

# Visual and Net Based Programming

## Object Oriented Programming

Agnik Saha

Department of Computer Science and Engineering

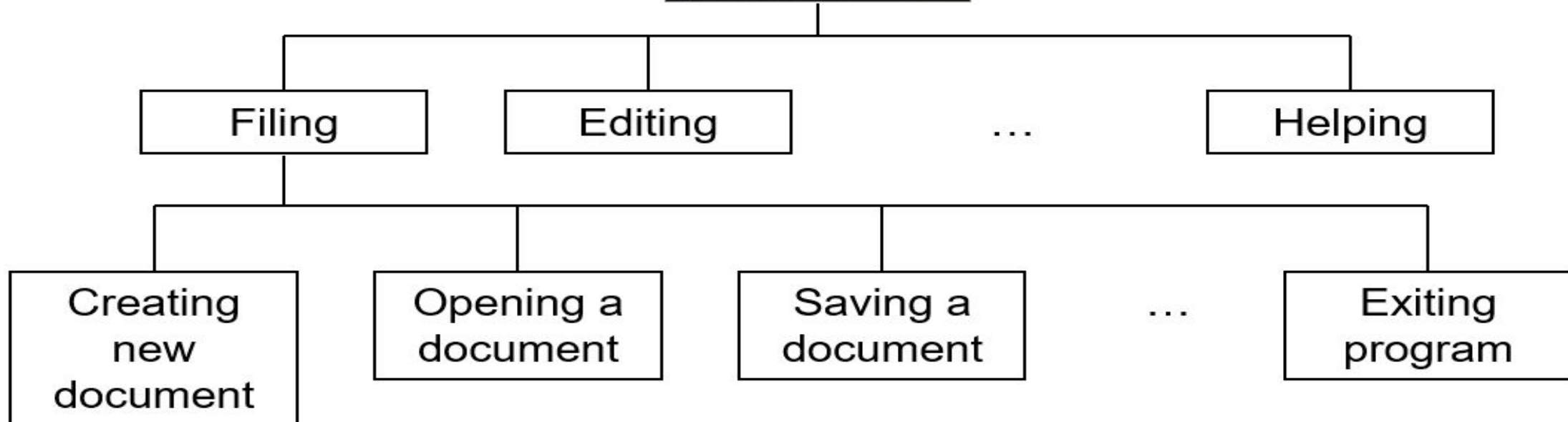
R. P. Shaha University

August 23, 2023

# An Example Of The Procedural Approach

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

## PowerPoint

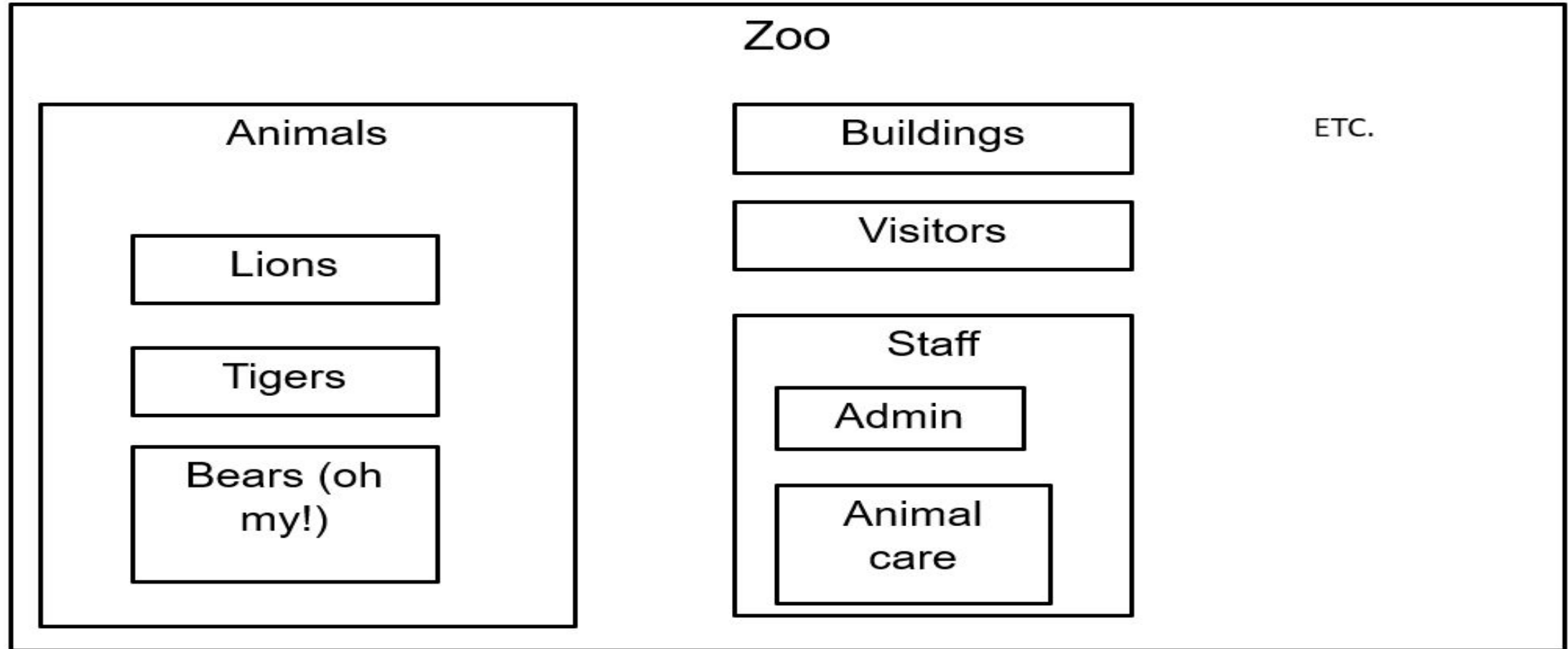


# What You Will Learn

- 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

# An Example Of The Object-Oriented Approach

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



# 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

# Example Exercise: Basic Real-World Alarm Clock

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



# Some Definitions

## What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

## What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

# 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!
```



# 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

# Defining A Java Class

## Format:

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

## Example (more explanations coming shortly):

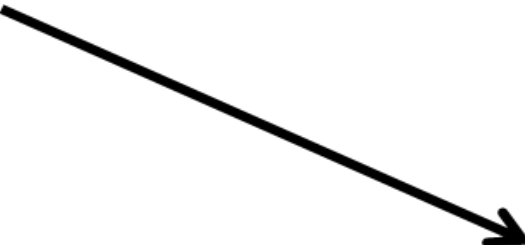
```
public class Person
{
    private int age; // Attribute
    public Person() { // Method
        age = in.nextInt();
    }
    public void sayAge() { // Method
        System.out.println("My age is " + age);
    }
}
```

# The First Object-Oriented Example

- Program design: each class definition (e.g., `public class <class name>`) must occur its own “dot-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/219/examples/intro\_OO/first\_helloOO

# 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.
```

