

Introduction To Object-Oriented Programming

Basic Object-Oriented principles such as encapsulation, overloading as well the object-oriented approach to design.

James Tam

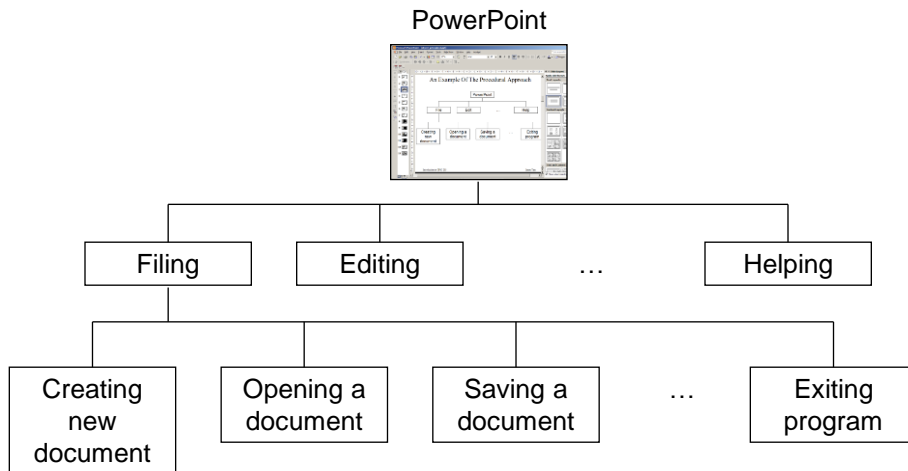
Reminder: What You Know

- There are different approaches to writing computer programs.
- They all involve decomposing your programs into parts.
- What is different between the approaches (how the decomposition occurs)/(criteria used for breaking things down")
- There approach to decomposition you have been introduced to thus far:
 - Procedural
 - Object-Oriented (~2 weeks for CPSC 231)

James Tam

An Example Of The Procedural Approach (Presentation Software)

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



James Tam

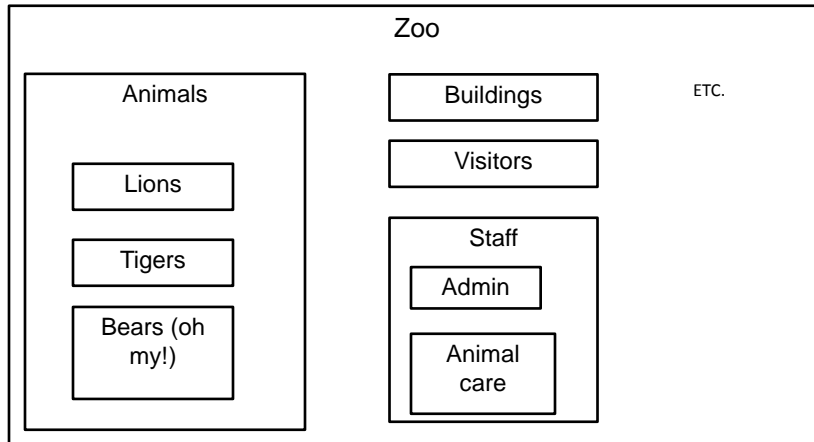
What You Will Learn

- How to break your program down into objects (**New term: "Object-Oriented programming"**)
- This and related topics comprise most of the remainder of the course

James Tam

An Example Of The Object-Oriented Approach (Simulation)

- Break down the program into entities (classes/objects - described with *nouns*)



James Tam

Classes/Objects

- Each class of object includes descriptive data.
 - Example (animals):
 - Species
 - Color
 - Length/height
 - Weight
 - Etc.
- Also each class of object has an associated set of actions
 - Example (animals):
 - Sleeping
 - Eating
 - Excreting
 - Etc.

James Tam

Example Exercise: Basic Real-World Alarm Clock

- What descriptive data is needed?
- What are the possible set of actions?



James Tam

Additional Resources

- A good description of the terms used in this section (and terms used in some of the later sections).

<http://docs.oracle.com/javase/tutorial/java/concepts/>

- A good walk through of the process of designing an object-oriented program, finding the candidate objects e.g., how to use the “find a noun” approach and some of the pitfalls of this approach.

