

## **Introduction To Object-Oriented Programming**

This section includes introductions to fundamental object-oriented principles such as information hiding, overloading, relationships between classes as well the object-oriented approach to design.

James Tam

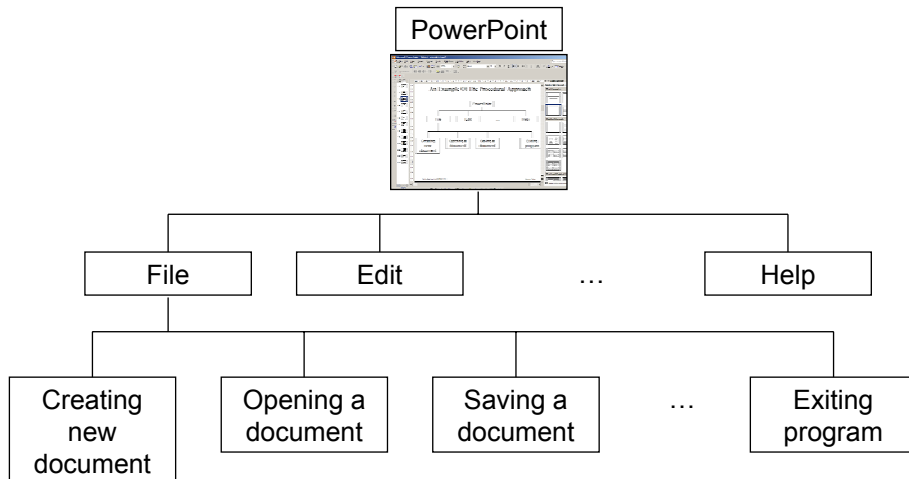
### **Reminder: What You Know**

- 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.

James Tam

## An Example Of The Procedural Approach

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



James Tam

## Object-Oriented Programming

- Break down the program into ‘physical’ components (*nouns*).
- Each of these physical components is an ‘object’.
- Objects include operations (functions which are referred to as ‘methods’ in the Object-Oriented paradigm) but also data (information about each object).
  - The methods can be determined by the actions (verbs) that each object should be able to complete.
  - The data can be determined by examining the type of information that each object needs to store.
- Example of an everyday object (from “*Starting out with Python*” by Tony Gaddis): An Alarm clock
  - What are the attributes of the alarm clock (information needed by the clock in order to properly function)
  - What are the methods of the alarm clock (operations that the clock must perform).

James Tam

## **Attributes Of The Alarm Clock**

- Current second (0 – 59)
- Current minute (0 – 59)
- Current hour (1 – 12)
- Alarm time (a valid hour and a valid minute)
- A flag to indicate if the alarm is set (true/false)

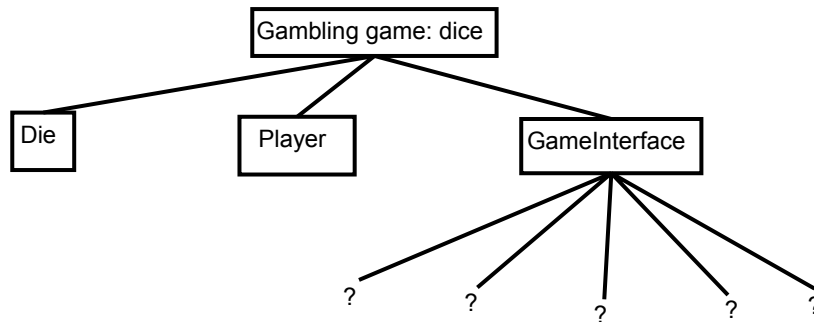
James Tam

## **Methods Of The Alarm Clock**

- Set time
- Set alarm time
- Turn alarm on/off

James Tam

## Example Decomposition Into Objects



James Tam

## Object-Oriented Programming: An Example

- How would the following program be decomposed into objects?
- What sort of attributes and methods would some of those objects consist of?

