ALGONQUIN COLLEGE

CST8132 OBJECT ORIENTED PROGRAMMING

Properties of OOP

- Encapsulation
- Relationships
- Packages

Professor : Dr. Anu Thomas

Email: thomasa@algonquincollege.com

Office: T314

Today's Topics

- Lab 2 – Store Management System I
- Objects
  - Attributes and Behaviors of Objects
  - Relationships between Objects
- Packages

Objects

- Objects are specified by Classes
- Objects have
  - State (properties or attributes or instance variables or fields)
  - Behavior (methods)

Object State

- Object state is given by the instance variables
- Instance variables are declared at the top of the class outside of any method
  - Variables declared inside a method are called local variables
  - they exist only when the method is running
- Instance variables can be primitive variables or reference variables
  - Primitive: the variable contains the actual value
    - `int num = 10;`
  - Reference
    - `Employee emp = new Employee();`

Instance Variables vs Local Variables

- Local variables
  - Declared INSIDE a block (method, for-loop, if statement etc.)
  - These are TEMPORARY variables used for programming purposes
  - These exist only when the block is running
  - Every time the block runs, the variable is born again (no old values)
  - These do not represent the state of the object
- Instance variables
  - Declared OUTSIDE methods in the class body
  - Come into existence when the object is instantiated (with new keyword)
  - Together these represent the state of the object

Common Mistake (instance vs local)

- Hiding an instance variable with an accidental local variable declaration

```
public class Lecture2 {
  private int x;
  public void doit(int val){
   int x = val;
  }
  public void printIt(){
   System.out.println("x : " + x);
  }

  public static void main(String[] args) {
   Lecture2 lec = new Lecture2();
   lec.doit(5);
  lec.printIt();
  }
}
```

Methods

- In OOP, we set up object(s) and start the processing by using (one of) the object(s) to call one of its methods

- We do two different things with a method
  - Define a method

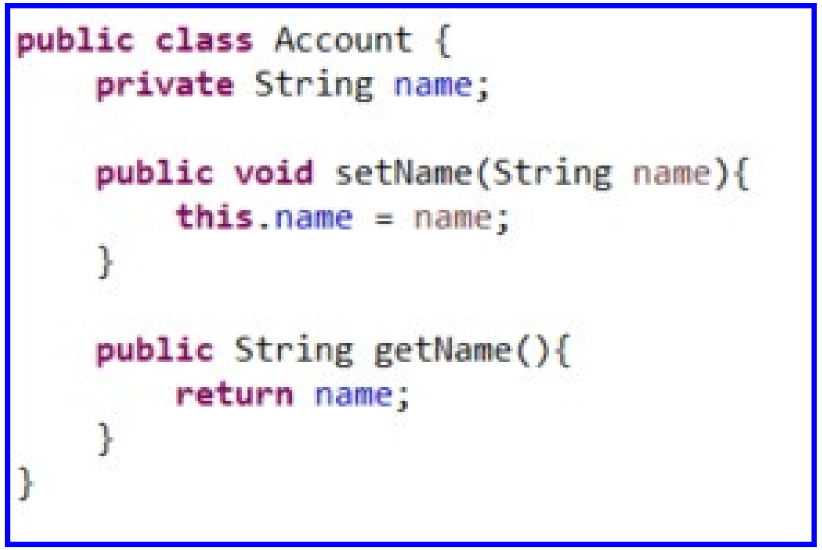
```
public class Lecture2 {
   public int add(int x, int y){
        return x+y;
      }
}
```

  - Invoke a method

```
Lecture2 lec = new Lecture2();
int sum = lec.add(3, 4);
```

