CPSC 233: Advanced Java And Object-Oriented Concepts

More on Java methods

Relations between classes
- Association
- Aggregation

Multiplicity

Issues associated with references

The finalize method

The static keyword

Classes and state

Debugging code

James Tam

More On Java Methods

- Method overloading and the signature of a method
- Message passing
- Implementation hiding

James Tam

More On Java Methods

- **Method overloading and the signature of a method**
- Message passing
- Implementation hiding

Method Overloading

- Same method name but the type, number or order of the parameters is different
- Used for methods that implement similar but not identical tasks.
- This is 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 Signatures And Method Overloading

A Method signature consists of:
- The name of the method
- The number **and** type of parameters (Reminder: Do not distinguish methods solely by the order of the parameters).

Usage of method signatures:
- To distinguish overloaded methods (same name but the type, number or ordering of the parameters is different).
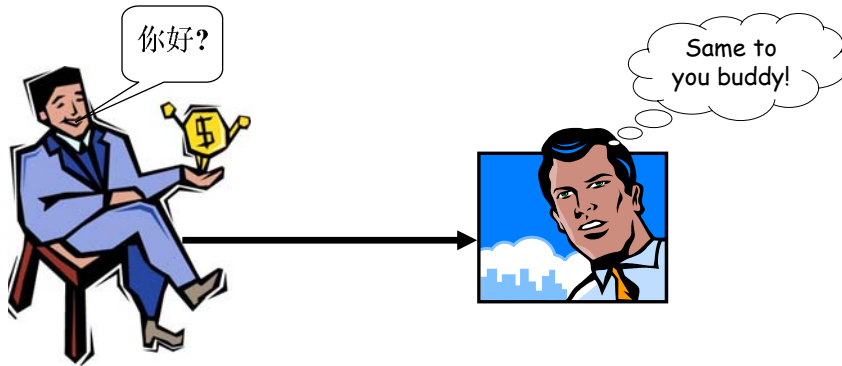
More On Java Methods

- Method overloading and the signature of a method
- **Message passing**
- Implementation hiding

Message Passing: General Definition

A communication between a sender and a receiver

你好?

Same to you buddy!

Message Passing: Object-Oriented Definition

- A request from the source object to the destination object to apply one its operations.
- An object invoking the method of an object (which may be the same object or a different object)

```
e.g.,
public class Bar
{
    public void aMethod ()
    {
        Foo f = new Foo ();
        f.getNum()
    }
}
```

A message is sent from an instance of Bar to an instance of Foo

Message Passing And Program Design

Procedural approach
- •Start with a function or procedure and pass the composite types to this method
- •i.e., procedure (record)
- •e.g.,

```
 CD =  record
        title    :  array [1..80] of char;
        artist   :  array [1..80] of char;
        price    :  real;
        rating   :  integer;
        category : char;
      end;
   Collection = array [1..NO] of CD;
        :                :
Collection tamjCollection;
        :                :
procedure displayCollection (tamjCollection : Collection);
```

Message Passing And Program Design (2)

Object-Oriented approach:
- •Start with an object and then determine which method to invoke
- •i.e., object.method ()
- •e.g.,

```
public class Foo
{
   private int num;
   public void setNum (int newValue)  { num = newValue; }
   public int getNum () { return num; }
}
        :          :
f.getNum();
```

More On Java Methods

- Method overloading and the signature of a method
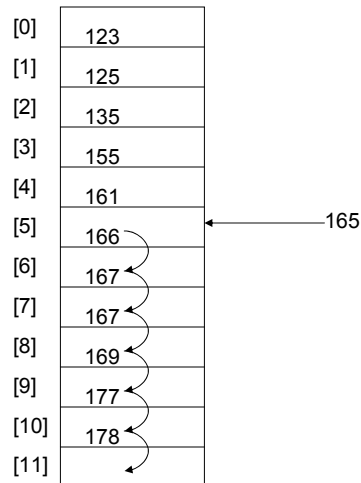- Message passing
- **Implementation hiding**

Implementation Hiding

- 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.
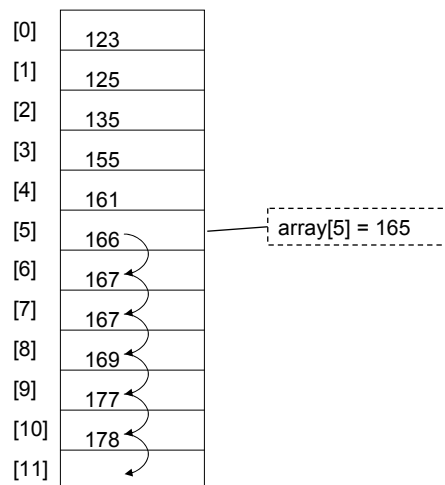
Implementation Hiding (2)