You are the Theron, apprentice to the Grey Lord, sent by your mentor to find the Firestaff hidden within the dungeon. You, as Theron Lord, first must enter the Hall of Champions (the first level of the dungeon) and find four champions to be your party. Each champion has a name, at least one class, and several physical attributes (such as strength, dexterity, wisdom, vitality etc.). The available classes are Fighter, Ninja, Wizard, and Priest.

The dungeon contains monsters, traps, doors, levers, and an assortment of other items to work with while you are exploring.



Game description from: <http://www.textfiles.com/>, Dungeon Master © FTL games.

James Tam

## 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)

James Tam

### 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
}
```

James Tam

## Defining A Java Class (2)

### Format of instance fields:

*<access modifier>*<sup>1</sup> *<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>*<sup>1</sup> *<return type>*<sup>2</sup> *<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)

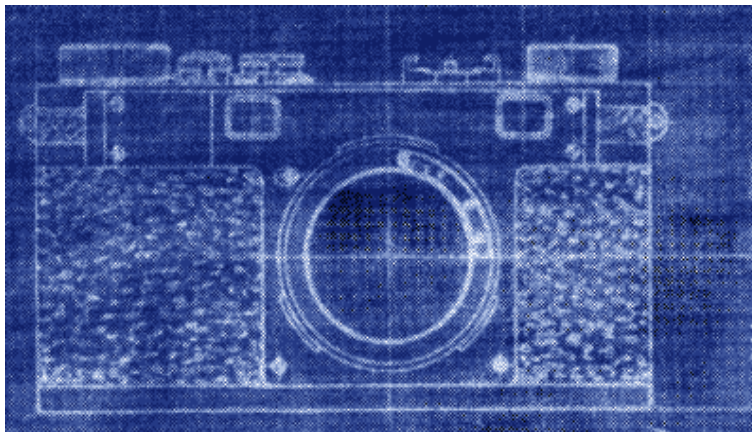
**Example** (complete class definition):

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

James Tam

## A Class Is Like A Blueprint

- It indicates the format for what an example of the class should look like (methods and attributes).
- No memory is allocated.



James Tam

## 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' (more on this later).

James Tam

## An Instance Is An Actual Example Of A Class

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



James Tam



## 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 ();

James Tam

## 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());
```

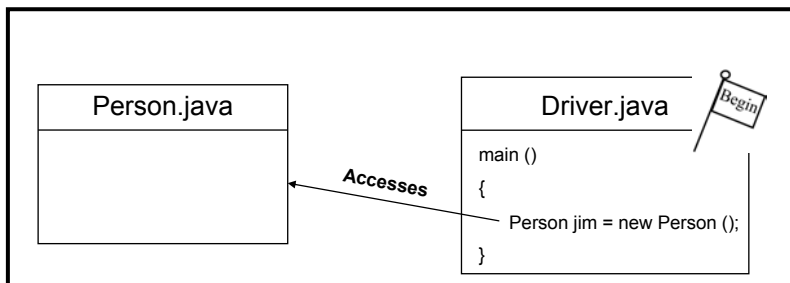
Note: In order to use the dot-operator "." the instance field or method cannot have a private level of access

James Tam

## Laying Out Your Program

- The program must contain a 'Driver' class (or equivalent).
- 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.

### Java program



James Tam

## Laying Out Your Program

- The code for each class should reside in its own separate file.

### Person.java

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

### Driver.java

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

James Tam

## **Putting It Altogether: First Object-Oriented Example**

- Example:

-The name of the online example is: firstOOExample.zip (extract all the files into the same directory)

```
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());
    }
}
```

James Tam

## **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;
    }
}
```

James Tam

## Compilation With Multiple Classes

- In the previous example there were two classes: ‘Driver’ and ‘Person’.
- One way (safest) to compile the program is to compile each source code (dot-Java) file:
  - javac Driver.java
  - javac Person.java
- However in this program a method of the Driver class refers to an instance of class Person.