Method Signature

- A method signature is the method's name and the list of its parameter types
- These methods all have the same signature:
  - `public void doIt(int x, int y, Account acc1){...};`
  - `public int doIt(int i, int j, Account acc2){...};`
  - `public Account doIt(int x, int y, Account acc3){...};`
- In other words, all of these are methods named doIt that takes two ints and an Account as parameters.
  - Return types and thrown exceptions are not considered to be a part of the method signature



```
public class Account {
    private String name;

    public void setName(String name){
        this.name = name;
    }

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

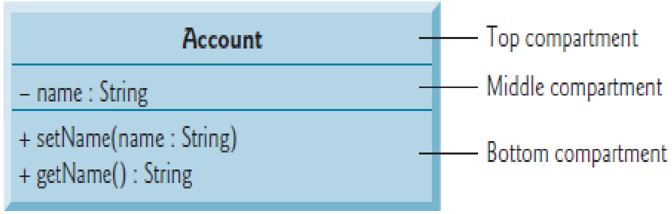
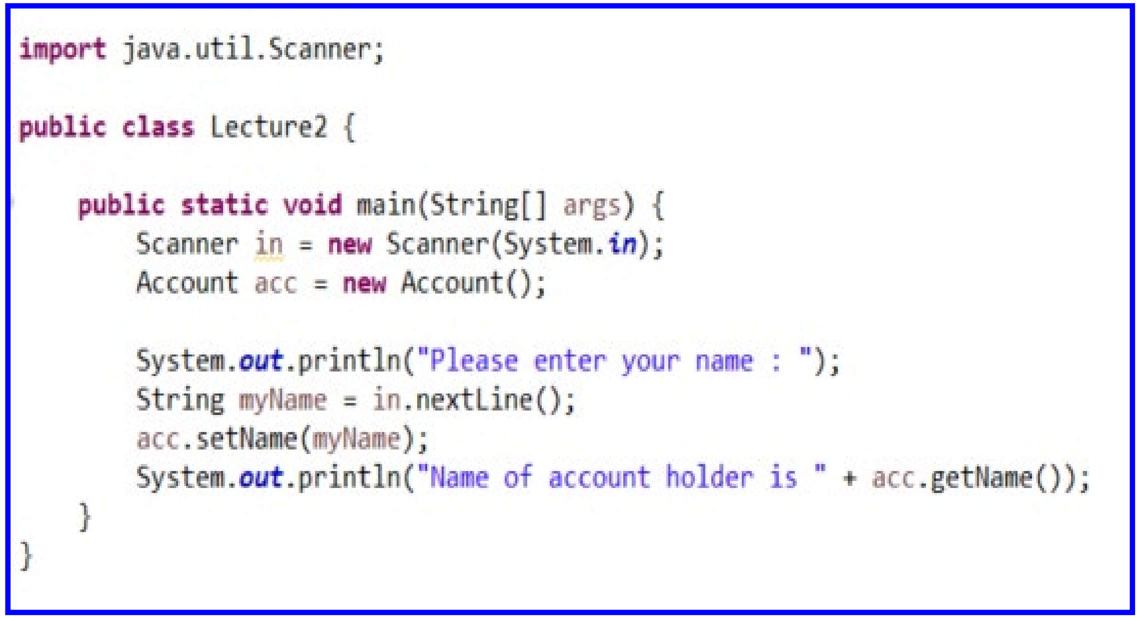


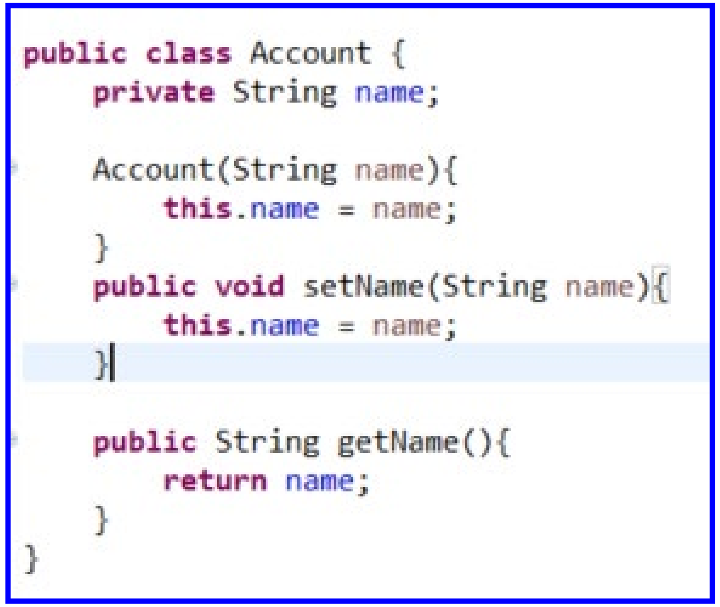
Fig. 3.3 | UML class diagram for class Account of Fig. 3.1.

Topics for discussion

- Account
  - Instance variable: name (String)
  - Setter
  - Getter
  - Why this is required in this example?
  - Return values of methods
  - Private vs public
- Driver class : Lecture2
  - Scanner for receiving user input
  - Instantiating an object – acc
  - default constructor
  - calling setters and getters
- UML
  - top compartment – class name
  - middle compartment – Attributes
  - bottom compartment – methods
  - +/- – access modifiers (- for private)



```
import java.util.Scanner;

public class Lecture2 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Account acc = new Account();

        System.out.println("Please enter your name : ");
        String myName = in.nextLine();
        acc.setName(myName);
        System.out.println("Name of account holder is " + acc.getName());
    }
}
```



```java
public class Account {
    private String name;

    Account(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }

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

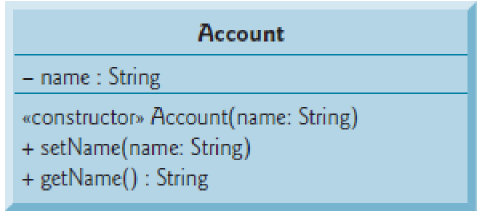


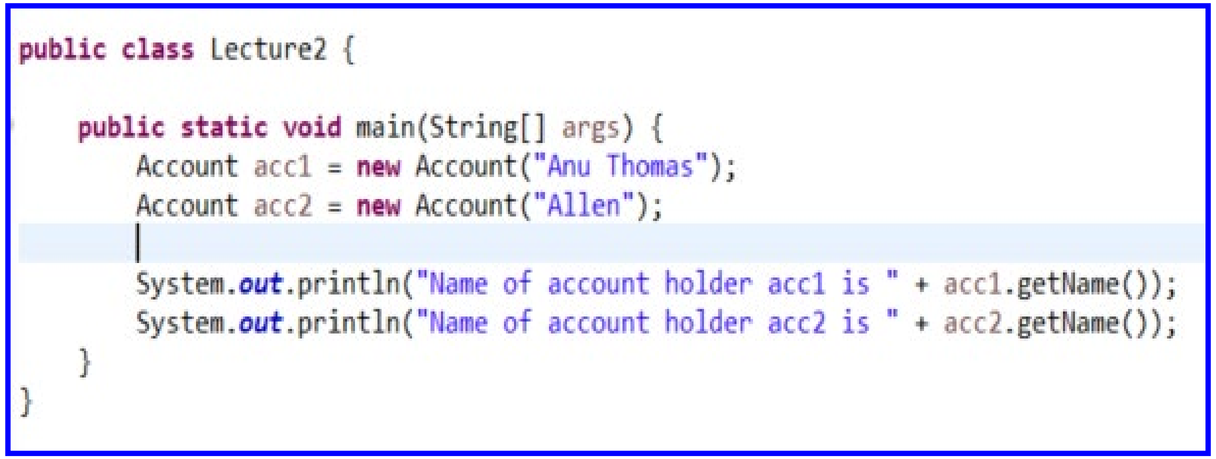
Fig. 3.7 | UML class diagram for **Account** class of Fig. 3.5.

Topics for discussion

•Account

- Constructor Syntax
- Any return value for constructors?
- parameterized constructor
- do we need a no-arg constructor?