List implemented as an array (add element)

[0]	123
[1]	125
[2]	135
[3]	155
[4]	161
[5]	166 ← 165
[6]	167
[7]	167
[8]	169
[9]	177
[10]	178
[11]	

James Tam

Implementation Hiding (2)

List implemented as an array (add element)

[0]	123
[1]	125
[2]	135
[3]	155
[4]	161
[5]	166 ← array[5] = 165
[6]	167
[7]	167
[8]	169
[9]	177
[10]	178
[11]	

James Tam
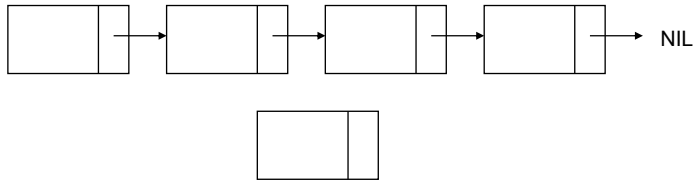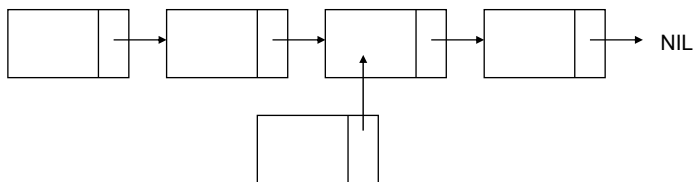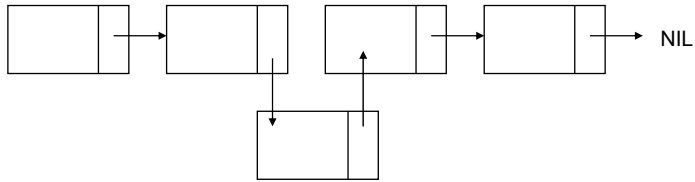
Implementation Hiding (3)

List implemented as a linked list (add element)

James Tam

Implementation Hiding (3)

List implemented as a linked list (add element)

James Tam

Implementation Hiding (3)

List implemented as a linked list (add element)

Implementation Hiding (4)

- 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.
- 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.

add (head, newElement)

???

Relations Between Classes

1. Association (this section of notes)
2. Aggregation (this section of notes)
3. Inheritance (next section of notes)

Associations Between Classes

- Allows for navigation between from one class to another (you can access the public parts of the class).
- Associations occur between classes when:
  - An instance of a class is a attribute of another class
  - An instance of a class a local variable in another class's method
- Also known as a "knows-a" relation
- Association relations allows for messages to be sent

Associations: Lights Is An Attribute Of Car

```
public class Car
{
    private Lights headLights = new Lights ();
              :
    public startCar () { headLights.turnOn(); }
}

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

Associations: Gasoline Is A Local Variable In A Method Of Class Car

```
public class Car
{
    public startCar ()
    {
        Gasoline fuel = new Gasoline ();
        fuel.burnupForIgnition();
    }
}

public class Gasoline
{
    public  void burnUpForIgnition () { .. }
}
```

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 Gasoline but not from Gasoline to Car (previous slides).

Directed Associations (2)

Bidirectional

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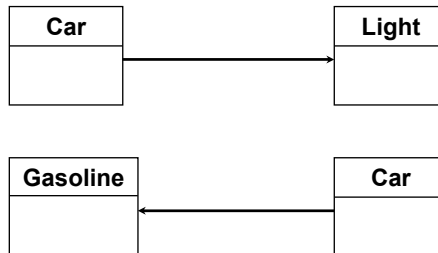- e.g.,

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

public class Lecture
{
    private Student [] studentList = new Student [250];
            :
}
```
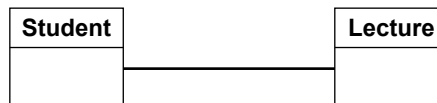
UML Representation Of Associations

Unidirectional associations

```
┌─────────┐              ┌─────────┐
│   Car   │              │  Light  │
├─────────┤─────────────▶├─────────┤
│         │              │         │
└─────────┘              └─────────┘
```

```
┌─────────┐              ┌─────────┐
│Gasoline │              │   Car   │
├─────────┤◀─────────────├─────────┤
│         │              │         │
└─────────┘              └─────────┘
```

Bidirectional associations

```
┌─────────┐              ┌─────────┐
│ Student │              │ Lecture │
├─────────┤──────────────├─────────┤
│         │              │         │
└─────────┘              └─────────┘
```