```
public static void main (String [] args)
{
    Person jim = new Person ();
}
```
- The Java compiler can detect that this dependency exists.

James Tam

## Compilation With Multiple Classes (2)

- The effect *in this example* is that when the Driver class is compiled, the code in class Person *may* also be compiled.
  - Typing: “java Driver.java” produces a “Driver.class” file (or produces an updated compiled version if a byte code file already exists).
  - If there is no “Person.class” file then one will be created.
  - If a “Person.class” file already exists then an updated version will *not* be created (unless you explicitly compile the corresponding source code file).
- Moral of the story: when making changes to multiple source code (dot-Java files) make sure that you compile each individual file or at least remove existing byte code (dot-class) files prior to compilation.

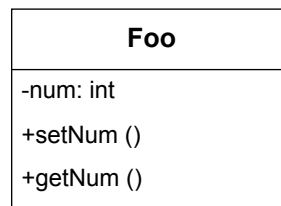
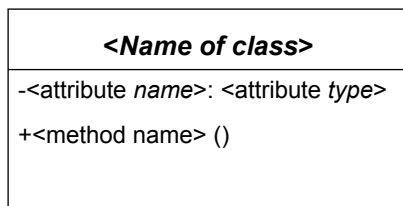
James Tam

## 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<sup>1</sup>
- For now avoid:
  - Defining instance fields / attributes for the Driver<sup>1</sup>
  - Defining methods for the Driver (other than the main method)<sup>1</sup>

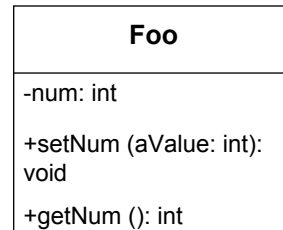
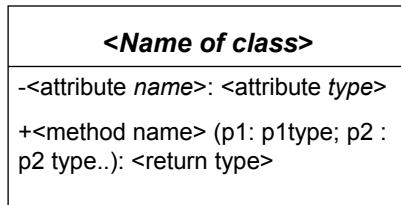
<sup>1</sup> Details will be provided later in this course

## UML<sup>1</sup> Representation Of A Class



<sup>1</sup> UML = Unified Modeling Language

## Class Diagrams With Increased Details



## Why Bother With UML?

- It's the standard way of specifying the major parts of a software project.
- It combined a number of different approaches and has become the standard notation.

## Why Represent A Program In Diagrammatic Form?

- Images are better than text for showing structural relations.

### **Text**

Jane is Jim's boss.

Jim is Joe's boss.

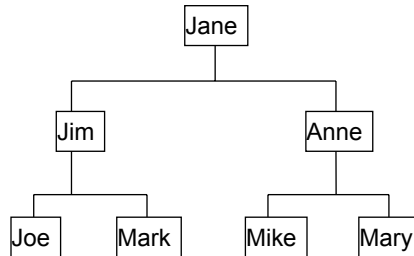
Anne works for Jane.

Mark works for Jim

Anne is Mary's boss.

Anne is Mike's boss.

### **Structure diagram**



James Tam

## 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; }
}
```

