Introduction To Object-Oriented Programming

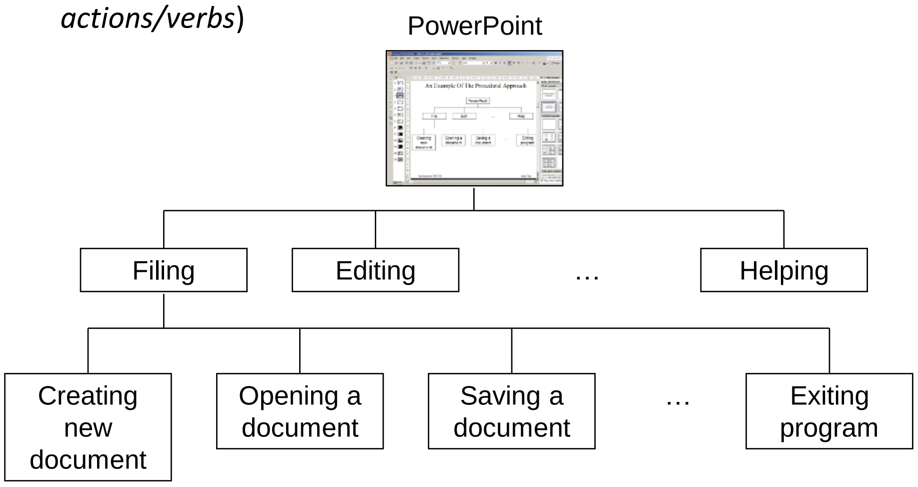
This section includes introductions to fundamental object-oriented principles such as encapsulation, overloading, relationships between classes as well the object-oriented approach to design.

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 is (how the decomposition occurs)/(criteria used)
- There approaches to decomposition you have been introduced to thus far (note: there are others):
  - Procedural
  - Object-Oriented.

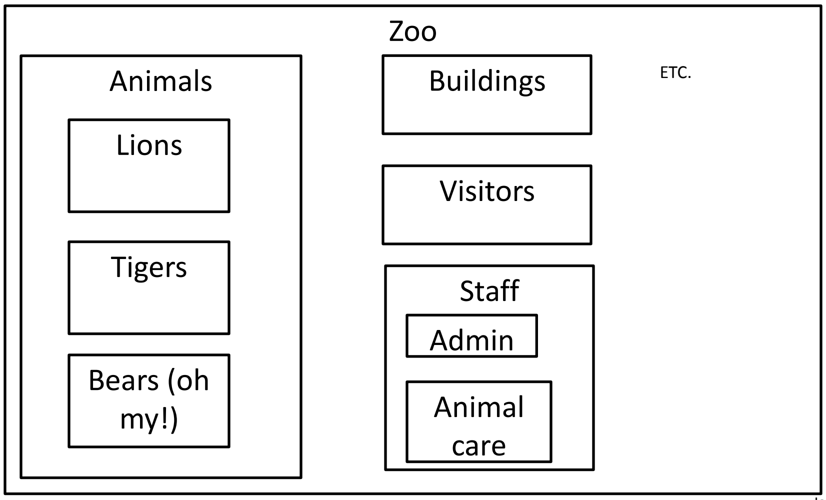
An Example Of The Procedural Approach (Presentation Software)

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



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):
    - Eating
    - Sleeping
    - 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

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 signed whole number
```

- 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**)

James Tam

Defining A Java Class

Format:

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

Example (more explanations coming):

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

James Tam

Defining The Attributes Of A Class In Java

- Attributes can be variable or constant (includes 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

What Are Attributes

- Data that describes each instance or example of a class.