Aggregation Relations Between Classes

- A stronger form of association between classes
- Can occur when a class consists of another class
  - An instance of a class an attribute of another class

  and
  - The first class is conceptually a part of the second class
- Also known as a "has-a" relation

e.g.,

```
public class Company
{
    private Department divisions [];
}
```

Graphical Representations Of Aggregations

Company				Department	

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

Association Vs. Aggregation

- Aggregation is a more specific form of association (one class consists of the other)
- Navigation is not the same as aggregation! (Again: One class is an attribute field of another class **AND** the first class is a part of the second class)

Association
```
public class BankAccount
{
    private Person accountHolder;
            :
}
```

Aggregation
```
public class Person
{
    private Head myHead;
    private Heart myHeart;
            :
}
```

James Tam

Issues Associated With References

- Parameter passing
- Assignment of references and deep vs. shallow copies
- Comparisons of references

James Tam

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

procedureName (p1);

Pass a copy

procedureName (p1 : parameter type);
begin
end;

Passing Parameters As Variable/Reference Parameters

procedureName (p1);

Pass (a
pointer to)
the variable

procedureName (var p1 : parameter type);

begin

end;

Parameter Passing In Java: Simple Types

All simple types are 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:
```java
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)

```java
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/233/examples/advancedOO/firstExample
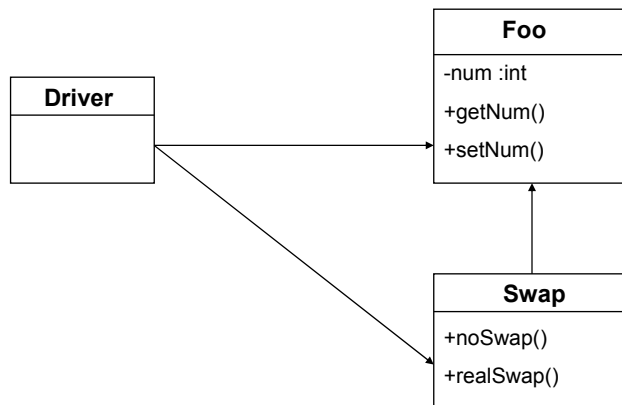
An Example Of Passing References In Java: The Driver Class

```
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

```java
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

```java
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
```

Passing (References) To Arrays As Method Parameters

- Because the copy of an address is stored in the method, passing arrays as a parameter works the same way as passing a reference to a class as a parameter.

```
e.g.,
char [] list = new char[4];
reference.changeOne(list);
        :        :        :
  public void changeOne (char [] list)
  {
     char [] temp = new char[4];
     list = temp;
  }
```

Passing (References) To Arrays As Method Parameters (2)

```
e.g.,
char [] list = new char[4];
reference.changeTwo(list);
        :        :        :

public void changeTwo (char [] list)
  {
     list[0] = '*';
     list[3] = '*';
  }
```

Changing The Value Of Simple Types That Are Passed Into Method

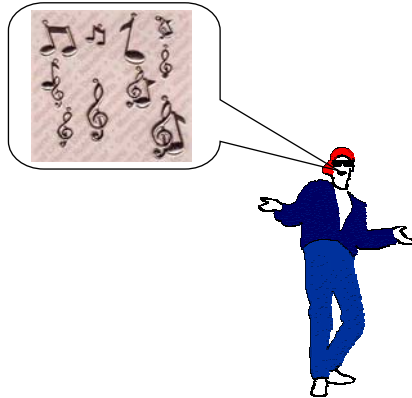- Use a return value to pass back the updated value.

```
int num = 0;
Foo f = new Foo ();
num = f.method (num);
 :   :   :
public class Foo
{
   public int method (int num) { return (num+10); }
}
```

- Mimic passing the simple type as a reference parameter.

Passing By Reference For Simple Types

•It cannot be done directly in Java

•You must use a wrapper!

Wrapper Class

• A class definition built around a simple type

e.g.,

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

Issues Associated With References

- Parameter passing
- **Assignment of references and deep vs. shallow copies**
- Comparisons of references

Shallow Copy Vs. Deep Copies

Shallow copy
- Copy the address in 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());
```

Remember: The Assignment Operator Effect
Reference Not The Data!