James Tam

## 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

James Tam

## 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**

James Tam



## 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

James Tam

## 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**

James Tam

## 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 Methods

```
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

James Tam

## 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

James Tam

## 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)

James Tam

## 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)

James Tam

## Referring To The Attributes And Methods Of A Class: Recap

1. Outside 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 () { .. }  
}

James Tam

## 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.

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

**NO!**

James Tam

## Shadowing

Scoping Rules:

1. Look for a local identifier (name of a 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

James Tam

## Encapsulation

- In Java: The ability bundle information (attributes) and behavior (methods) into a single entity.
- In Java this is done through a class definition.
- Other languages: C (“struct”), C++/Python (“class”), Pascal (“record”).

James Tam

## Information Hiding

- An important part of Object-Oriented programming and takes advantage of encapsulation.
- Protects the inner-workings (data) of a class.

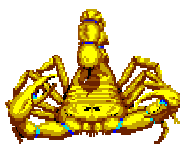


- Only allow access to the core of an object in a controlled fashion (use the *public* parts to access the *private* sections).

James Tam

## Illustrating The Need For Information Hiding: An Example

- Creating a new monster: “The Critter”
- Attribute: Height (must be 60” – 72”)



James Tam

## Illustrating The Need For Information Hiding: An Example

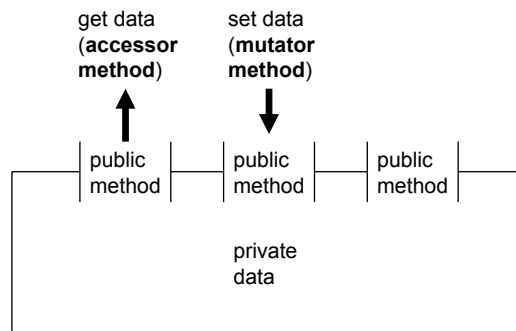
- Creating a new monster: “The Critter”
- Attribute: Height (must be 60” – 72”)



James Tam

## 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



James Tam



## 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;
}
```

James Tam

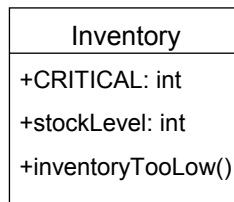
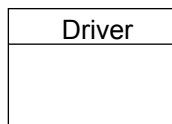
## How Does Hiding Information Protect The Class?

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

- e.g., range checking for inventory levels (0 – 100)

- The name of the online example is:

secondExampleNoProtection.zip



James Tam

## 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;
    }
}
```

James Tam

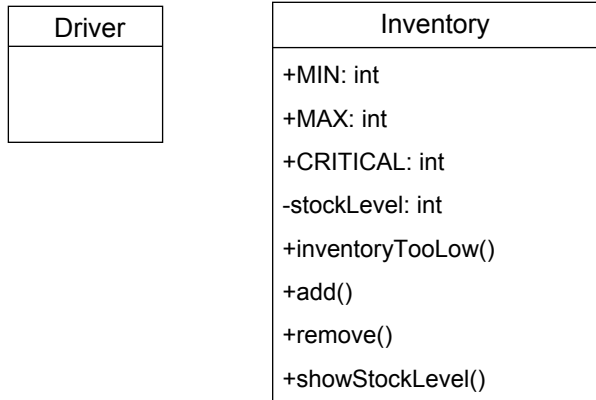
## The Driver Class

```
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);
    }
}
```

James Tam

## Utilizing Information Hiding: An Example

- The name of the complete online example is:  
thirdExampleInformationHiding



James Tam

## 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;
    }
}
```

James Tam

## 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
```

James Tam

## 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); }
}
```

James Tam

## 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;
    }
}
```

James Tam

## 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 private attributes via public mutators and accessors

James Tam

## **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 programming 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>

James Tam

## **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.

James Tam

## Method Signatures And Program Design

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

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

    }
}

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

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

    }
}

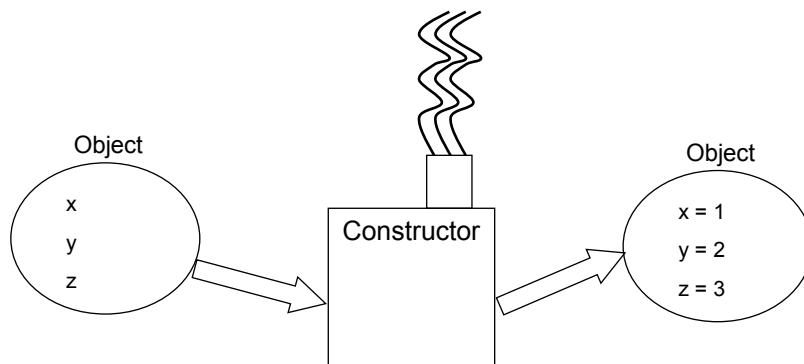
public static void main ()
{
    Foo f = new Foo ();
    f.fun ();
}
```