```
Age: 0.5
Weight: 7
```

```
Age: 35
Weight: 165
```

```
Age: 30
Weight: 125
```

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
{
    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).

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

- **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);
```

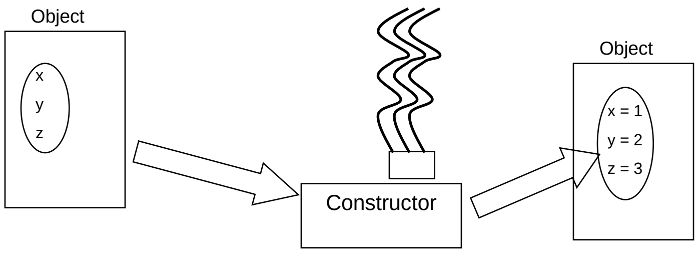
Creates new object

Variable names: 'jim', 'in'

James Tam

Constructor

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

Call to constructor (creates something 'new')

James Tam

Calling Methods

- 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
Scanner in = new Scanner(System.in);
System.out.print("Enter your age: ");
age = in.nextInt();
```

James Tam

Putting It All Together: First Object-Oriented Example

•Online example:

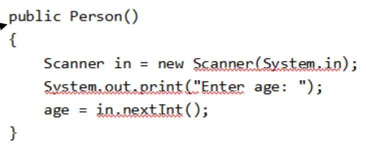
–It resides under the path:
`/home/233/examples/introOO/first`

–There's two Java files: `Driver.java, Person.java`

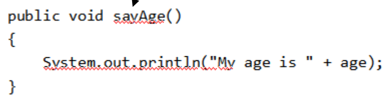
James Tam

Class Driver

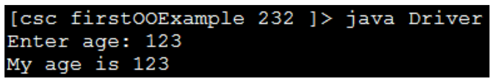
```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person();
        jim.sayAge();
    }
}
```



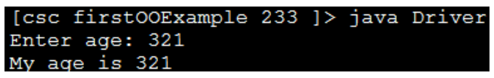
```
public Person()
{
    Scanner in = new Scanner(System.in);
    System.out.print("Enter age: ");
    age = in.nextInt();
}
```



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



```
[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

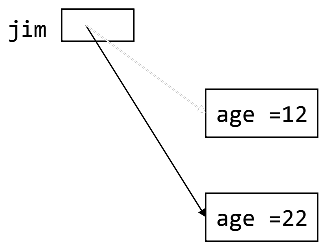
- Two stages (can be combined but don't forget a step)
  - Create a variable that refers to an object e.g., `Person jim;`
  - Create a *new* object e.g., `jim = new Person();`
    - `The keyword 'new'` calls the constructor to create a new object in memory
  - Observe the following

```
Person jim;
```

Jim is a reference to a Person object

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

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



`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 `Driver`[1]
- For now avoid:
  - Defining attributes for the `Driver`[1]
  - Defining methods for the `Driver` (other than the `main()` method)[1]

1 Details will be provided later in this course

Laying Out Your Program

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

Person.java

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

Driver.java

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

- All the Java source code files for a single program should reside in the same directory.

James Tam

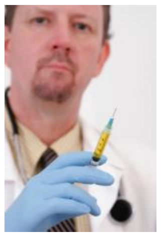
Compiling Multiple Classes

- 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`

James Tam

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

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 whose type is this class (in this case a `Person`) will be able to invoke these class 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!")

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

# Functions are called without creating an object
sayBye()


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

First 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()`?

Viewing And Modifying Attributes

1) Accessor methods: ‘`get()`’ method

- Used to determine the current value of an attribute
- Example:

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

2) 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

V2: First O-O Example

Location:

```
/home/233/examples/introOO/secondAccesorsMutators
```

James Tam

Class Person

Notable differences: constructor, `getAge()` replaces `sayAge()`

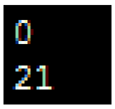
```
public class Person
{
    private int age;
    public Person() {
        age = 0;
    }
    public int getAge() {
        return(age);
    }

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

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

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

James Tam

Class Driver

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