- Driver class : Lecture2
  - How constructor gets invoked?
  - At this time, can it be possible to have a statement Account acc3 = new Account();



```java
public class Lecture2 {

    public static void main(String[] args) {
        Account acc1 = new Account("Anu Thomas");
        Account acc2 = new Account("Allen");

        System.out.println("Name of account holder acc1 is " + acc1.getName());
        System.out.println("Name of account holder acc2 is " + acc2.getName());
    }
}
```

Properties of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Process of hiding details of an object from other objects
  - So programmers can’t do unexpected things to data inside the object
  - Reusability
- Leads to abstraction & encourages modularity
- Easier maintenance

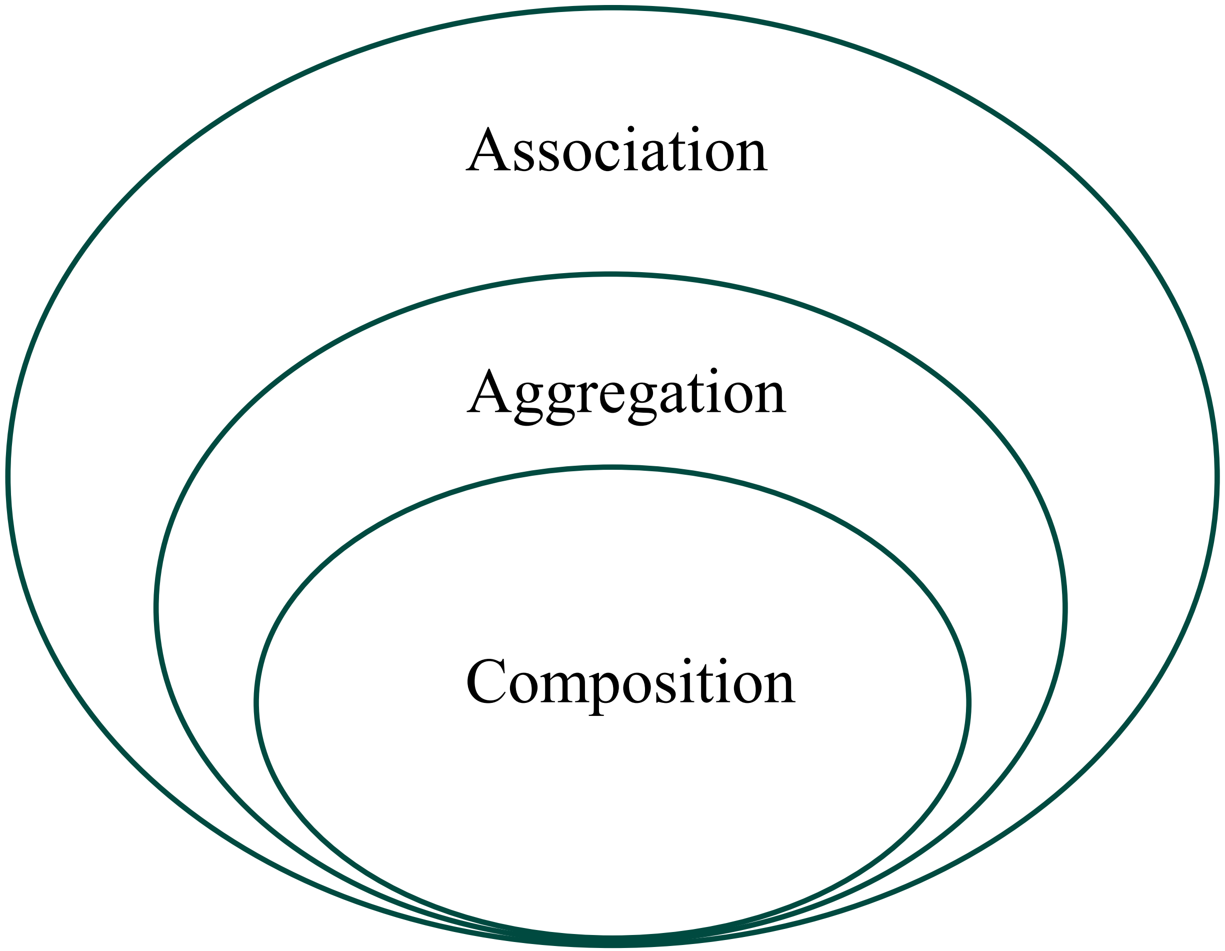
How can encapsulation help?

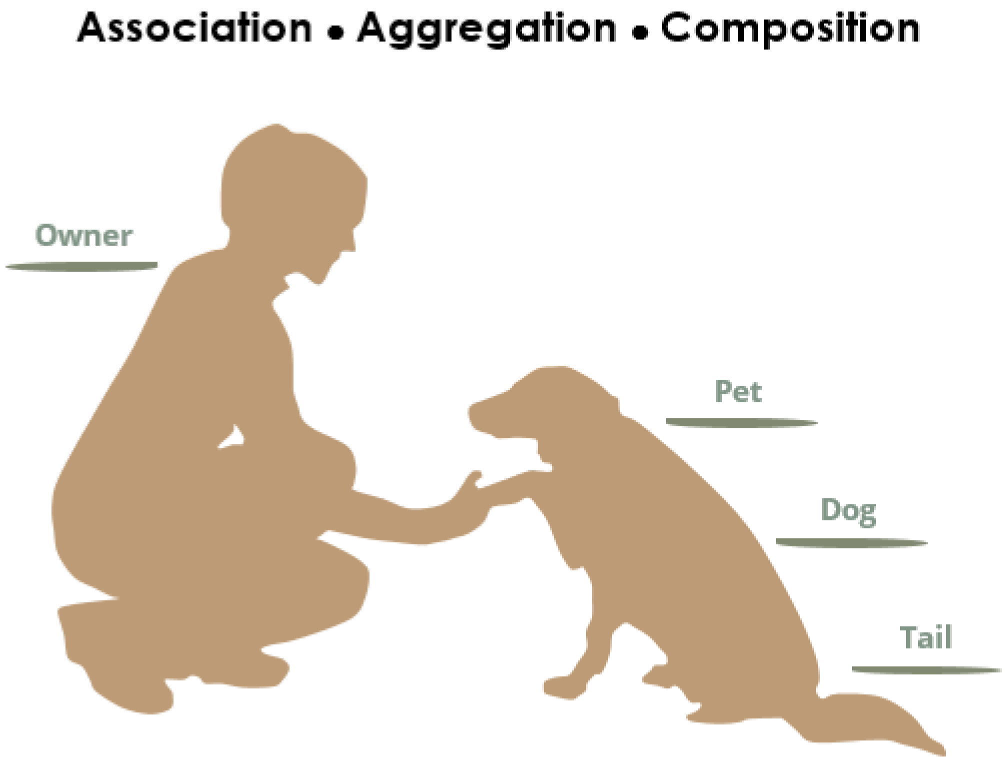
- Every object has control over member functions/data
- It will be specified which member is accessible and which is not
- Members that are only created for internal use of the object are hidden from outsiders
  - Well-intended outsiders won’t make unwanted mistakes
  - Evil-intended outsiders have more difficulty hacking the code