This change  
has broken  
me! ☹

James Tam

## 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.



James Tam

## 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.

James Tam

## 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";  
}
```

James Tam



## Overloading The Constructor

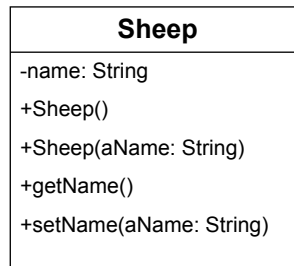
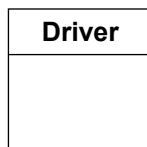
- Similar to other methods, constructors can also be overloaded
- Each version is distinguished by the number, type and order of the parameters

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

James Tam

## Constructors: An Example

- The name of the online example is:  
fourthExampleUsingConstructors.zip



James Tam

## 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);
    }
}
```

James Tam

## The Sheep Class (2)

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

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

James Tam

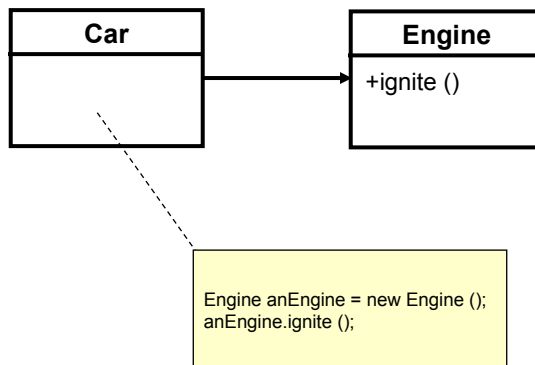
## The Driver Class

```
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();
    }
}
```

James Tam

## Association Relations Between Classes

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



James Tam

## 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 headLights;
    public start ()
    {
        anEngine.ignite ();
        headLights.turnOn ();
    }
}
```

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

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

James Tam

## 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).

James Tam

## Directed Associations (2)

- Bidirectional

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

-e.g.,

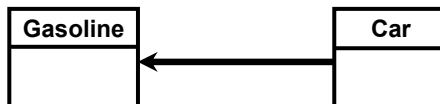
```
public class Student
{
    private Lecture [] lectureList = new Lecture [5];
    :
}

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

James Tam

## UML Representation Of Associations

### Unidirectional associations



### Bidirectional associations



James Tam

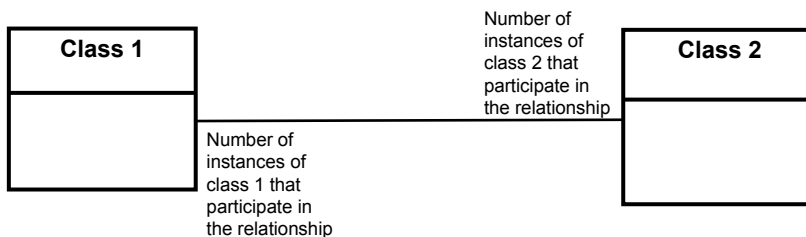
## Multiplicity

- 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

James Tam

## Multiplicity In UML Class Diagrams



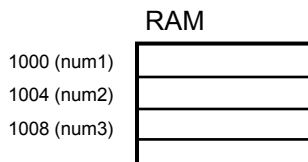
James Tam

## Review/New Topic: Hardware

- Computer memory: RAM (Random Access Memory).
- Consists of a number slots that can each store information.



- Normally locations in memory are not accessed via the numeric addresses but instead through variable names.



