

## Topic 3: Classes

Objective for last topic:

- convert skills from Python into Java – no big new concepts

Now we start learning about OOP – new concepts for many students.

## Object-Oriented Programming

Three important elements:

- encapsulation (*this topic*)
  - inheritance
  - polymorphism
- } (*next topic*)

Encapsulation means:

Putting several pieces of data together & viewing them as a unit  
(*an object*)

Includes "information hiding" – constrains how you can use & change data inside an object

## Simplest Kind of Class

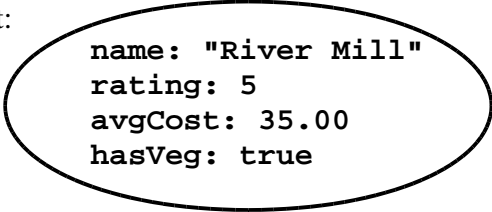
Objective: group several pieces of data together

Example: information about a restaurant

- name (string)
- rating (integer)
- average cost of a meal (double)
- vegetarian choices? (boolean)

## Object Example

Restaurant object:



```
name: "River Mill"  
rating: 5  
avgCost: 35.00  
hasVeg: true
```

**name, rating, avgCost, hasVeg:**  
attributes or "instance variables"

Class: template/blueprint for a kind of object  
Look at simple **Restaurant** class....

## Classes & Files

**Restaurant** class goes in file **Restaurant.java**

– a Java requirement

One class per file (usually).

simple example using **Restaurant** class....

note constructor & "." notation

**Dining.java** and **Restaurant.java** in same folder.

**Dining** code automatically finds **Restaurant** class.

## Vocabulary

**Restaurant** = class

object can be an *instance* of **Restaurant**

name, rating, etc: *instance variables* – every instance has them

other terms: attributes, fields

creating an object: *instantiating* the class

Every value in Java is either a primitive value

(number, char, boolean)

or an object.

arrays are special objects with their own syntax

**String** is a predefined Java class – contains characters & length.

## Instance Methods

Objects can also contain methods.

Anthropomorphize:

object remembers information (instance variables)

also knows how to do things (instance methods)

Example: add instance methods to **Restaurant** to

increase rating by one (but not go over 5)

change name

ask if rating is 3 or more

Uses for instance methods:

- return or output information about the object
- change information inside the object

## Special Method: toString

Common need: String representation of object for output

Convention: create **toString** method, returning String

Add **toString** to **Restaurant**....

Using **toString**:

```
System.out.println(rivMill.toString());
```

shortcut:

```
System.out.println(rivMill);
```

Java knows about **toString** – special method name.

If you try to print an object, Java automatically calls its **toString**.

## Writing Your Own Constructors

to create and initialize an object:

```
Restaurant hardRock = new Restaurant();
hardRock.name = "Hard Rock Cafe";
hardRock.rating = 4;
hardRock.avgCost = 12.50;
hardRock.hasVeg = true;
```

Create a constructor with parameters to do this in one step....

## Java Constructor Rules

1. If you don't define any constructors, you get a default constructor with no parameters.
2. If you define a constructor, you lose the default constructor.
3. If you want your own *plus* a zero-parameter constructor, you must write a zero-parameter constructor.

## Public vs. Private Instance Variables

```
public class Restaurant {
    ....
    public String name;
    ....
}
```

```
// in another class:
Restaurant r = new...
System.out.print(r.name);
r.name = "Golden Griddle";
// using r.name is legal
```

```
public class Restaurant {
    ....
    private int rating;
    ....
}
```

```
// in another class:
Restaurant r = new...
System.out.print(r.rating);
r.rating = 3;
// not legal!!!
```

## Why Use Private???

Many reasons, depending on circumstances:

1. Protect yourself against mistakes – get control over values (example: rating must be between 1 and 5)
2. Emphasis on what you can do with object, not format of data.
3. Possible to change representations without affecting user.
4. Create a "read-only" attribute.

## Get & Set Methods

get method: return value of an attribute

set method: change value of an attribute (if you want to allow this)  
often includes checks

These are ways to access & change values of instance variables,  
even if private.

## Changes To Restaurant Class

- make name private so it can't be changed
- make average cost private & provide general query methods only
- make rating private to allow changes in representation later and to prevent illegal values

## New Example: Employee class

useful for a payroll program....

## Class Interface vs. Implementation

Interface: How to use the class.

- names & types of *public* variables and methods
- plus comments/external documentation describing use

Implementation: Inner workings of the class

- *private* variables and methods
- method bodies (how methods work)

Goal of information hiding:

- Provide abstract view of class for users (only what they need to know)
- You can change the implementation without affecting users

Who are "users" of a class?