```
Foo f1;
Foo f2;
f1 = new Foo ();
f2 = new Foo ();
f1.setNum(1);
f2.setNum(2);
System.out.println("f1=" + f1.getNum() + "\tf2=" + f2.getNum());
f1 = f2;
f1.setNum(10);
f2.setNum(20);
System.out.println("f1=" + f1.getNum() + "\tf2=" + f2.getNum());
```

Issues Associated With References

- Parameter passing
- Assignment of references and deep vs. shallow copies
- **Comparisons of references**

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 most every method of an object there exists a pointer
to the object ("this")
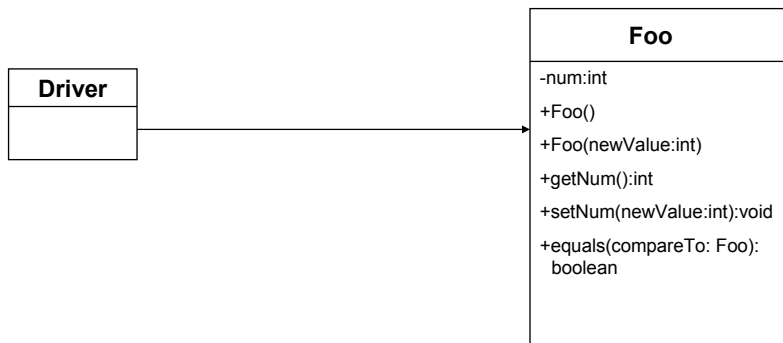
```
e.g.,
Foo f1 = new Foo ();
Foo f2 = new Foo ();
f1.setNum(10);

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

Uses Of "This" When Checking For Equality:
UML Diagram

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

Foo
-num:int
+Foo()
+Foo(newValue:int)
+getNum():int
+setNum(newValue:int):void
+equals(compareTo: Foo): boolean

Driver

Checking For Equality: The Driver Class

```java
public class Driver
{
   public static void main (String [] args)
   {
      Foo f1 = new Foo(1);
      Foo f2 = new Foo(2);
      if (f1.equals(f2))
         System.out.println("Data of f1 and f2 the same.");
      else
         System.out.println("Data of f1 and f2 are not the same.");
   }
}
```

Uses Of "This": Checking For Equality (2)

```java
public class Foo
{
   private int num;

   public Foo ()
   {
      num = 0;
   }

   public Foo (int newValue)
   {
      num = newValue;
   }

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

Uses Of "This": Checking For Equality (3)

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

 public boolean equals (Foo compareTo)
 {
   if (num == compareTo.num)
     return true;
   else
     return false;
 }
}
```

Uses Of "This": Checking For Equality (3)

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

 public boolean equals (Foo compareTo)
 {
   if (this.num == compareTo.num)
     return true;
   else
     return false;
 }
}
```

Explicit Vs. Implicit Parameters

Explicit parameters
- •Are the parameters enclosed within the brackets of a method call.
- •e.g.,

  Foo f = new Foo ();

  int no = 10;
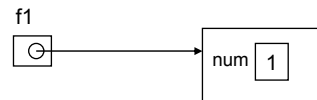
  f.setNum(no);

Implicit parameters
- •Do not need to be explicitly passed into a method in order to be used
- •The "this" reference is the implicit parameter

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:

f1

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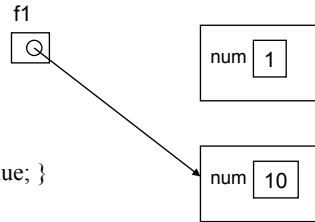
num | 1

num | 10

The Finalize Method

Example sequence:

f1

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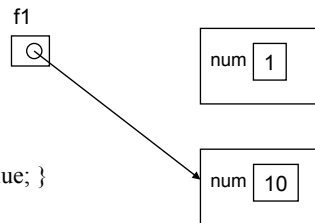
num | 1

num | 10

When???

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);
```
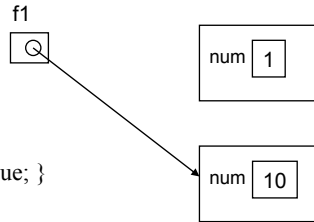
f1

num | 1

num | 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);
```
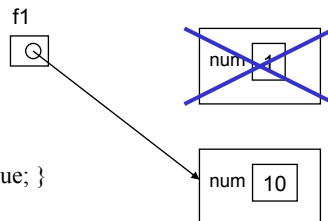
f1

num | 1

num | 10

f1.finalize()

Synopsis Of 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.

A Previous Example Revisited: Class Sheep
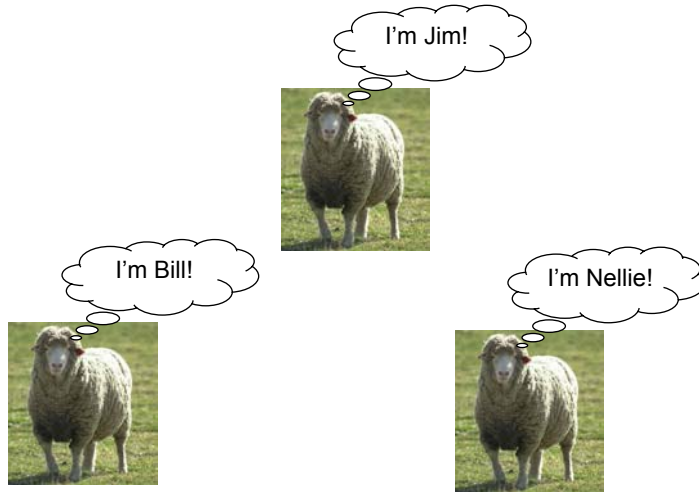
```
public class Sheep
{
   private String name;