Abstraction

- Process of finding commonalities between different objects
  - Results in hierarchy of superclass-subclass
    - Inheritance
  - Also, defining an abstract behavior to represent common behavior of subclasses
    - Subclasses may implement the behavior in different ways
      - polymorphism

Relationship between Objects



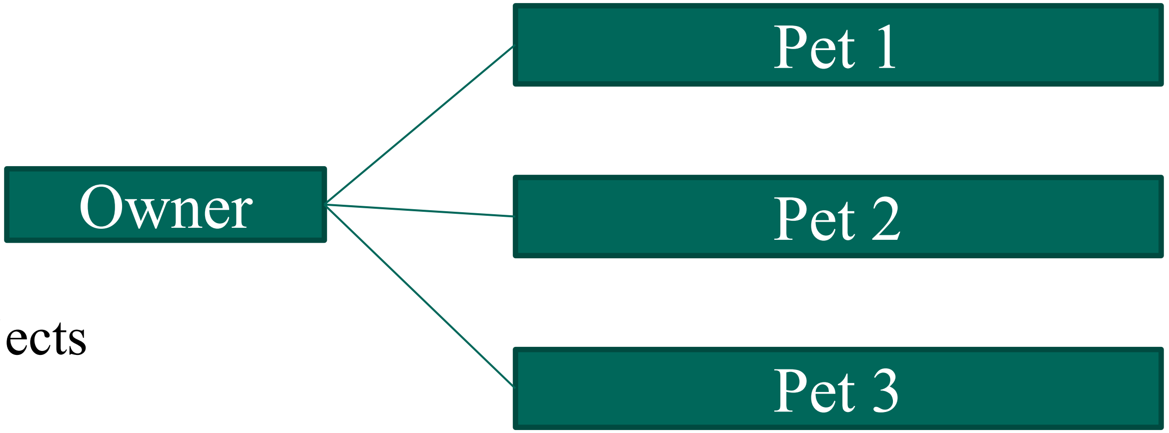


- owners feed pets, pets please owners (Association)
- a tail is a part of both dogs and cats (Aggregation / composition)
- a cat is a kind of pet (Inheritance / Generalization)

Picture taken from: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

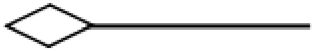
Relationships between Objects - Association

- Association (general relationship, using) ——— or ⟶
  - Weakest relationship
  - Class A uses/references Class B
  - If there is an import statement, there is at least an association between the classes
- Associations can be unidirectional or bidirectional (one-to-one, one-to-many, many-to-one, many-to-many)
- Examples:
  - Owner class & Pet class (one-to-many)
    - Owner can have more than one Pets
    - Owner and Pet are associated through their objects
  - Professor & Student (many-to-many)
    - Professor can have more than one Student
    - Student can have more than one Professor

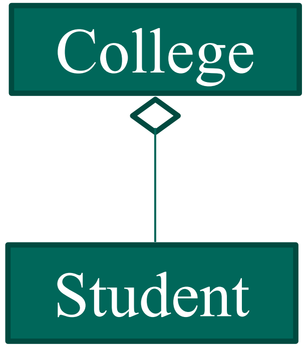


Relationships between Objects - Aggregation

- Aggregation (has-a relationship)

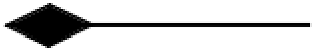


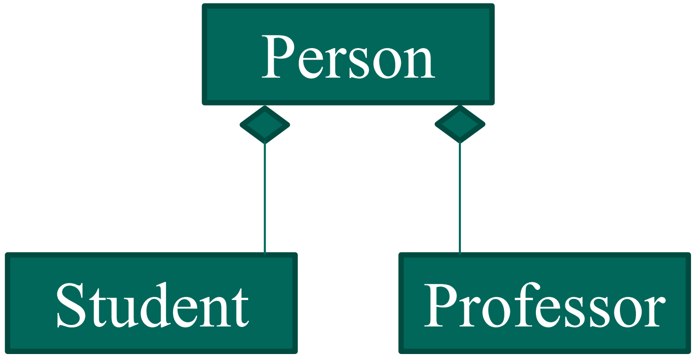
  - Special form of Association
  - Class A has Class B as a property (instance or class variable)
  - No strict ownership between these class
  - Both classes can exist separately of one another
  - Unidirectional association



- Examples:
  - Ducks in a pond example:
    - A duck has-a pond
    - Duck class has a Pond attribute, which refers to the pond the duck is currently swimming in
    - If the Duck dies, the Pond does not disappear
  - College & Student
    - College has multiple students

Relationships between Objects - Composition

- Composition (stronger has-a, consists of) 
  - Restricted form of aggregation
  - Whole/Part relationship where the Part cannot exist without the Whole
  - There is a strict ownership between the Whole and the Part.
- Examples:
  - A Vehicle has a Cabin and Trunk Space, which do not exist without the enclosing Vehicle
  - A Business is composed of Departments which do not exist without the Business
  - A Student has a Person object(with personal properties), which do not exist without the Person
  - Trees have leaves
    - A Tree has-a Leaf
    - A Tree has a Leaf array (array of many leaves actually)
    - If the Tree dies, the leaves die, as they are physically part of the tree



Relationships between Objects - Summary

- Relationship between objects (from weakest to strongest)
  1. Association – “Uses”
  2. Aggregation – “Has”
  3. Composition – “Consist of” or “part of”(stronger Has)
- Cardinality – how many objects are there?

