods That Are Implemented

- The particular methods implemented for a class will vary depending upon the application.
- However two methods that are commonly implemented for many classes:
  - toString
  - equals

“Method: toString”

- It’s commonly written to allow easy determination of the state of a particular object (contents of important attributes).
- This method returns a string representation of the state of an object.
- It will automatically be called whenever a reference to an object is passed as a parameter is passed to the “print/println” method.
- The full example can be found online under:

/home/courses/219/examples/introductionOO/eighthExample

Class Person: Version 1

```
public class Person
{
    private String name;
    private int age;
    public Person () {name = "No name"; age = -1; }
    public void setName (String aName) { name = aName; }
    public String getName () { return name; }
    public void setAge (int anAge) { age = anAge; }
    public int getAge () { return age; }
}
```

Class Person: Version 2

```
public class Person2
{
    private String name;
    private int age;
    public Person2 () {name = "No name"; age = -1; }
    public void setName (String aName) { name = aName; }
    public String getName () { return name; }
    public void setAge (int anAge) { age = anAge; }
    public int getAge () { return age; }

    public String toString ()
    {
        String temp = "";
        temp = temp + "Name: "+ name + "\n";
        temp = temp + "Age: " + age + "\n";
        return temp;
    }
}
```

The Driver Class

```
class Driver
{
    public static void main (String args [])
    {
        Person p1 = new Person ();
        Person2 p2 = new Person2 ();
        System.out.println(p1);
        System.out.println(p2);
    }
}
```

"Method: equals"

- It's written in order to determine if two objects of the same class are in the same state (attributes have the same data values).
- The full example can be found online under:

/home/courses/219/examples/introductionOO/ninthExample

The Driver Class

```
class Driver
{
   public static void main (String args [])
   {
      Person p1 = new Person ();
      Person p2 = new Person ();
      if (p1.equals(p2) == true)
         System.out.println ("Same");
      else
         System.out.println ("Different");

      p1.setName ("Foo");
      if (p1.equals(p2) == true)
         System.out.println ("Same");
      else
         System.out.println ("Different");
   }
}
```

The Person Class

```
public class Person
{
   private String name;
   private int age;
   public Person () {name = "No name"; age = -1; }
   public void setName (String aName) { name = aName; }
   public String getName () { return name; }
   public void setAge (int anAge) { age = anAge; }
   public int getAge () { return age; }
   public boolean equals (Person aPerson)
   {
      boolean flag;
      if ((name.equals(aPerson.getName())) && (age == aPerson.getAge ()))
          flag = true;
      else
          flag = false;
      return flag;
   }
}
```

After This Section You Should Now Know

- How to define classes, instantiate objects and access different part of an object
- What is the difference between a class, a reference and an object
- How to represent a class using class diagrams (attributes, methods and access permissions) and the relationships between classes
- Scoping rules for attributes, methods and locals
- What is encapsulation and how is it done
- What is information hiding, how is it done and why is it important to write programs that follow this principle
- What are accessor and mutator methods and how they can be used in conjunction with information hiding

After This Section You Should Now Know (2)

- What is method overloading and why is this regarded as good style
- What is method overloading, how is it done, why is it done
- What is a constructor and how is it used
- What is an association, how do directed and non-directed associations differ, how to represent associations and multiplicity in UML
- What is multiplicity and what are kinds of multiplicity relationships exist
- How are the different parameter passing mechanisms (value and reference) implemented in Java
- What is implementation hiding

After This Section You Should Now Know (3)

- What is a static method and attribute, when is appropriate for something to be static and when is it inappropriate (bad style)
- Two useful methods that should be implemented for almost every class: toString and equals