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

   public void setName (String newName) { name = newName; }
}
```
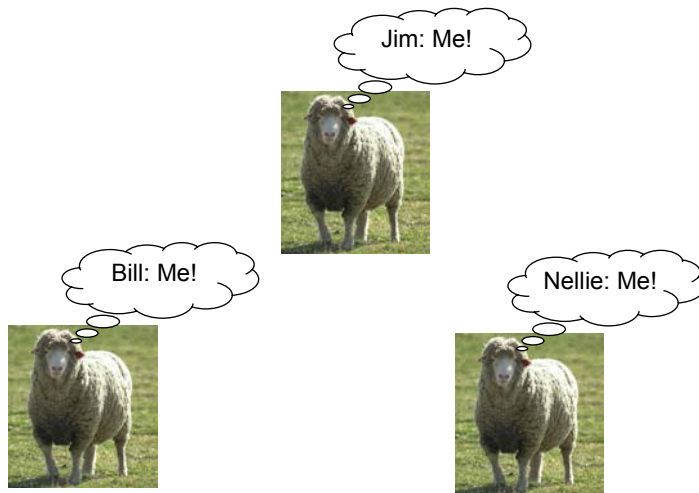
We Now Have Several Sheep

I'm Jim!

I'm Bill!

I'm Nellie!

Question: Who Tracks The Size Of The Herd?

Jim: Me!

Bill: Me!

Nellie: Me!

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;
        :       :       :
}
```

name: Bill	name: Jim	name: Nellie

The Need For Static (Class Fields)

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

```
Class Sheep
  flockSize
```

object   object   object

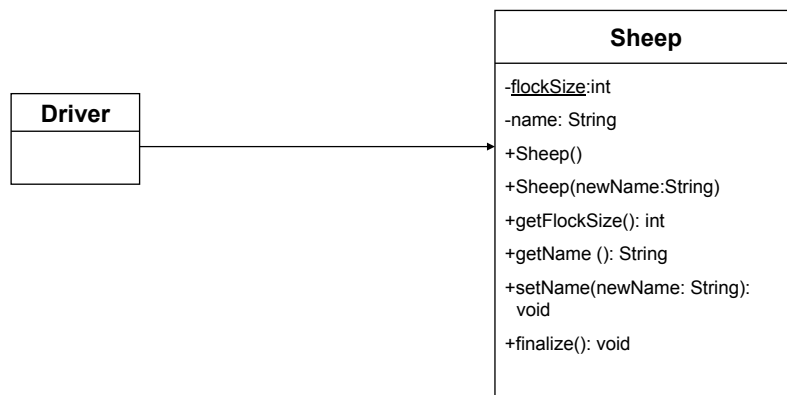| name: Bill	name: Jim	name: Nellie

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/thirdExample

Sheep
-flockSize:int
-name: String
+Sheep()
+Sheep(newName:String)
+getFlockSize(): int
+getName (): String
+setName(newName: String): void
+finalize(): void

Driver

Static Data And Methods: The Driver Class

```java
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)

```java
      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

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

Static Data And Methods: The Sheep Class (2)

```java
  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.
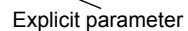
A Common Error With Static Methods

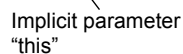Recall: The "this" reference is an implicit parameter that is automatically passed into a method.

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

Classes And State

- The state of an object is determined by the values of it's attributes.
- The state of objects can be modeled by State diagrams.
- Not all attributes are modeled, attributes that typically are modeled in the state of an object include:
  - Attributes that can only take on a limited range of values e.g., boolean
  - Attributes that have restrictions regarding the values that it may take on e.g., programmer defined ranges for a long may only include 1, 2, 3 or 4.
  - Designers are only interested when attributes take on a specific range of values e.g., negative integer values are ignored.

Example Class: Adventurer