<http://archive.eiffel.com/doc/manuals/technology/oosc/finding/page.html>

James Tam

Types In Computer Programs

- Programming languages typically come with a built in set of types that are known to the translator

```
int num;  
// 32 bit whole number (e.g. operations: +, -, *, /, %)  
  
String s = "Hello";  
// Unicode character information (e.g. operation:  
concatenation)
```

- Unknown types of variables cannot be arbitrarily declared!

```
Person tam;  
// What info should be tracked for a Person  
// What actions is a Person capable of  
// Compiler error!
```

James Tam

A Class Must Be First Defined

- A class is a new type of variable.
- The class definition specifies:
 - What descriptive data is needed?
 - Programming terminology: attributes = data (**New definition**)
 - What are the possible set of actions?
 - Programming terminology: methods = actions (**new definition**)
 - A method is the Object-Oriented equivalent of a function

James Tam

Defining A Java Class

Format:

```
public class <name of class>
{
    attributes
    methods
}
```

Example (more explanations coming shortly):

```
public class Person
{
    private int age; // Attribute
    public Person() { // Method
        age = in.nextInt();
    }
    public void sayAge () { // Method
        System.out.println("My age is " + age);
    }
}
```

James Tam

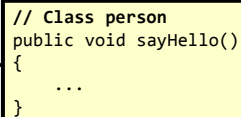
The First Object-Oriented Example

- Program design: each class definition (e.g., `public class <class name>`) must occur its own ".java" file).
- Example program consists of two files in the same directory:
 - (From now on your programs must be laid out in a similar fashion):
 - `Driver.java`
 - `Person.java`
 - Full example: located in UNIX under:
`/home/233/examples/intro_00/first_hello00`

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person aPerson = new Person();
        aPerson.sayHello();
    }
}
```



```
// Class person
public void sayHello()
{
    ...
}
```

```
I don't wanna say hello.
```

James Tam

Class Person

```
public class Person
{
    public void sayHello()
    {
        System.out.println("I don't wanna say hello.");
    }
}
```

James Tam

New Concepts: Classes Vs. Objects

- Class:
 - Specifies the characteristics of an entity but is not an instance of that entity
 - Much like a blue print that specifies the characteristics of a building (height, width, length etc.)



www.colorbox.com

James Tam

New Concepts: Classes Vs. Objects (2)

- Object:
 - A specific example or instance of a class.
 - Objects have all the attributes specified in the class definition



Images: James Tam

James Tam

main() Method

- Language requirement: There must be a `main()` method - or equivalent – to determine the starting execution point.
- Style requirement: the name of the class that contains `main()` is often referred to as the “Driver” class.
 - Makes it easy to identify the starting execution point in a big program.
- Do not instantiate instances of the Driver¹
- For now avoid:
 - Defining attributes for the Driver¹
 - Defining methods for the Driver (other than the `main()` method)¹

¹ Details may be provided later in this course

Compiling Multiple Classes

- One way (safest) is to compile all code (dot-Java) files when any code changes.
- Example:
 - `javac Driver.java`
 - `javac Person.java`
 - (Alternatively use the ‘wildcard’): `javac *.java`

Why Must Classes Be Defined

- Some classes are already pre-defined (included) in a programming language with a list of attributes and methods e.g., String
- Why don't more classes come 'built' into the language?
- The needs of the program will dictate what attributes and methods are needed.



James Tam

Defining The Attributes Of A Class In Java

- Attributes can be variable or constant (preceded by the 'final' keyword), for now stick to the former.
- **Format:**
<access modifier>¹ <type of the attribute> <name of the attribute>;

- **Example:**

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

1) Although other options may be possible, *attributes are almost always set to private* (more on this later).

James Tam

New Term: Object State

- Attributes: Data that describes each instance or example of a class.
- Different objects have the same attributes but the values of those attributes can vary
 - Reminder: The class definition specifies the attributes and methods for *all objects*
- Example: two 'monster' objects each have a health attribute but the current value of their health can differ
- The current value of an object's attribute's determines it's state.



Age: 35
Weight: 192



Age: 50
Weight: 125



Age: 0.5
Weight: 7

James Tam

Defining The Methods Of A Class In Java

Format:

```
<access modifier>1 <return type>2 <method name> (<p1 type> <p1 name>, <p2 type>  
<p2 name>...)  
{  
    <Body of the method>  
}
```

Example:

```
public class Person  
{  
    // Method definition  
    public void sayAge() {  
        System.out.println("My age is " + age);  
    }  
}
```

- 1) For now set the access modifier on all your methods to 'public' (more on this later).
- 2) Return types: includes all the built-in 'simple' types such as char, int, double...arrays and classes that have already been defined (as part of Java or third party extras)

James Tam

Parameter Passing: Different Types

Parameter type	Format	Example
Simple types	<code><method>(<type> <name>)</code>	<code>method(int x, char y) { ... }</code>
Objects	<code><method>(<class> <name>)</code>	<code>method(Person p) { ... }</code>
Arrays	<code><method>(<type> []... <name>)</code>	<code>method(Map [][] m) { ... }</code>

When calling a method, only the names of the parameters must be passed e.g., `System.out.println(num);`

James Tam

Return Values: Different Types

Return type	Format	Example
Simple types	<code><type> <method>()</code>	<code>int method() { return(0); }</code>
Objects	<code><class> <method>()</code>	<pre>Person method() { Person p = new Person(); return(p); }</pre>
Arrays	<code><type>[]... <method>()</code>	<pre>Person [] method() { Person [] p = new Person[3]; return(p); }</pre>

James Tam

What Are Methods

- Possible behaviors or actions for each instance (example) of a class.



Walk()
Talk()



Walk()
Talk()



Fly()



Swim()

James Tam

Instantiation

- **New definition:** Instantiation, creating a new instance or example of a class.
- Instances of a class are referred to as *objects*.
- **Format:**

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

- **Examples:**

```
Person jim = new Person();  
Scanner in = new Scanner(System.in);
```

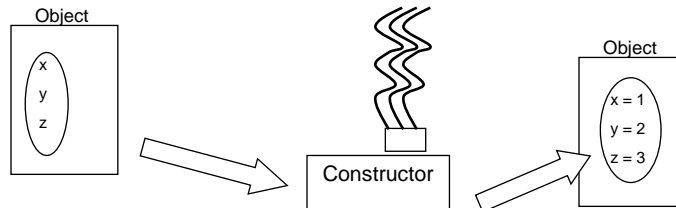
Creates new object

Variable names: 'jim',
'in'

James Tam

Constructor

- **New term:** A special method 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 e.g., `Person aPerson = new Person();`

Call to constructor
(creates something
'new')

```
class Person {  
    // Constructor  
    public Person() {  
        ...  
    }  
}
```

- Constructors can take parameters but **never** have a return type.

James Tam

New Term: Default Constructor

- Takes no parameters
- If no constructors are defined for a class then a default constructor comes 'built-into' the Java language.

- e.g.,

```
class Driver {  
    main() {  
        Person aPerson = new Person();  
    }  
}  
  
class Person {  
    private int age;  
}
```

James Tam

Calling Methods (Outside The Class)

- You've already done this before with pre-created classes!
- First create an object (previous slides)
- Then call the method for a particular variable.

- **Format:**

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

- **Examples:**

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

```
// Previously covered example, calling Scanner class method  
Scanner in = new Scanner(System.in);  
System.out.print("Enter your age: ");  
age = in.nextInt();
```

Scanner
variable



Calling
method



James Tam

Second Object-Oriented Example

- Learning concepts:
 - Attributes
 - Constructors
 - Accessing class attributes in a class method
- Location of full example:
`/home/233/examples/intro_00/second_attributeConstructor`

James Tam

Class Driver

```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person();
        jim.sayAge();
    }
}

public void sayAge() {
    System.out.println
    ("My age is " + age);
}
```

```
public Person() {
    Scanner in = new
    Scanner(System.in);
    System.out.print("Enter age: ");
    age = in.nextInt();
}
```

```
[csc firstOOExample 232 ]> java Driver
Enter age: 123
My age is 123
```

```
[csc firstOOExample 233 ]> java Driver
Enter age: 321
My age is 321
```

James Tam

Class Person

```
public class Person
{
    private int age;
    public Person()
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter age: ");
        age = in.nextInt();
    }

    public void sayAge()
    {
        System.out.println("My age is " + age);
    }
}
```

James Tam

Creating An Object

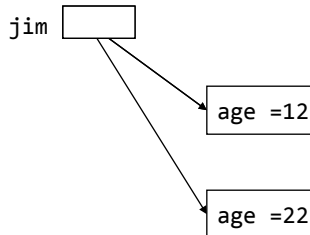
- Two stages (can be combined but don't forget a step)
 - Create a variable that refers to an object e.g., `Person jim;`
 - Create a **new** object e.g., `jim = new Person();`
 - The keyword 'new' calls the constructor to create a new object in memory
 - Observe the following