Examples of Objects

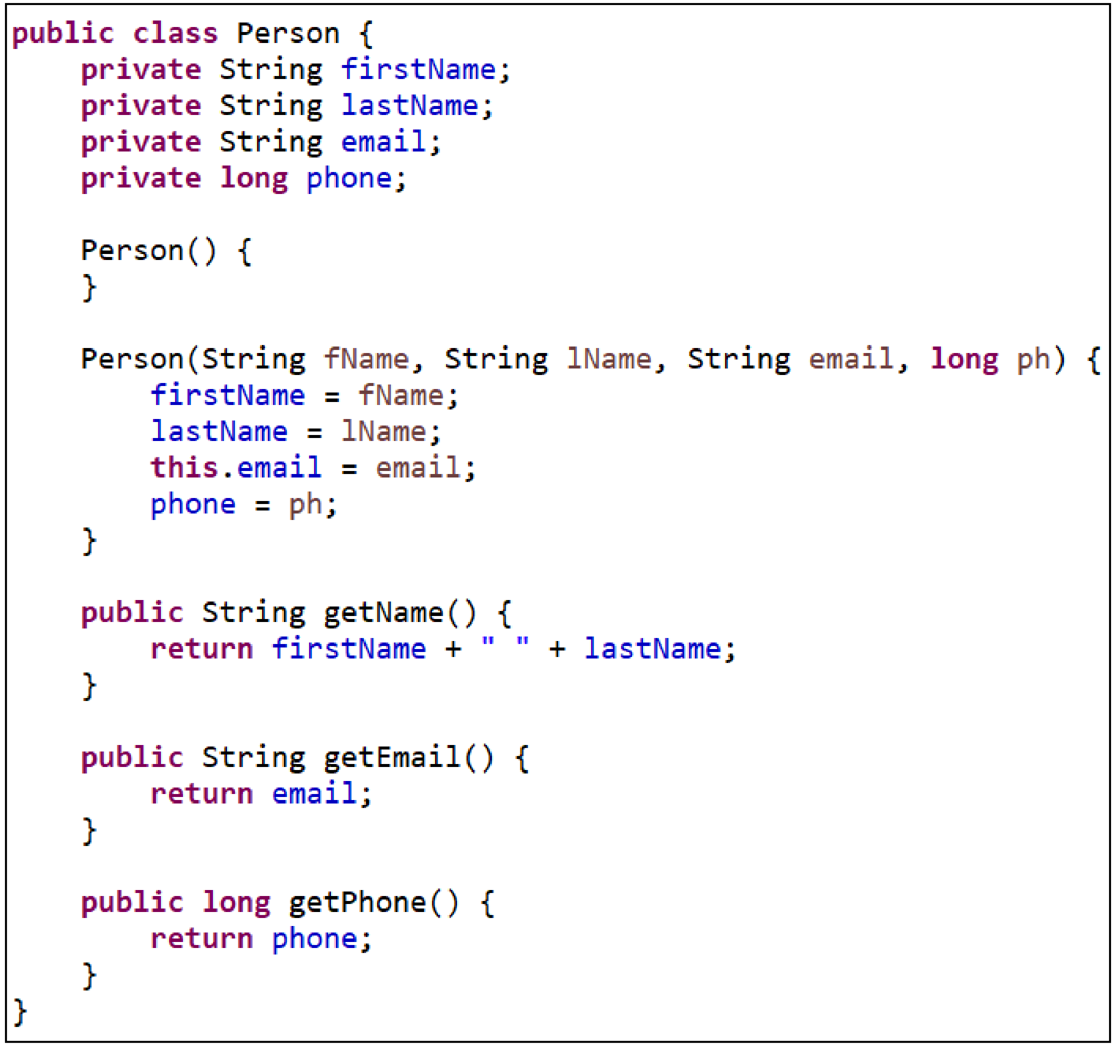
1. Patient
    1. Patient ID
    2. Name
    3. Address
    4. Telephone
2. Doctor
    1. Employee ID
    2. Name
    3. Address
    4. Telephone

What do you see here?