```
public class Adventurer
{
    private boolean okay;
    private boolean poisoned;
    private boolean confused;
    private boolean dead;
        :
}
```

Class Adventurer: The Set Of States

Injected with poison

Okay

Receive antidote   **Poisoned**

After (10 minutes)

Hit by confusion spell

Receive cure

Confused

Dead

Resurrected
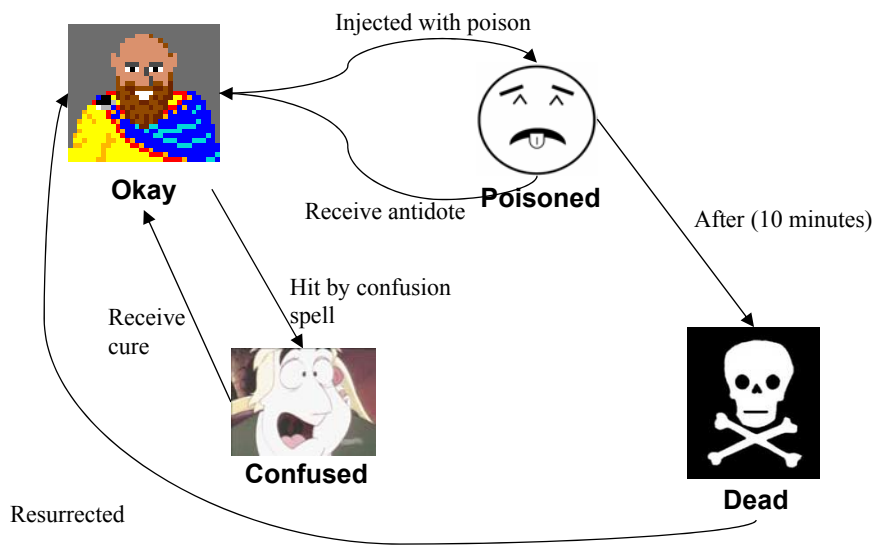
Class Adventurer: State Diagram

Okay

Injected with poison

Poisoned

Receive antidote

After (10 minutes)

Receive cure

Resurrected

Hit by confusion spell

Confused

Dead
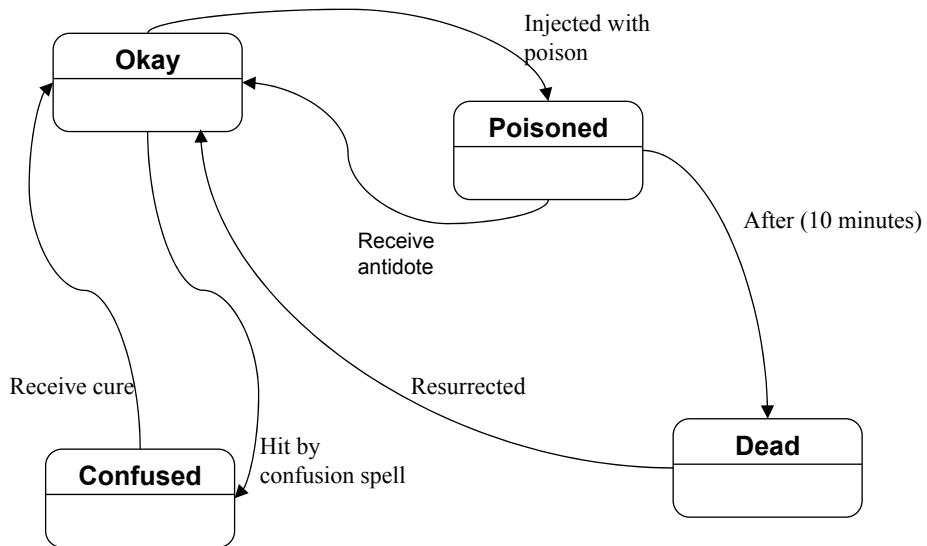
Determining The State Of Objects

- Determining the value of the attributes of a object (state) can be a useful debugging tool.

An Example Of Determining Object's State:
UML Diagram

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

Driver
main ()

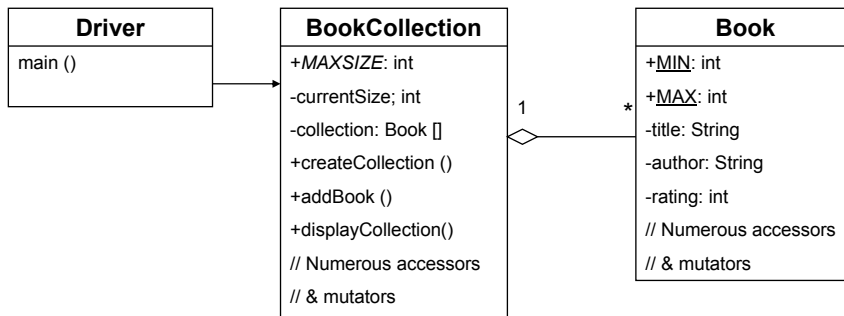| BookCollection |
| --- |
| +*MAXSIZE*: int |
| -currentSize; int |
| -collection: Book [] |
| +createCollection () |
| +addBook () |
| +displayCollection() |
| // Numerous accessors |
| // & mutators |

1   *

Book
+<u>MIN</u>: int
+<u>MAX</u>: int
-title: String
-author: String
-rating: int
// Numerous accessors
// & mutators

An Example Of Determining Object's State:
The Driver Class