# Class Person

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

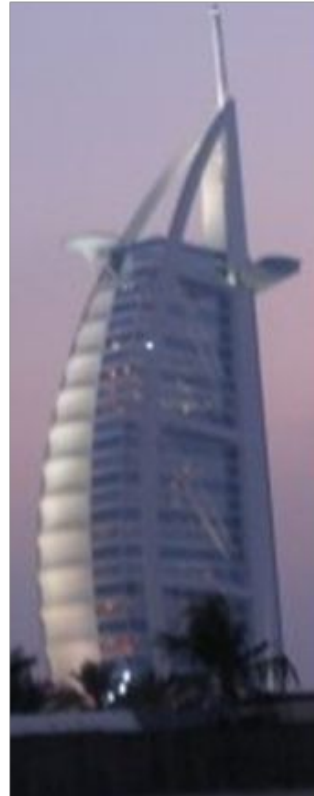
# New Concepts: Classes Vs. Objects

## 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.)

# New Concepts: Classes Vs. Objects

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



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



# 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.



# 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> <type of the attribute> <name of the attribute>;*

- Example:**

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

# Defining

- Attributes can be variable or constant (preceded by the 'final' keyword), for now stick to the former.

- **Format:**

*<access modifier> <type of the attribute> <name of the attribute>;*

- **Example:**

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

# New Term: Object State

- Similar to how two variables can contain different data.
- 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

# 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);  
    }  
}
```

# Parameter Passing: Different Types

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

# Return Values: Different Types

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



# What are methods

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



Walk()  
Talk()



Walk()  
Talk()



Fly()



Swim()

# 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**

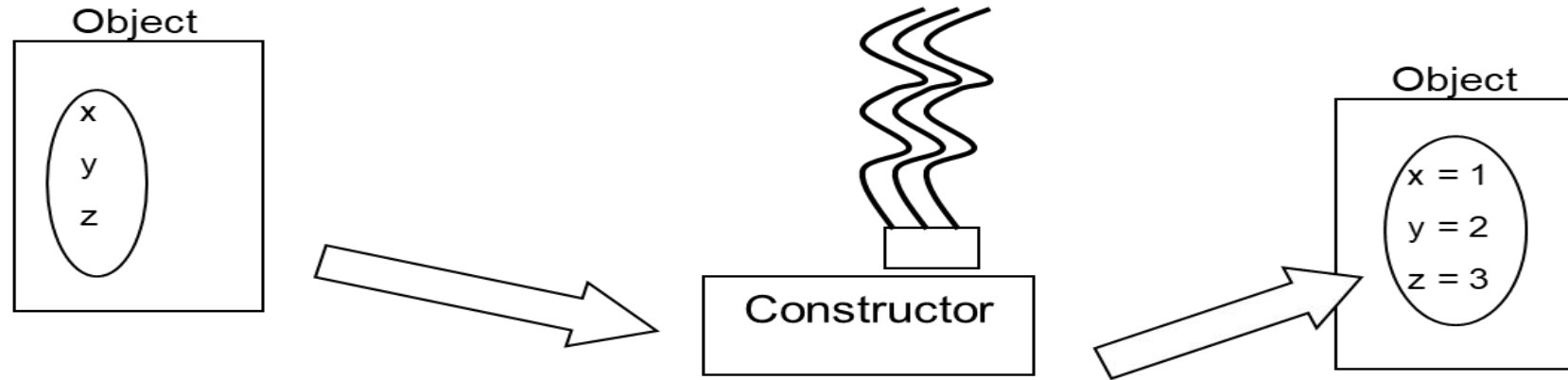


**Variable names: 'jim',  
'in'**



# 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.

# 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;  
}
```