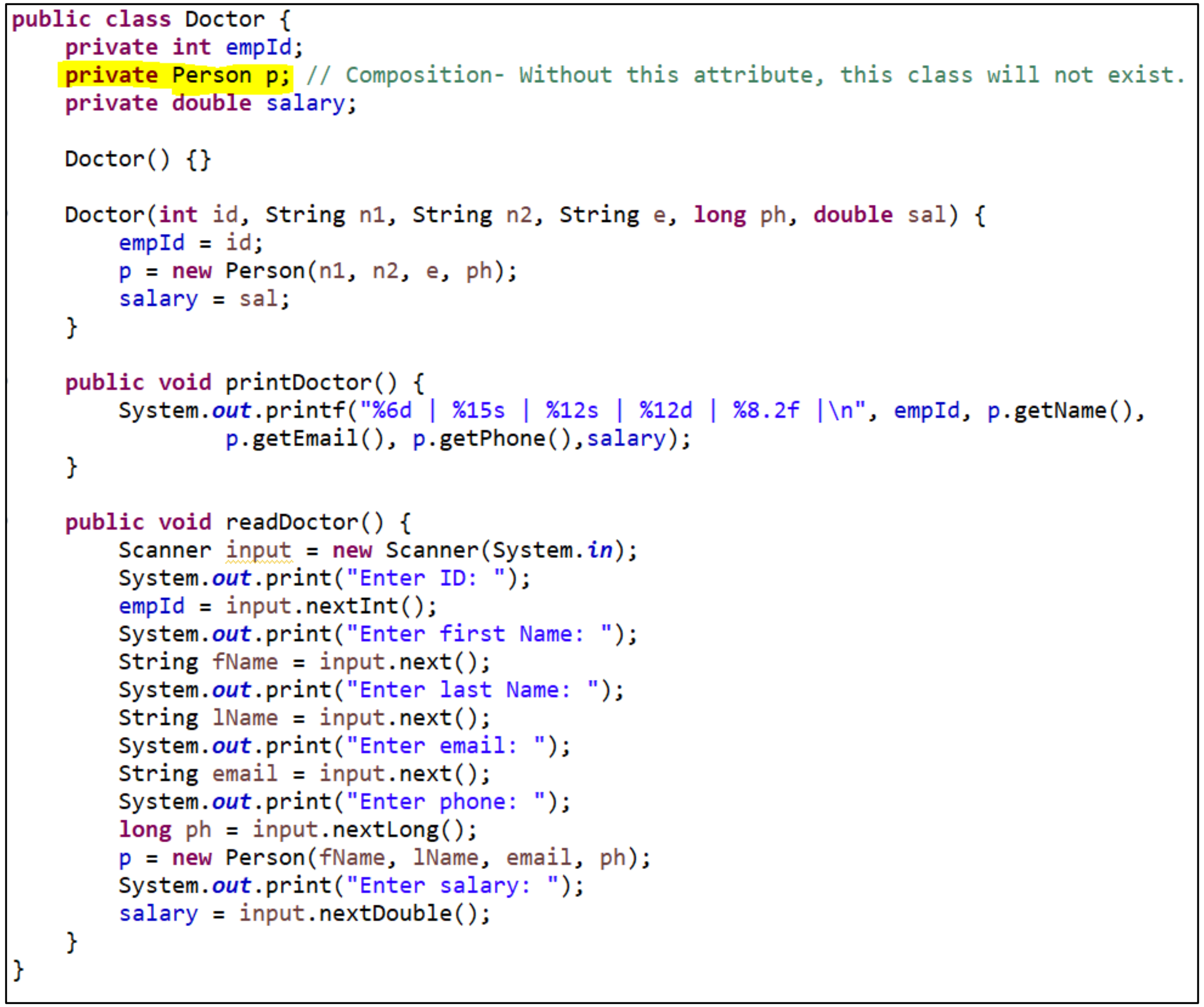
Composition - Example

- Doctor and Patient share attributes
- A new design:
  - Person
    - Name
    - Address
    - Telephone
  - Doctor and Patient classes ‘has-a’ Person object OR
  - Doctor and Patient classes consists of a Person object
  - Without a Person object, Doctor & Patient cannot exist

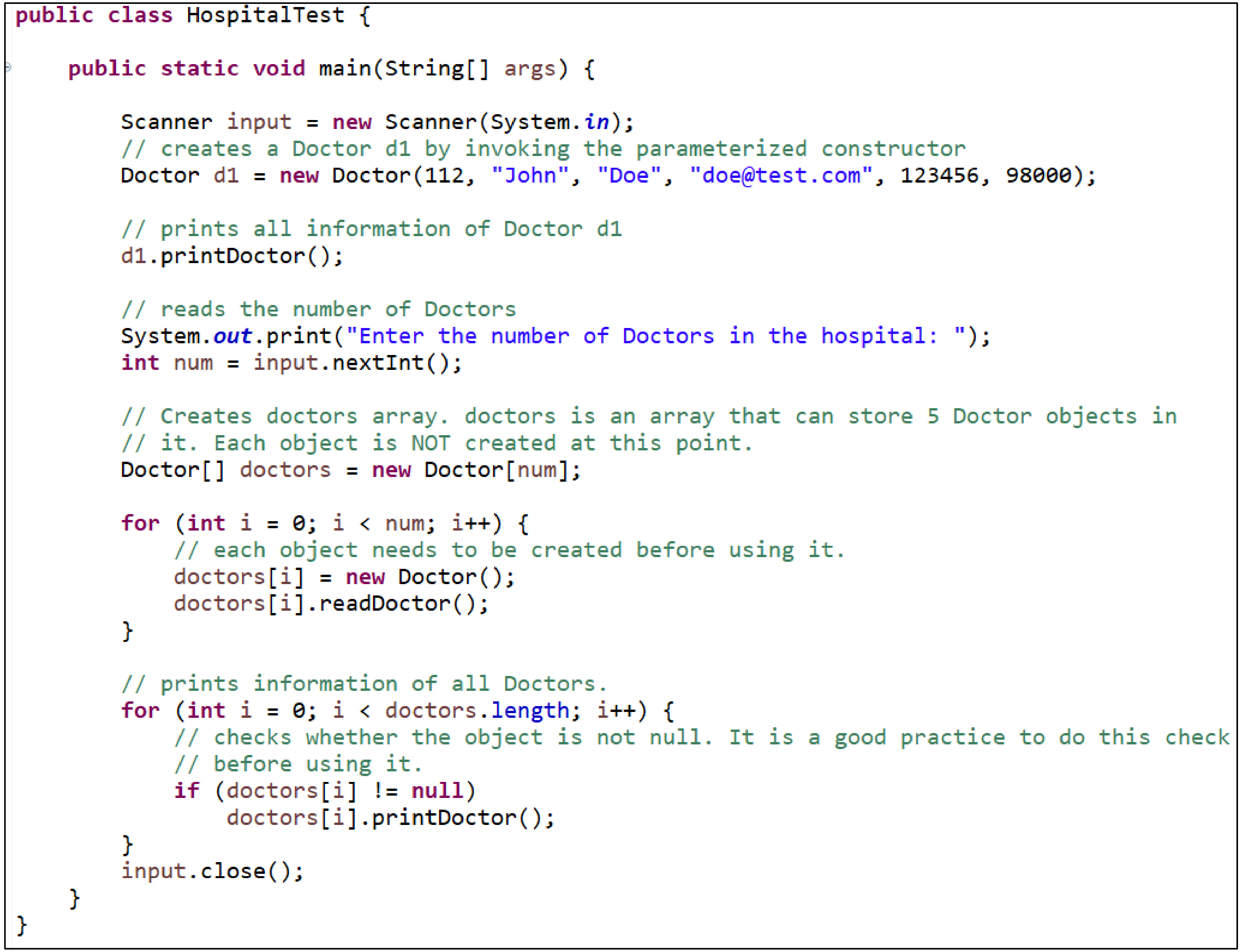
Example – Hospital System - Composition



```java
public class Person {
    private String firstName;
    private String lastName;
    private String email;
    private long phone;

    Person() {
    }

    Person(String fName, String lName, String email, long ph) {
        firstName = fName;
        lastName = lName;
        this.email = email;
        phone = ph;
    }

    public String getName() {
        return firstName + " " + lastName;
    }

    public String getEmail() {
        return email;
    }

    public long getPhone() {
        return phone;
    }
}
```



```java
public class Doctor {
    private int empId;
    private Person p; // Composition- Without this attribute, this class will not exist.
    private double salary;

    Doctor() {}

    Doctor(int id, String n1, String n2, String e, long ph, double sal) {
        empId = id;
        p = new Person(n1, n2, e, ph);
        salary = sal;
    }

    public void printDoctor() {
        System.out.printf("%6d | %15s | %12s | %12d | %8.2f |\n", empId, p.getName(),
                p.getEmail(), p.getPhone(),salary);
    }

    public void readDoctor() {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter ID: ");
        empId = input.nextInt();
        System.out.print("Enter first Name: ");
        String fName = input.next();
        System.out.print("Enter last Name: ");
        String lName = input.next();
        System.out.print("Enter email: ");
        String email = input.next();
        System.out.print("Enter phone: ");
        long ph = input.nextLong();
        p = new Person(fName, lName, email, ph);
        System.out.print("Enter salary: ");
        salary = input.nextDouble();
    }
}
```



```java
public class HospitalTest {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        // creates a Doctor d1 by invoking the parameterized constructor
        Doctor d1 = new Doctor(112, "John", "Doe", "doe@test.com", 123456, 98000);

        // prints all information of Doctor d1
        d1.printDoctor();

        // reads the number of Doctors
        System.out.print("Enter the number of Doctors in the hospital: ");
        int num = input.nextInt();

        // Creates doctors array. doctors is an array that can store 5 Doctor objects in
        // it. Each object is NOT created at this point.
        Doctor[] doctors = new Doctor[num];

        for (int i = 0; i < num; i++) {
            // each object needs to be created before using it.
            doctors[i] = new Doctor();
            doctors[i].readDoctor();
        }

        // prints information of all Doctors.
        for (int i = 0; i < doctors.length; i++) {
            // checks whether the object is not null. It is a good practice to do this check
            // before using it.
            if (doctors[i] != null)
                doctors[i].printDoctor();
        }
        input.close();
    }
}
```