```
0
21
```

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 non-default values (value of parameters passed in).

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

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

Example: Multiple Constructors

- Location:

  `/home/233/examples/introOO/thirdContructorOverloading`

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

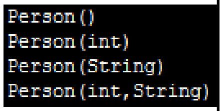
Class Person (3)

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

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

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

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

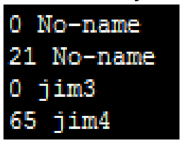
James Tam

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

Terminology: Method Signature

- Method signatures consist of: the type, number and order of the parameters.
- The signature can determine which method should be called:

```
Person p1 = new Person();
Person p2 = new Person(25);
```

James Tam

Terminology: Overloading

- Methods with the same name but a different method signature.
- Used for 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

James Tam

Method Overloading: Avoid

- Distinguishing methods solely by the order of the parameters.
- Overloading methods but having an identical implementation.
- Why?

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! ☹

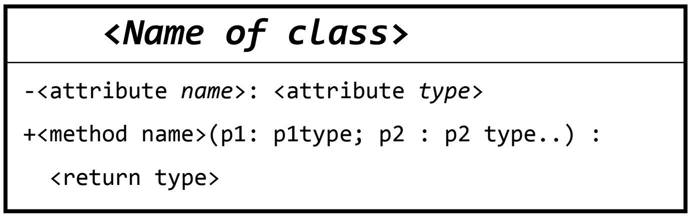
UML[1] Representation Of A Class

```
<Name of class>
-<attribute name>: <attribute type>
+<method name>()
```

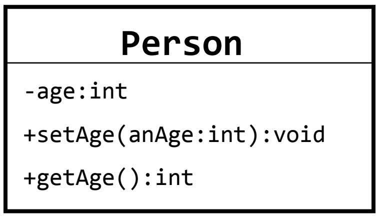
```
Person
-age: int
+getAge()
+setAge()
```

1 UML = Unified Modeling Language

UML[1] Class(Increased Details)



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



```
Person
-age:int
+setAge(anAge:int):void
+getAge():int
```

1 UML = Unified Modeling Language

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.

Local Variables Vs. Attributes

- Example:
  - What is/are local variables vs. attributes
  - When should something be local vs. an attribute

```
class Person
{
    private String [] childrenName = new String[10];
    private int age;

    public nameFamily()
    {
       int i;
       Scanner in = new Scanner(System.in);
       for (i = 0; i < 10; i++)
       {
           childrenName[i] = in.nextLine();
       }
    }
}
```

James Tam

Local Variables

- Local variables (also applies to local constants)
  - Declared within the body of a method.
  - Scope: They can only be used or accessed in that method (after they have been declared).
  - When to use: Typically store temporary information that is used only in that method.

```
public nameFamily()
{
    int i;
    Scanner in = new Scanner(System.in);
    for (i = 0; i < 10; i++)
    {
        childrenName[i] = in.nextLine();
    }
}
```

Scope of 'i' (int)

Scope of 'in' (Scanner)

Attributes

- Variable attributes (ignore constants for now)
  - Declared inside the body of a class definition but outside the body of that classes' methods.
  - Typically there is a separate attribute for each instance of a class and it lasts for the life of the object.
    - Created and initialized when the object is created by calling the constructor.

```
class Person
{
    private String [] childrenName = new String[10];
    private int age;
    /*
      For each person it's logical to track the age and
      the names any offspring.