```
Person jim;
```

```
jim = new Person(12);
```

```
jim = new Person(22);
```

Jim is a reference to a Person object



James Tam

Terminology: Methods Vs. Functions

- Both include defining a block of code that be invoked via the name of the method or function (e.g., `print()`)

- **Methods** a block of code that is *defined within a class definition* (Java example):

```
public class Person
{
    public Person() { ... }

    public void sayAge() { ... }
}
```

- Every object that is an instance of this class (e.g., `jim` is an instance of a `Person`) will be able to invoke these methods.

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

James Tam

Terminology: Methods Vs. Functions (2)

- **Functions** a block of code that is *defined outside or independent of a class* (Python example – it's largely not possible to do this in Java):

```
# Defining method sayBye()
class Person:
    def sayBye(self):
        print("Hosta lavista!")

# Method are called via an object
jim = Person()
jim.sayBye()

# Defining function: sayBye()
def sayBye():
    print("Hosta lavista!")

# Functions are called without creating an object
sayBye()
```

James Tam

Methods Vs. Functions: Summary & Recap

Methods

- The Object-Oriented approach to program decomposition.
- Break the program down into classes.
- Each class will have a number of methods.
- Methods are invoked/called through an instance of a class (an object).

Functions

- The procedural (procedure = function) approach to program decomposition.
- Break the program down into functions.
- Functions can be invoked or called without creating any objects.

James Tam

Second Example: Second Look

Calls in Driver.java

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

```
jim.sayAge();
```

Person.java

```
public class Person {  
    private int age;  
  
    public Person() {  
        age = in.nextInt();  
    }  
  
    public void sayAge() {  
        System.out.println("My age  
        is " + age);  
    }  
}
```

More is needed:

- What if the attribute 'age' needs to be modified later?
- How can age be accessed but not just via a print()?

James Tam

Viewing And Modifying Attributes

1) New terms: Accessor methods: 'get()' method

- Used to determine the current value of an attribute

- Example:

```
public int getAge()  
{  
    return(age);  
}
```

2) New terms: Mutator methods: 'set()' method

- Used to change an attribute (set it to a new value)

- Example:

```
public void setAge(int anAge)  
{  
    age = anAge;  
}
```

James Tam

Version 2 Of The Second (Real) O-O Example

Location:

/home/233/examples/intro_00/third_accesorsMutators

James Tam

Class Person

- Notable differences: constructor is redesigned, `getAge()` replaces `sayAge()`, `setAge()` method added

```
public class Person
{
    private int age;
    public Person() {
        ...
        age = in.nextInt();
    }

    public void sayAge() {
        System.out.println("My age
            is " + age);
    }
}

public class Person
{
    private int age;
    public Person() {
        age = 0;
    }
    public int getAge() {
        return(age);
    }

    public void setAge
        (int anAge){
        age = anAge;
    }
}
```

James Tam

Class Driver

```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person();
        System.out.println(jim.getAge()); 0
        jim.setAge(21);
        System.out.println(jim.getAge()); 21
    }
}
```

James Tam

Calling Methods: Inside The Class

- You have seen this implicitly in the examples but here are the explicit syntax requirements you need to know well.
- Calling a method inside the body of the class (where the method has been defined)
 - You can just directly refer to the method (or attribute)

```
public class Person {
    private int age;

    public void birthday() {
        becomeOlder(); // access a method
    }

    public void becomeOlder() {
        age++; // access an attribute
    }
}
```

James Tam

Calling Methods: Outside The Class

- Calling a method outside the body of the class (i.e., in another class definition)
- The method must be prefaced by a variable (actually a reference to an object – more on this later).

```
public class Driver {
    public static void main(String [] args) {
        Person bart = new Person();
        Person lisa = new Person();
        // Incorrect! Who ages?
        becomeOlder();

        // Correct. Happy birthday Bart!
        bart.becomeOlder();
    }
}
```

James Tam

Constructors

- Constructors are used to initialize objects (set the attributes) as they are created.
- Different versions of the constructor can be implemented with different initializations e.g., one version sets all attributes to default values while another version sets some attributes to the value of parameters.
- **New term:** method overloading, same method name, different parameter list.