```
public class Driver
{
  public static void main (String [] args)
  {
    BookCollection tamj = new BookCollection ();
    tamj.createCollection();
    tamj.displayCollection();
  }
}
```

An Example Of Determining Object's State:
The BookCollection Class

```
public class BookCollection
{
   public final static int MAX_SIZE = 4;
   private int currentSize;
   private Book [] collection;

   public BookCollection ()
   {
      currentSize = 0;
      collection = new Book [MAX_SIZE];
   }
```

An Example Of Determining Object's State:
The BookCollection Class (2)

```
public void createCollection ()
{
   int i;
   if (collection == null)
   {
      currentSize = 0;
      collection = new Book [MAX_SIZE];
   }
   if (currentSize != 0)
      System.out.println("Collection is not empty.");
   else
   {
      for (i = 0; i < MAX_SIZE; i++)
      {
         addBook();
      }
   }
}
```

An Example Of Determining Object's State: The BookCollection Class (3)

```
public int getCurrentSize () { return currentSize; }

public void setCurrentSize (int newSize) { currentSize = newSize; }

public void addBook ()
{
  if ((currentSize+1) < MAX_SIZE)
  {
    Book b = new Book ();
    b.setAllFields();
    collection[currentSize] = b;
    currentSize++;
  }
}
```

An Example Of Determining Object's State: The BookCollection Class (4)

```
public void displayCollection ()
{
  int i;
  int no;
  System.out.println("\nDISPLAYING COLLECTION");
  no = 1;
  for (i = 0; i < currentSize; i++)
  {
    System.out.println("\tBook #"+no);
    System.out.println("\tTitle: " + collection[i].getTitle());
    System.out.println("\tAuthor: " + collection[i].getAuthor());
    System.out.println("\tRating: " + collection[i].getRating());
    System.out.println();
    no++;
  }
}
} // End of the BookCollection class
```

An Example Of Determining Object's State: The Book Class

```
public class Book
{
    public static final int MIN = 1;
    public static final int MAX = 5;
    private String title;
    private String author;
    private int rating;

    public Book ()
    {
        title = "No title given";
        author = "No author listed";
        rating = -1;
    }
```

An Example Of Determining Object's State: The Book Class (2)

```
    public Book (String newTitle, String newAuthor, int newRating)
    {
        title = newTitle;
        author = newAuthor;
        rating = newRating;
    }

    public String getTitle () { return title; }
    public void setTitle (String newTitle) { title = newTitle; }
    public String getAuthor () { return author; }
    public void setAuthor (String newAuthor) { author = newAuthor; }
    public int getRating () { return rating; }
```

An Example Of Determining Object's State: The Book Class (3)

```
public void setRating (int newRating)
{
   if ((newRating >= MIN) && (newRating <= MAX))
      rating = newRating;
   else
      System.out.println("The rating must be a value between " +
                            MIN + " and " + MAX +  " inclusive");
}
```
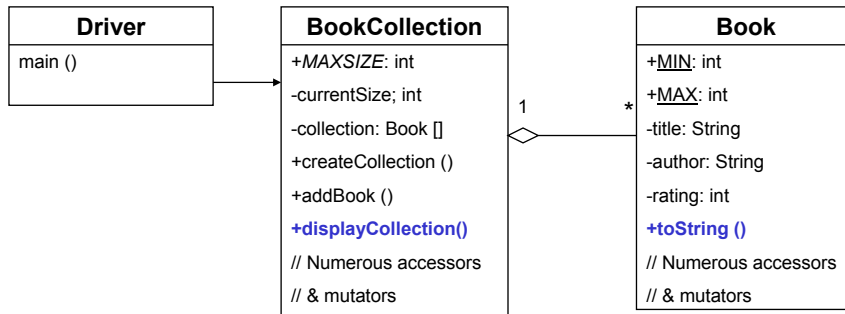
An Example Of Determining Object's State: The Book Class (4)

```
public void setAllFields ()
{
   System.out.print("Enter the title of the book: ");
   title = Console.in.readLine();
   System.out.print("Enter the author of the book: ");
   author = Console.in.readLine();
   do
   {
      System.out.print("How would you rate the book (" + MIN +
                     "=worst, " + MAX + "=best): ");
      rating = Console.in.readInt();
      if ((rating < MIN) || (rating > MAX))
         System.out.println("Rating must be a value between " +
                  MIN + " and " + MAX);
   } while ((rating < MIN) || (rating > MAX));
   Console.in.readLine();
}
} // End of class Book
```

An Revised Example Of Determining Object's State: UML Diagram

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

Driver	BookCollection	Book
main ()	+*MAXSIZE*: int	+<u>MIN</u>: int
	-currentSize; int	+<u>MAX</u>: int
	-collection: Book []	-title: String
	+createCollection ()	-author: String
	+addBook ()	-rating: int
	+displayCollection()	**+toString ()**
	// Numerous accessors	// Numerous accessors
	// & mutators	// & mutators

1   *

An Example Of Determining Object's State: The Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
     BookCollection tamj = new BookCollection ();
     tamj.createCollection();
     tamj.displayCollection();
   }
}
```