Picture from Computers in your future by Pfaffenberger B

James Tam

## Variables: Storing Data Vs. Address

- What you have seen so far are variables that store data.
  - Simple types: integers, real numbers, Booleans etc.
  - Composite types: arrays, strings etc.
- Other types of variables (e.g., Java variables which appear to be objects) hold addresses of variables.

```
Foo aFoo;  
aFoo = new Foo ();
```

  - The variable 'aFoo' is a reference to an object (contains the address of an object so it \*refers\* to an object).
  - Dynamic memory allocation: objects are created/instantiated only as needed.
- De-referencing: using an address to indirectly access data.
- Most times when you access instance variables in Java you directly access the object through the address of that object but knowing that an address is involved is important!

James Tam

## Variables: Storing Data Vs. Address (2)

- Even with high-level languages like Java, there *will* be times that programs will be working with the numeric address rather than the variable that the address is referring to.

James Tam


## De-Referencing: Java Example

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

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

```
f1 = f2;
```

Exactly what is  
being copied  
here?



James Tam



## Automatic Garbage Collection Of Java References

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

### References

f1(Address of a "Foo")



f2 (Address of a "Foo")



### Dynamic memory

Object (Instance of a "Foo")



Object (Instance of a "Foo")



James Tam

## Automatic Garbage Collection Of Java References (2)

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

### References

f1



f2



### Dynamic memory

Object (A "Foo")



Object (A "Foo")



James Tam

## Automatic Garbage Collection Of Java References (2)

- Dynamically allocated memory is automatically freed up when it is no longer referenced e.g., f2 = null; (a null reference means that the reference refers to nothing, it doesn't contain an address).

### References



### Dynamic memory

Object (A "Foo")



Object (A "Foo")



Free



James Tam

## Caution: Not All Languages Provide Automatic Garbage Collection!

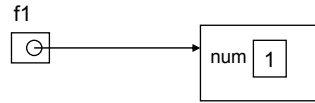
- Some languages do not provide automatic garbage collection (e.g., C, C++, Pascal).
- In this case dynamically allocated memory must be manually freed up by the programmer.
- Memory leak: memory that has been dynamically allocated but has not been freed up after it's no longer needed.
  - Memory leaks are a sign of poor programming style and can result in significant slowdowns.

James Tam

## The Finalize Method

- Example sequence:

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

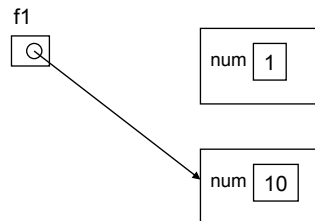


James Tam

## 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);
```

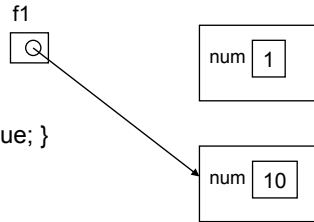


James Tam

## 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);
```

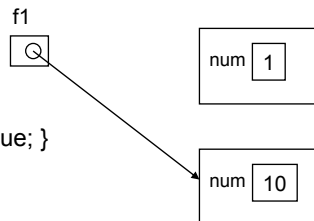


James Tam

## 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);
```



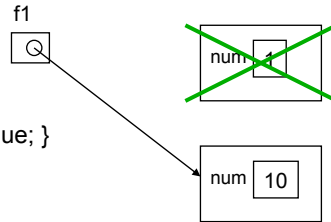
f1.finalize()

James Tam

## 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()

James Tam

## 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 (i.e., void)
  - 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.

James Tam

## Common Errors When Using References

- Forgetting to initialize the reference
- Using a null reference

James Tam

## 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

James Tam

## 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)

James Tam

## 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;  
    }  
    :      :  
}
```

James Tam

## 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.**

James Tam

## This ()

- It’s an invocation to the constructor of a class.
- It can be used when constructors have been overloaded.

James Tam



## More Of This ()

- Example:

```
public class Foo
{
    private int num1;
    private int num2;
    public Foo ()
    {
        num1 = 0;
        num2 = 0;
    }
    public Foo (int n1)
    {
        this ();
        num1 = n1;
    }
}
```

James Tam

## 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

James Tam

## **After This Section You Should Now Know (2)**

- What is method overloading and why is this regarded as good style
- 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