    */
}
```

James Tam

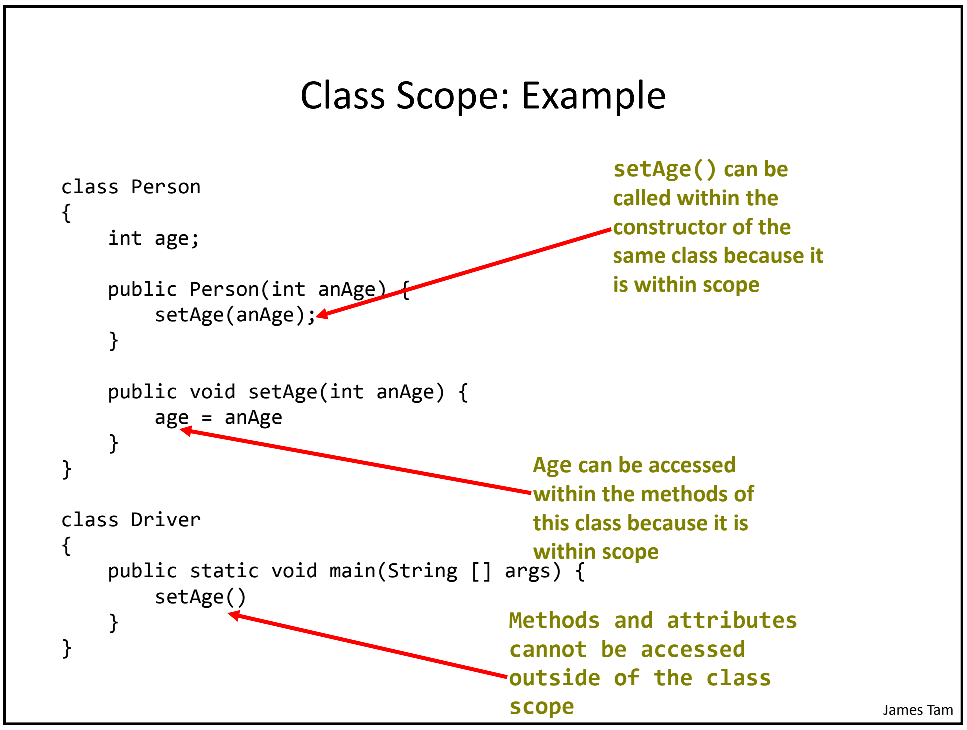
Scope Of Attributes (And Methods)

- Anywhere within the class definition.

```
class Person
{
    private int age;

    public nameFamily()
    {
       int i;
        Scanner in = new Scanner(System.in);
        for (i = 0; i < 10; i++)
        {
            childrenName[i] = in.nextLine();
        }
    }
    // The scope of any attributes or methods
    // declared or defined here is the entire class
    // definition.
}
```

$$\underbrace{\text{class body}}_{\text{Scope of 'nameFamily'}} \quad \underbrace{\text{class body}}_{\text{Scope of 'age'}}$$

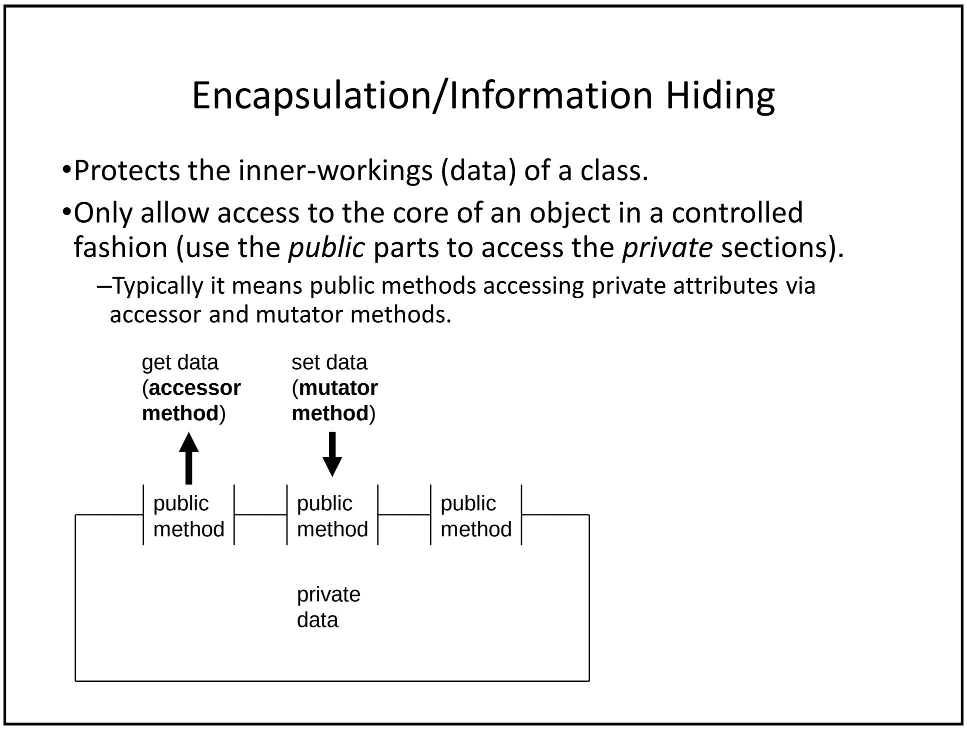
James Tam



Class Scope: Example

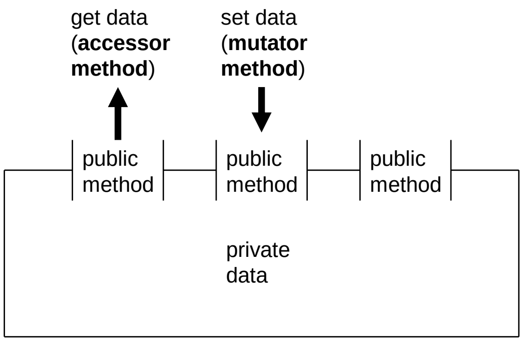
```
class Person
{
    int age;