- other classes in program

## “this”

Sometimes useful to refer to whole object inside an object method.

Example: Suppose there's a method in another class that takes an **Employee** as a parameter:

```
Accounting.writeCheck(Employee e) {...
```

We want to call from an instance method:

```
public void zero() {
    Accounting.writeCheck(this);
    payOwed = 0;
} // end zero
```

## Another use for “this”

```
// Constructor uses awkward parameter names to avoid
// confusion
public Employee(String theName, String theTitle,
                double theWage) {
    name = theName;
    jobTitle = theTitle;
    wage = theWage;
    ....
}
```

```
// Alternate version using "this"
public Employee(String name, String jobTitle,
                double wage) {
    this.name = name;
    this.jobTitle = jobTitle;
    this.wage = wage;
    ....
}
```

## Overloaded Constructors

```
// create employee - general case
public Employee(String theName, String title, double w) {
    name = theName;
    jobTitle = title;
    payOwed = 0;
    if (w < 0) {
        System.out.println("Error");
        wage = 0;
    }
    else
        wage = w;
} // end constructor
```

```
// create employee with default starting wage
public Employee(String theName, String title) {
    name = theName;
    jobTitle = title;
    payOwed = 0;
    wage = 10.0;
} // end constructor
```

duplicated code

## A Better Way

```
// create employee - general case
public Employee(String theName, String title, double w) {
    name = theName;
    jobTitle = title;
    payOwed = 0;
    if (w < 0) {
        System.out.println("Error");
        wage = 0;
    }
    else
        wage = w;
} // end constructor
```

```
// create employee with default starting wage
public Employee(String theName, String title) {
    this(theName, title, 10.0); // call to other constructor
} // end constructor
```

## Constants

In constructor from last slide:

```
public Employee(String theName, String title) {  
    this(theName, title, 10.0);  
} // end constructor
```

Problem with using 10.0 here?

Better version using named constant:

```
public static final double MINIMUM_WAGE = 10.0;  
public Employee(String theName, String title) {  
    this(theName, title, MINIMUM_WAGE);  
} // end constructor
```

Why is it OK for the constant to be public?

## Class Variables (Static)

**name**: instance variable

Every **Employee** object has a name

A **name** is a property of a particular **Employee**

A class variable is a property of a whole class.

One value, visible to all the instances of the class.

Example for **Employee**: class variable to keep track of maximum wage being paid

Initially set to zero (at start of program)

Updated when?

## Class Method

Good information hiding:

**maxWage** is private

use public “get” method to access

other classes may not change directly

To call a class method from outside the class:

use class name, not object name

```
Employee.getMaxWage()
```

## constants & static

```
public static final double MINIMUM_WAGE = 10.0;
```

Why is the constant static?

## Static-Only Classes

Example: **Math** class

contains useful constants and methods:

```
Math.PI
Math.sqrt()
Math.log()
etc.
```

no instance variables

everything is static

Typical Java program:

- one static-only "main" class
- other classes containing mix of static & instance data

## References & Aliases

```
Employee e = new Employee(....);
```

What does Java do?

1. finds a free spot in memory
2. creates a new **Employee** object in that memory
3. variable **e** holds a *reference* to that object – its address in memory

Possible to have two variables referring to the same object --  
two *aliases* for the object.

When you pass an object as a parameter, you're passing a reference.  
Parameter & argument are two aliases for the same object.

## Aliasing Example (1)

```
Employee a = new Employee("Mickey", "mouse", 10);
Employee b = new Employee("Donald", "duck", 20);
Employee c = b;
c.raise(5);
b.raise(5);
a.raise(5);
System.out.println(a);
System.out.println(b);
System.out.println(c);
```

## Aliasing Example (2)

```
Employee d = new Employee("d", "d job", 10);
Employee e = new Employee("e", "e job", 10);
Employee f = new Employee("f", "f job", 10);
f = e;
e = d;
d = f;
d.pay(5);
d = e;
d.pay(7);
d = f;
d.pay(3);
System.out.println(d);
System.out.println(e);
System.out.println(f);
```

# Immutable Classes

A class is *immutable* if:

- all instance variables are private
- there are no methods that change the instance variables

Once an immutable object is created, its contents never change.

How do the instance variables get values?

# Example: String

In Java, **String** is an immutable class.