```
public Person(int anAge) {
    age = anAge;
    name = "No-name";
}

public Person() {
    age = 0;
    name = "No-name";
}

// Calling the versions (distinguished by parameter list)
Person p1 = new Person(100);    Person p2 = new Person();
```

James Tam

Example: Multiple Constructors

- Location:

/home/233/examples/intro_00/fourth_constructorOverloading

James Tam

Class Person

```
public class Person
{
    private int age;
    private String name;

    public Person()
    {
        System.out.println("Person()");
        age = 0;
        name = "No-name";
    }
}
```

James Tam

Class Person(2)

```
public Person(int anAge) {
    System.out.println("Person(int)");
    age = anAge;
    name = "No-name";
}

public Person(String aName) {
    System.out.println("Person(String)");
    age = 0;
    name = aName;
}

public Person(int anAge, String aName) {
    System.out.println("Person(int,String)");
    age = anAge;
    name = aName;
}
```

James Tam

Class Person (3)

```
public int getAge() {
    return(age);
}

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

public void setAge(int anAge) {
    age = anAge;
}

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

James Tam

Class Driver

```
public class Driver {
    public static void main(String [] args) {
        Person jim1 = new Person(); // age, name default
        Person jim2 = new Person(21); // age=21
        Person jim3 = new Person("jim3"); // name="jim3"
        Person jim4 = new Person(65,"jim4");
        // age=65, name = "jim4"

        System.out.println(jim1.getAge() + " " +
            jim1.getName());
        System.out.println(jim2.getAge() + " " +
            jim2.getName());
        System.out.println(jim3.getAge() + " " +
            jim3.getName());
        System.out.println(jim4.getAge() + " " +
            jim4.getName());
    }
}
```

```
Person()
Person(int)
Person(String)
Person(int, String)
```

```
0 No-name
21 No-name
0 jim3
65 jim4
```

James Tam

New Terminology: Method Signature

- Method signatures consist of: the type, number and order of the parameters.
- The signature will determine the overloaded method called:
Person p1 = new Person();
Person p2 = new Person(25);

James Tam

Overloading And Good Design

- Overloading: methods that implement similar but not identical tasks.
- Examples include class constructors but this is not the only type of overloaded methods:
 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>
- Benefit: just call the method with required parameters.

James Tam

Method Overloading: Things To Avoid

- Distinguishing methods solely by the order of the parameters.
 m(int, char);
 Vs.
 m(char, int);
- Overloading methods but having an identical implementation.
- Why are these things bad?

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()
    {
    }
}
```

After:

```
class Foo
{
    void fun(int num)
    {
    }
}
```

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

**This change
has broken
me! ☹**

James Tam

Graphical Summary Of Classes

- UML (Unified modeling language) class diagram
 - Source "*Fundamentals of Object-Oriented Design in UML*" by Booch, Jacobson, Rumbaugh (Dorset House Publishing: a division of Pearson) 2000
 - UML class diagram provides a quick overview about a class (later you we'll talk about relationships between classes)
- There's many resources on the Safari website:
 - <http://proquest.safaribooksonline.com.ezproxy.lib.ucalgary.ca/>
 - Example "**Sams Teach Yourself UML in 24 Hours, Third Edition**" (**concepts**)
 - Hour 3: Working with Object-Oriented
 - Hour 4: Relationships
 - Hour 5: Interfaces (reference for a later section of notes "hierarchies")

James Tam

UML Class Diagram

<Name of class>

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

Person

-age:int
+getAge():int
+getFriends():Person []
+setAge(anAge:int):void

James Tam

Why Bother With UML?

- It combined a number of different approaches and has become the standard notation.
- It's the standard way of specifying the major parts of a software project.
- Graphical summaries can *provide a useful overview* of a program (especially if relationships must be modeled)
 - Just don't over specify details

James Tam

Back To The 'Private' Keyword

- It syntactically means this part of the class cannot be accessed outside of the class definition.
 - You should **always** do this for variable attributes, *very rarely do this for methods* (more later).

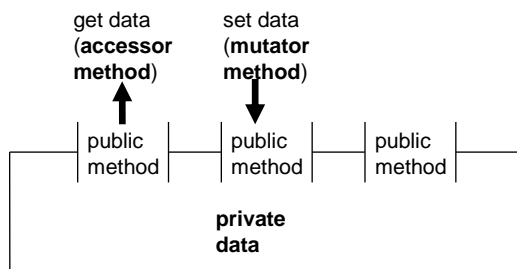
- Example