    public Person(int anAge) {
        setAge(anAge);
    }

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

class Driver
{
    public static void main(String [] args) {
        setAge()
    }
}
```

setAge() can be called within the constructor of the same class because it is within scope
Age can be accessed within the methods of this class because it is within scope
Methods and attributes cannot be accessed outside of the class scope
James Tam



Encapsulation/Information Hiding

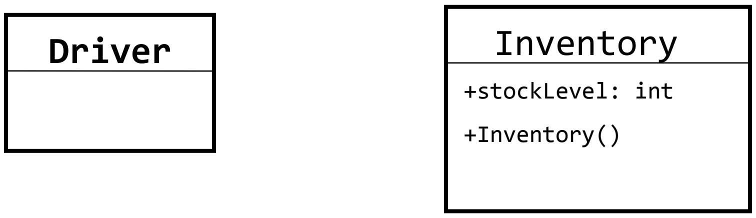
- Protects the inner-workings (data) of a class.
- Only allow access to the core of an object in a controlled fashion (use the *public* parts to access the *private* sections).
  - Typically it means public methods accessing private attributes via accessor and mutator methods.



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/introOO/fourthNoProtection`

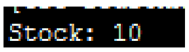


Class Inventory

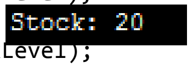
```
public class Inventory
{
    public int stockLevel;

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

Class Driver

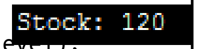
```
public class Driver
{
    public static void main (String [] args)
    {
        Inventory chinook = new Inventory ();
        chinook.stockLevel = 10;
        System.out.println ("Stock: " + chinook.stockLevel);
        chinook.stockLevel = chinook.stockLevel + 10;
        System.out.println ("Stock: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



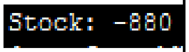
```
Stock: 10
```



```
Stock: 20
```



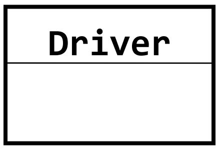
```
Stock: 120
```

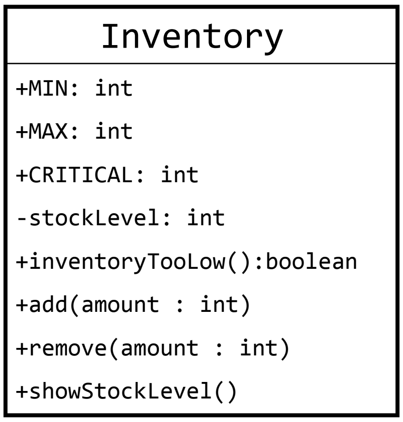