All Strings are immutable – can't be changed.

```
String s = "CISC 124";  
s = s.substring(0,4);
```

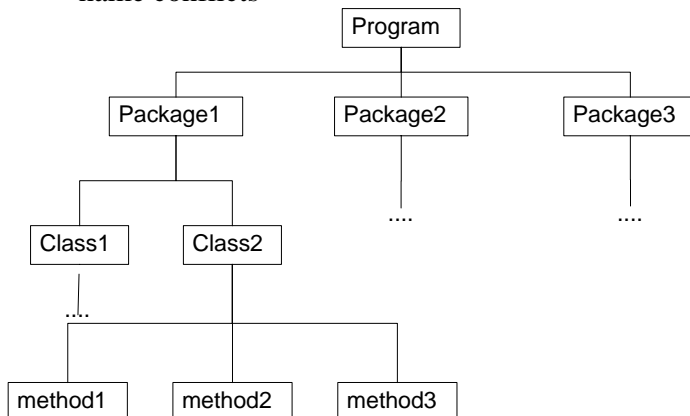
Have we changed a string??

# Packages

Package = collection of classes

Reasons to use packages:

- organizing large programs
- distribution
- name conflicts



# java.lang

Package **java.lang** contains very commonly used classes.

Automatically imported into your program – no import statement needed.

Examples:

**String**

**System**

Digression: Java API documentation

<https://docs.oracle.com/javase/8/docs/api/>



## null

Special value: **null**.

Any object type can take the value **null**:

```
String s = null;
Employee e = null;
Scanner sc = null;
```

....

Means "no object" -- like **none** in Python.

Implemented as an address of zero.

## Default Values

If an instance/class variable isn't explicitly initialized, it gets a default value:

```
numbers: 0
boolean: false
characters: '\0'
objects: null
```

Different rule for local variables (inside methods): If variable isn't initialized before use it's an error.

## One More Example: Time

Things to note:

- overloaded constructors, including copy constructor
- use of static methods
- two kinds of addition methods (one instance, one static)
- **equals** method

## Representation Choices

Class as it is:

- remembers hour, minute, second
- set/get methods, **toString** are simple
- arithmetic is more difficult

Another possibility:

- replace hour/minute/second with total seconds
- arithmetic is simple
- set/get and **toString** are more difficult
- saves space

If we changed representation:

- what would implementor have to change?
- what would users have to change?

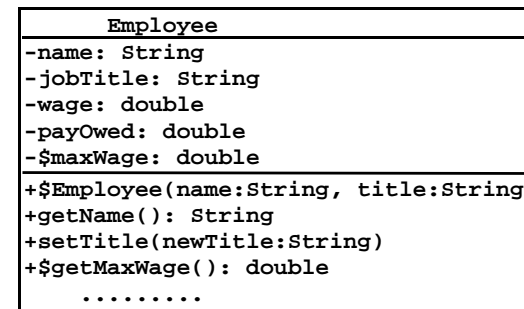
# Convenient Notation: UML

Unified Modeling Language  
 different kinds of diagrams for representing program design  
 UML class diagrams: shows contents of classes

\*\*\*I will not ask you to *write* UML for assignments/quizzes/exams.  
 \*\*\*I may ask you to *read* simple UML class diagrams

# UML

Detailed class diagram for the Employee class:



← name of class

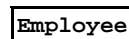
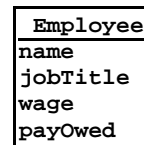
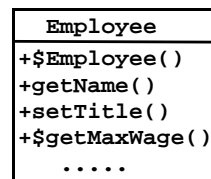
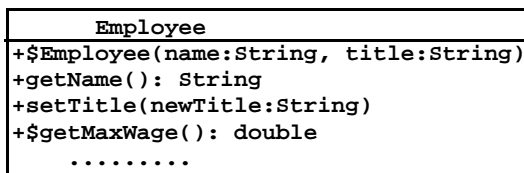
← attributes

← methods

"+" means public  
 "-" means private  
 "\$" means static

# UML

Previous diagram showed lots of detail. Other possible class diagrams for Employee:

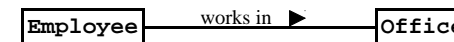


Amount of detail depends on:

- stage of planning
- audience
- purpose of diagram

# UML

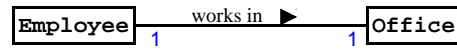
Class diagrams may also show relationships between classes



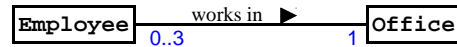
"works in" is name of relationship  
 triangle arrow shows direction (an Employee *works in* an Office)

# UML

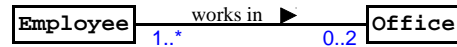
Relationships may have *multiplicity*:



1-1 correspondence between employees and offices



Every office has 0-3 employees working in it.  
Every employee has exactly one office.



- some employees with no office or two offices
- no empty offices
- no set upper limit on number of people in an office

# Complex Class Diagram