```
public class Person {
    private int age;
    public Person() {
        age = 12; // OK - access allowed here
    }
}
public class Driver {
    public static void main(String [] args) {
        Person aPerson = new Person();
        aPerson.age = 12; // Syntax error: program won't
                          // compile!
    }
}
```

James Tam

New Term: Encapsulation/Information Hiding

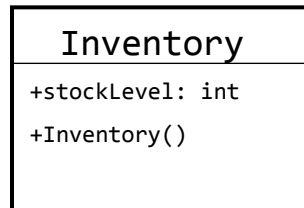
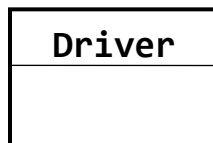
- Protects the inner-workings (data) of a class.
- Only allow access to the core of an object in a controlled fashion (use the *public* parts to access the *private* sections).
 - Typically it means public methods accessing private attributes via accessor and mutator methods.
 - Controlled access to attributes:
 - Can prevent invalid states
 - Reduce runtime errors



James Tam

How Does Hiding Information Protect Data?

- Protects the inner-workings (data) of a class
 - e.g., range checking for inventory levels (0 – 100)
- Location of the online example:
 - /home/233/examples/intro_00/fifth_noProtection



James Tam

Class Inventory

```
public class Inventory
{
    public int stockLevel;

    public Inventory()
    {
        stockLevel = 0;
    }
}
```

James Tam

Class Driver

```
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);
    }
}
```

Stock: 10

Stock: 20

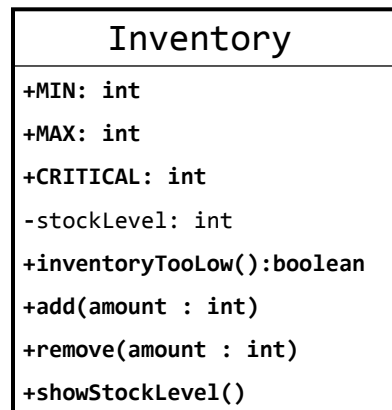
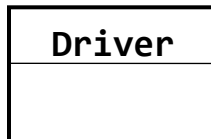
Stock: 120

Stock: -880

James Tam

Utilizing Information Hiding: An Example

- Location of the online example:
-/home/233/examples/intro_00/sixth_encapsulation



James Tam

Class Inventory

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

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

James Tam

Class Inventory (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;
    }
}
```

James Tam

Class Inventory (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 ()); Inventory: 10
        chinook.add (10);
        System.out.println(chinook.showStockLevel ()); Inventory: 20
        chinook.add (100);
        System.out.println(chinook.showStockLevel ()); Inventory: 20
        chinook.remove (21);
        System.out.println(chinook.showStockLevel ()); Inventory: 20
        // JT: The statement below won't work and for $
        // chinook.stockLevel = -999;
    }
}
```

James Tam

Add(): Try Adding 100 items to 20 items

```
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 Adding 100 item will cause stock to
        become greater than 100 units (overstock)
    }
} // End of method add
```

James Tam

Remove(): Try To Remove 21 Items From 20 Items

```
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
    {
        Removing 21 item will cause stock to
        become less than 0 units (understock)
        stockLevel = temp;
    }
}

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

James Tam

New Terms And Definitions

- Object-Oriented programming
- Class
- Object
- Class attributes
- Class methods
- Object state
- Instantiation
- Constructor (and the Default constructor)
- Method
- Function

James Tam

New Terms And Definitions (2)

- Accessor method (“get”)
- Mutator method (“set”)
- Method overloading
- Method signature
- Encapsulation/information hiding
- Multiplicity/cardinality

James Tam

After This Section You Should Now Know

- How to define classes, instantiate objects and access different part of an object
- What is a constructor and how is it defined and used
- What are accessor and mutator methods and how they can be used in conjunction with encapsulation
- What is method overloading and why is this regarded as good style
- How to represent a class using class diagrams (attributes, methods and access permissions) and the relationships between classes
- What is encapsulation/information-hiding, how is it done and why is it important to write programs that follow this principle

James Tam

Copyright Notification

- “Unless otherwise indicated, all images in this presentation are used with permission from Microsoft.”

slide 72

James Tam