```
Stock: -880
```

Utilizing Information Hiding: An Example

- Location of the online example:
  - /home/233/examples/introOO/fifthEncapsulation





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);
    }
```

Class Inventory (2)

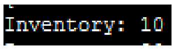
```
    public void add(int amount)
    {
        int temp;
        temp = stockLevel + amount;
        if (temp > MAX)
        {
            System.out.println();
            System.out.print("Adding " + amount +
                             " item will cause stock ");
            System.out.println("to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

Class Inventory (3)

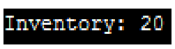
```
    public void remove(int amount)
    {
        int temp;
        temp = stockLevel - amount;
        if (temp < MIN)
        {
            System.out.print("Removing " + amount +
                             " item will cause stock ");
            System.out.println("to become less than " + MIN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

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

The Driver Class

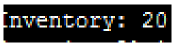
```
public class Driver
{
    public static void main (String [] args)
    {
        Inventory chinook = new Inventory ();
        chinook.add (10);
        System.out.println(chinook.showStockLevel ());
        chinook.add (10);
        System.out.println(chinook.showStockLevel ());
        chinook.add (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;
    }
}
```



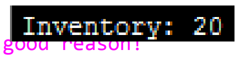
```
Inventory: 10
```



```
Inventory: 20
```



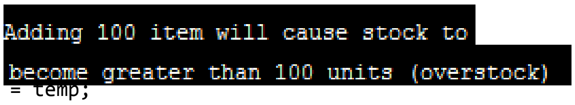
```
Inventory: 20
```



```
Inventory: 20
```

Add( ): Try Adding 100 items to 20 items

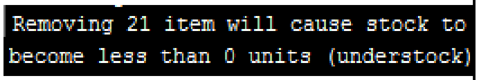
```
public void add(int amount)
{
    int temp;
    temp = stockLevel + amount;
    if (temp > MAX)
    {
        System.out.println();
        System.out.print("Adding " + amount +
                         " item will cause stock ");
        System.out.println("to become greater than " + MAX + " units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



```
Adding 100 item will cause stock to
become greater than 100 units (overstock)
```

Remove( ): Try To Remove 21 items from 20 items

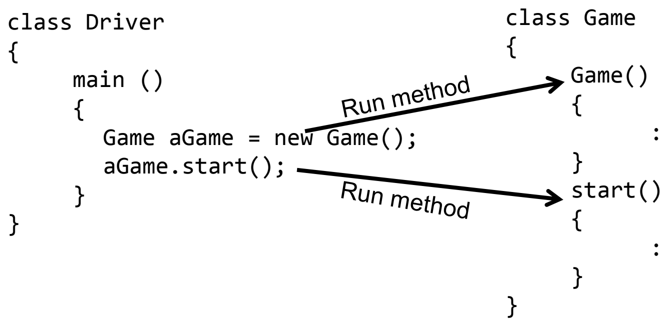
```
public void remove(int amount)
{
    int temp;
    temp = stockLevel - amount;
    if (temp < MIN)
    {
        System.out.print("Removing " + amount +
                         " item will cause stock ");
        System.out.println("to become less than " + MIN + " units
            (understock)");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

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



```
Removing 21 item will cause stock to
become less than 0 units (understock)
```

Messaging Passing

- Invoking the methods of another class.

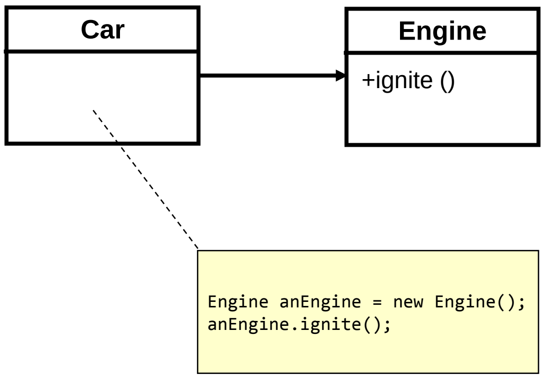


```
class Driver
{
     main ()
     {
       Game aGame = new Game();
       aGame.start();
     }
}
```

```
class Game
{
     Game()
     {
           :
     }
     start()
     {
           :
     }
}
```

Association Relations Between Classes

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



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

Associations Between Classes

- One type of association relationship is a 'has-a' relation (also known as "aggregation").
  - E.g. 1, A car <has-a> engine.
  - E.g. 2, A lecture <has-a> student.
- Typically this type of relationship exists between classes when a class is an attribute of another class.