An Example Of Determining Object's State: The BookCollection Class

```
public class BookCollection
{
    public final static int MAX_SIZE = 4;
    private int currentSize;
    private Book [] collection;

    public BookCollection ()
    {
        currentSize = 0;
        collection = new Book [MAX_SIZE];
    }
```

An Example Of Determining Object's State: The BookCollection Class (2)

```
public void createCollection ()
{
    int i;
    if (collection == null)
    {
        currentSize = 0;
        collection = new Book [MAX_SIZE];
    }
    if (currentSize != 0)
        System.out.println("Collection is not empty.");
    else
    {
        for (i = 0; i < MAX_SIZE; i++)
        {
            addBook();
        }
    }
}
```

An Example Of Determining Object's State:
The BookCollection Class (3)

```
public int getCurrentSize () { return currentSize; }

public void setCurrentSize (int newSize) { currentSize = newSize; }

public void addBook ()
{
  if ((currentSize+1) < MAX_SIZE)
  {
    Book b = new Book ();
    b.setAllFields();
    collection[currentSize] = b;
    currentSize++;
  }
}
```

An Example Of Determining Object's State:
The BookCollection Class (4)

```
public void displayCollection ()
{
  int i;
  int no;
  System.out.println("\nDISPLAYING COLLECTION");
  no = 1;
  for (i = 0; i < currentSize; i++)
  {
    System.out.println("\tBook #"+no);
    System.out.println(collection[i]);
    System.out.println();
    no++;
  }
}
} // End of the BookCollection class
```

An Example Of Determining Object's State:
The Book Class

```
public class Book
{
    public static final int MIN = 1;
    public static final int MAX = 5;
    private String title;
    private String author;
    private int rating;

    public Book ()
    {
        title = "No title given";
        author = "No author listed";
        rating = -1;
    }
```

An Example Of Determining Object's State:
The Book Class (2)

```
    public Book (String newTitle, String newAuthor, int newRating)
    {
        title = newTitle;
        author = newAuthor;
        rating = newRating;
    }

    public String getTitle () { return title; }
    public void setTitle (String newTitle) { title = newTitle; }
    public String getAuthor () { return author; }
    public void setAuthor (String newAuthor) { author = newAuthor; }
    public int getRating () { return rating; }
```

An Example Of Determining Object's State:
The Book Class (3)

```
 public void setRating (int newRating)
 {
   if ((newRating >= MIN) && (newRating <= MAX))
     rating = newRating;
   else
     System.out.println("The rating must be a value between " +
                        MIN + " and " + MAX +  " inclusive");
 }
```

An Example Of Determining Object's State:
The Book Class (4)

```
 public void setAllFields ()
 {
   System.out.print("Enter the title of the book: ");
   title = Console.in.readLine();
   System.out.print("Enter the author of the book: ");
   author = Console.in.readLine();
   do
   {
     System.out.print("How would you rate the book (" + MIN +
                      "=worst, " + MAX + "=best): ");
     rating = Console.in.readInt();
     if ((rating < MIN) || (rating > MAX))
       System.out.println("Rating must be a value between " +
                MIN + " and " + MAX);
   } while ((rating < MIN) || (rating > MAX));
   Console.in.readLine();
 }
} // End of class Book
```

A Revised Example Of Determining Object's State: The Book Class (5)

```
public String toString ()
{
    String temp = new String ();
    temp = temp + "\tTitle: " + title + "\n";
    temp = temp + "\tAuthor: " + author + "\n";
    temp = temp + "\tRating: " + rating + "\n";
    return temp;
}
}   // End of class Book
```

You Should Now Know

- New terminology and concepts relevant to methods: message passing, method signatures, overloading of methods
- What is implementation hiding and what is the benefit of employing it
- Two types of relationships that can exist between classes: associations and aggregation
- Some specific issues and problems associated with Java references
  - The parameter passing mechanism that is employed for different types in Java
  - How does the assignment and comparison of references work in Java
  - What is the "this" reference: how does it work and when is used

You Should Now Know (2)

- More advanced concepts in the Java garbage collection process: the finalize method and how it fits into the garbage collection of memory referred to by a reference
- What is the difference between static and instance methods, and static and instance attributes and when to should each one be employed
- Classes and states:
  - What is meant by the state of an instance of a class
  - Debugging programs by examining the state of instances