Always comment your code properly. In the previous classes, comments are removed to save space in screenshots.

Lab2 – Store Management System I

- Composition
- Inheritance

Packages

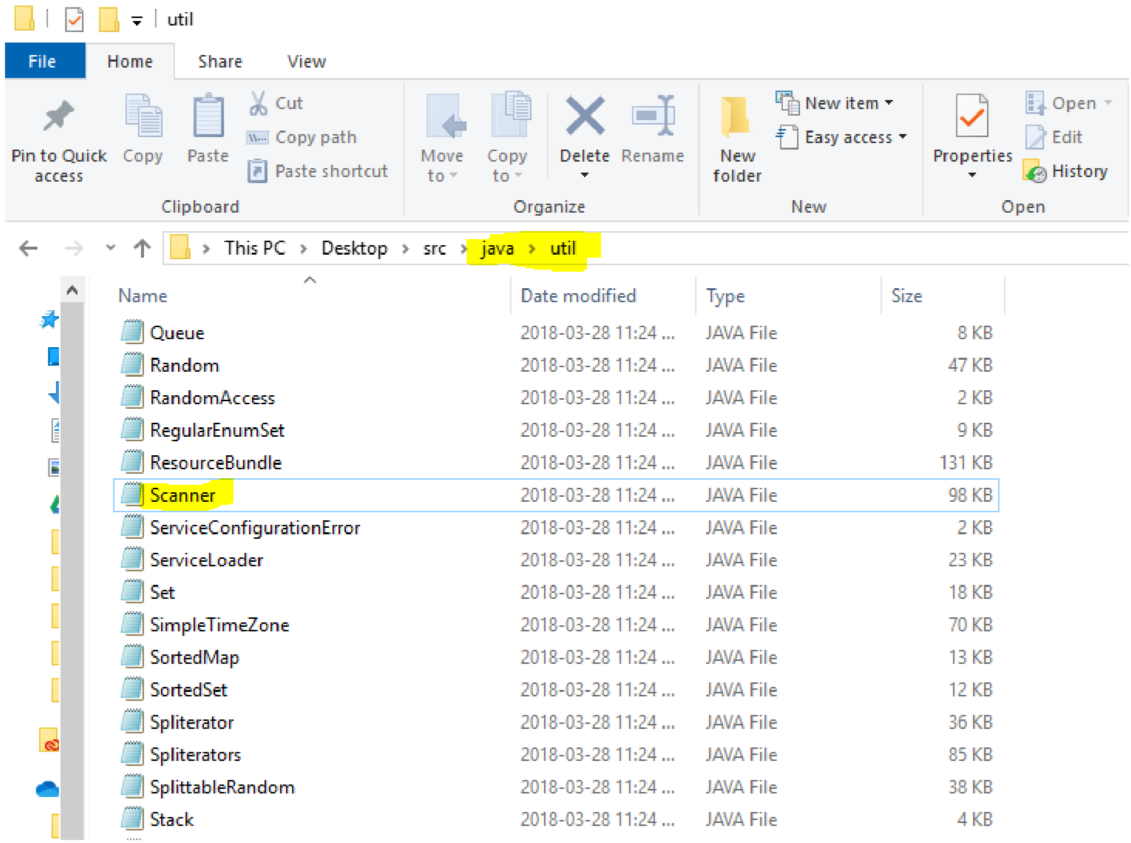
A collection of related classes

- Groups related classes together
- Namespace to avoid naming conflicts
- Provides a layer of access/protection
- Easy access
- Can also contain sub packages

Package ←→ directory (folder)
Class ←→ file

Packages and directories

```java
import java.util.Scanner;
```



This PC › Desktop › src › java › util

Name	Date modified	Type	Size
Queue	2018-03-28 11:24 ...	JAVA File	8 KB
Random	2018-03-28 11:24 ...	JAVA File	47 KB
RandomAccess	2018-03-28 11:24 ...	JAVA File	2 KB
RegularEnumSet	2018-03-28 11:24 ...	JAVA File	9 KB
ResourceBundle	2018-03-28 11:24 ...	JAVA File	131 KB
Scanner	2018-03-28 11:24 ...	JAVA File	98 KB
ServiceConfigurationError	2018-03-28 11:24 ...	JAVA File	2 KB
ServiceLoader	2018-03-28 11:24 ...	JAVA File	23 KB
Set	2018-03-28 11:24 ...	JAVA File	18 KB
SimpleTimeZone	2018-03-28 11:24 ...	JAVA File	70 KB
SortedMap	2018-03-28 11:24 ...	JAVA File	13 KB
SortedSet	2018-03-28 11:24 ...	JAVA File	12 KB
Spliterator	2018-03-28 11:24 ...	JAVA File	36 KB
Spliterators	2018-03-28 11:24 ...	JAVA File	85 KB
SplittableRandom	2018-03-28 11:24 ...	JAVA File	38 KB
Stack	2018-03-28 11:24 ...	JAVA File	4 KB

Package declaration

```
package shape;
public class Rectangle{

}
```

File Rectangle.java will be saved in the folder named shape.