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

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

Directed Associations

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

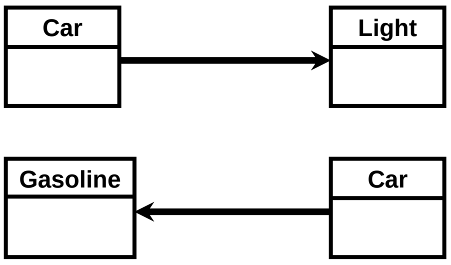
Directed Associations (2)

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

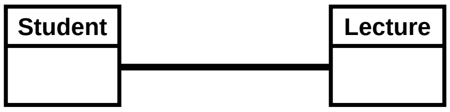
```
public class Student
{
    private Lecture [] myRegistration = new Lecture [5];
                    ...
}

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

UML Representation Of Associations

Unidirectional associations



Bidirectional associations

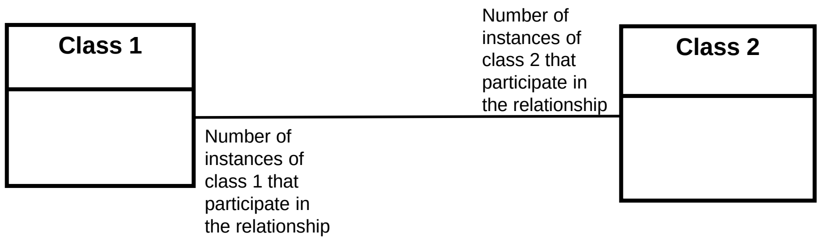


Multiplicity

- It indicates the number of instances that participate in a relationship

Multiplicity	Description
1	Exactly one instance
n	Exactly "n" instances {n: a positive integer}
n..m	Any number of instances in the inclusive range from "n" to "m" {n, m: positive integers}
*	Any number of instances possible

Multiplicity In UML Class Diagrams


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

Why Represent A Program In Diagrammatic Form (UML)?

- Images are better than text for showing structural relations.

Text

Jane is Jim's boss.

Jim is Joe's boss.

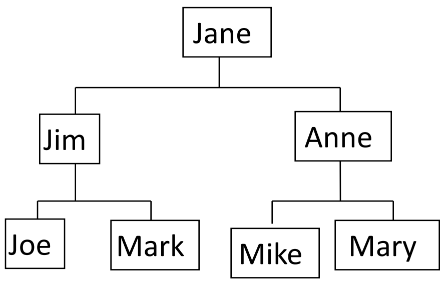
Anne works for Jane.

Mark works for Jim

Anne is Mary's boss.

Anne is Mike's boss.

Structure diagram



- UML can show relationships between classes at a glance

Relationships Between Classes

- Design rule of thumb.
- It can be convenient to create a relationship between classes (allow methods to be invoked/messages to be passed).
- But unless it is necessary for a relationship to exist between classes do not create one.
- That's because each time a method can be invoked there is the potential that the object whose method is called can be put into an invalid state (similar to avoiding the use of global variables to reduce logic errors).

James Tam

After This Section You Should Now Know

- How to define classes, instantiate objects and access different part of an object
- How to represent a class using class diagrams (attributes, methods and access permissions) and the relationships between classes
- What is encapsulation, 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 encapsulation
- What is method overloading and why is this regarded as good style

James Tam

After This Section You Should Now Know (2)

- Scoping rules for attributes, methods and locals
- 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

James Tam