Do previous example but with constructor setting arguments show how attributes can be set at run time rather than fixed

# 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**

# Calling Methods: Outside The Class You've Defined

- 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();  
    }  
}
```

# Calling Methods: Inside The Class

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  
    }  
}
```

# 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
```



# 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);
    }
}
```

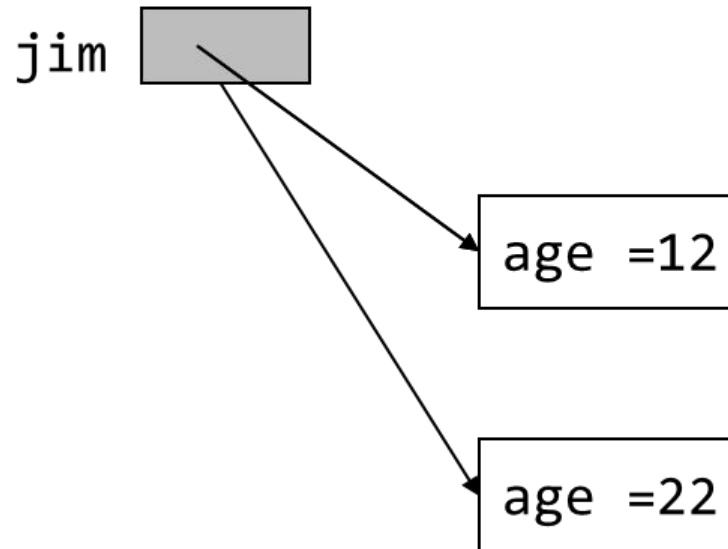
# Creating An Object

- 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**



# 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();
```

# Second Example

## 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) **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;  
}
```

# 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 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();
```

# Class Person

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



# 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;  
}
```

# 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;  
    }  
}
```

# 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
```

# 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);

# 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.

# 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?

# UML Class Diagram

***<Name of class>***

-<attribute name>: <attribute type>

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

-age:int

+getAge():int

+getFriends():Person []

+setAge(anAge:int):void

# 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



# Back To The 'Private' Keyword

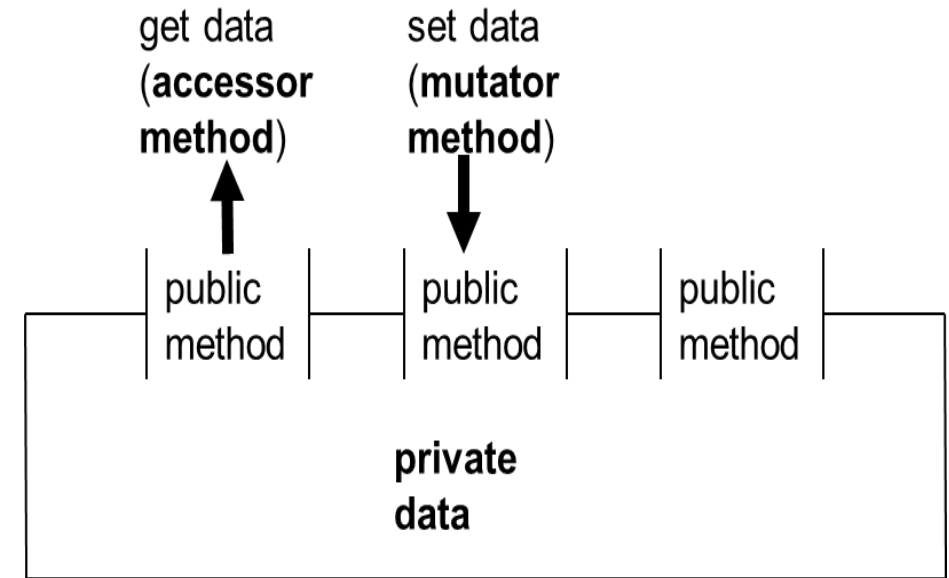
- It syntactically means this part of the class cannot be accessed outside of the class definition.

- You should **always** do this for variable attributes, *very rarely do this* for methods (more later).

```
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!  
    }  
}
```

# 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



# How Does Hiding Information Protect Data?

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



```
classDiagram
    class Driver
    class Inventory {
        +stockLevel: int
        +Inventory()
    }
```

**Driver**

**Inventory**

+stockLevel: int  
+Inventory()