Importing a package

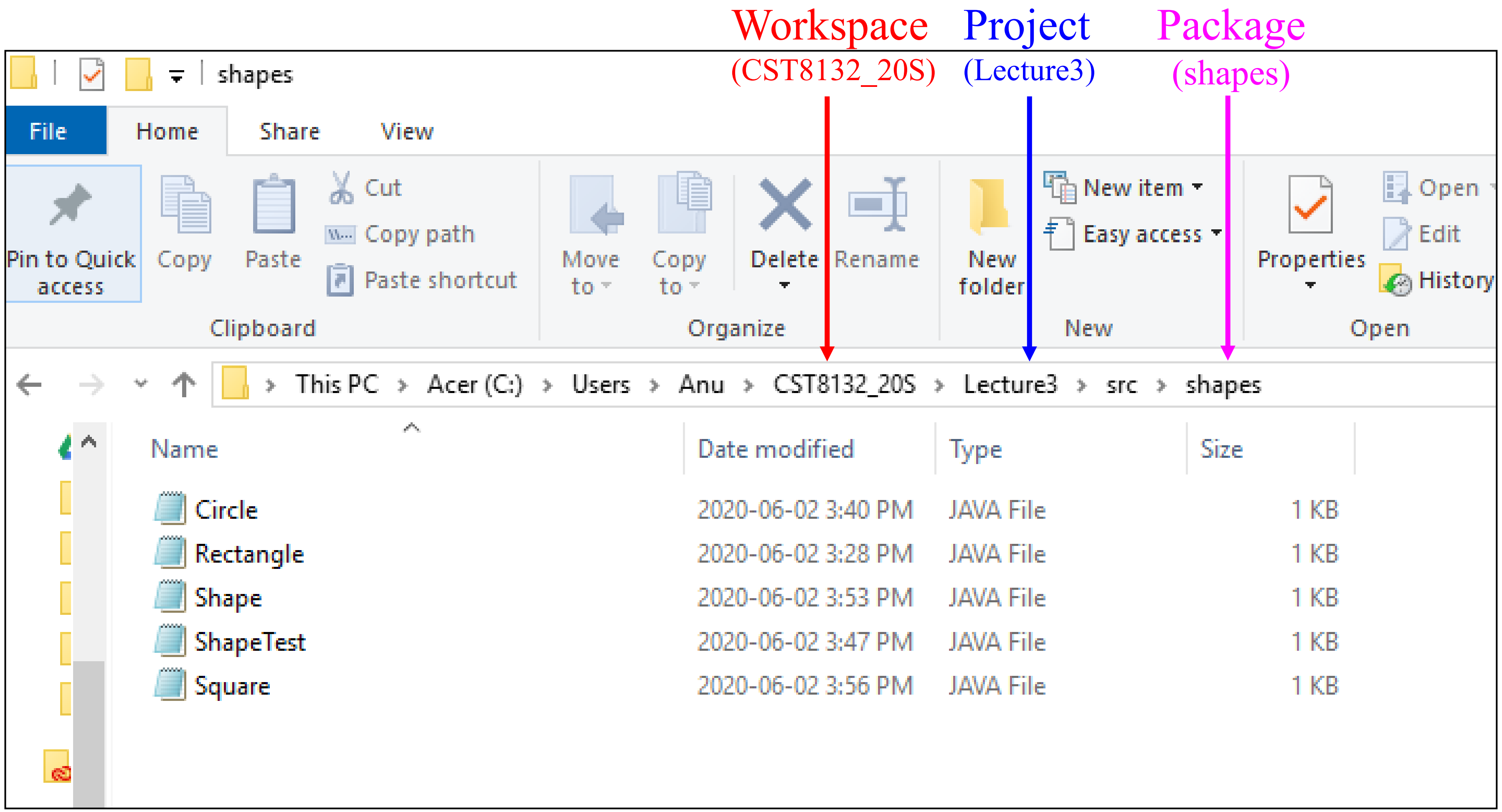
```
import packageName.*;
```

Example:

```
import java.util.*;
public class Rectangle{

}
```

Rectangle will import util package

Packages



Workspace (CST8132_20S)

Project (Lecture3)

Package (shapes)

This PC › Acer (C:) › Users › Anu › CST8132_20S › Lecture3 › src › shapes

Name	Date modified	Type	Size
Circle	2020-06-02 3:40 PM	JAVA File	1 KB
Rectangle	2020-06-02 3:28 PM	JAVA File	1 KB
Shape	2020-06-02 3:53 PM	JAVA File	1 KB
ShapeTest	2020-06-02 3:47 PM	JAVA File	1 KB
Square	2020-06-02 3:56 PM	JAVA File	1 KB

Importing a class

```
import packageName.className;
```

Example:

```
import java.util.Scanner;
public class Rectangle{

}
```

- Importing single classes has high precedence
    - If we import .*, a class with the same name in the current directory will override
    - If we import .className, it will not.

Referring to Packages

```java
java.util.Scanner input = new java.util.Scanner(System.in);
```

We can use a type from any package without importing it… just use the full name

Default package

- If we do not declare a package, files will be added to the default, unnamed package
- Classes in the default package
  - Cannot be imported
  - Cannot be used by classes in other packages
- The package java.lang is implicitly imported in all programs by default

Access Modifiers

- Private
  - Only instances of the class itself can access it
- Protected
  - Only instances of the class and instances of the subclasses can access
- Public
  - Everyone has access. can be accessed from within the class, outside the class, within the package and outside the package
- Default (Package)
  - Members of the same package can access all members