

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;
    }
}

```

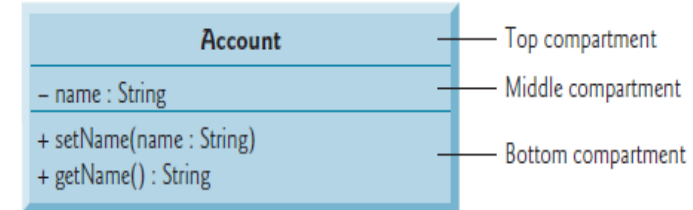


Fig. 3.3 | UML class diagram for class Account of Fig. 3.1.

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());
    }
}

```

```

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;
    }
}

```

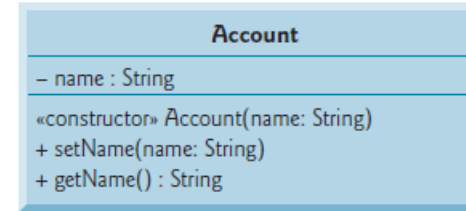


Fig. 3.7 | UML class diagram for Account class of Fig. 3.5.

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();`

```

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

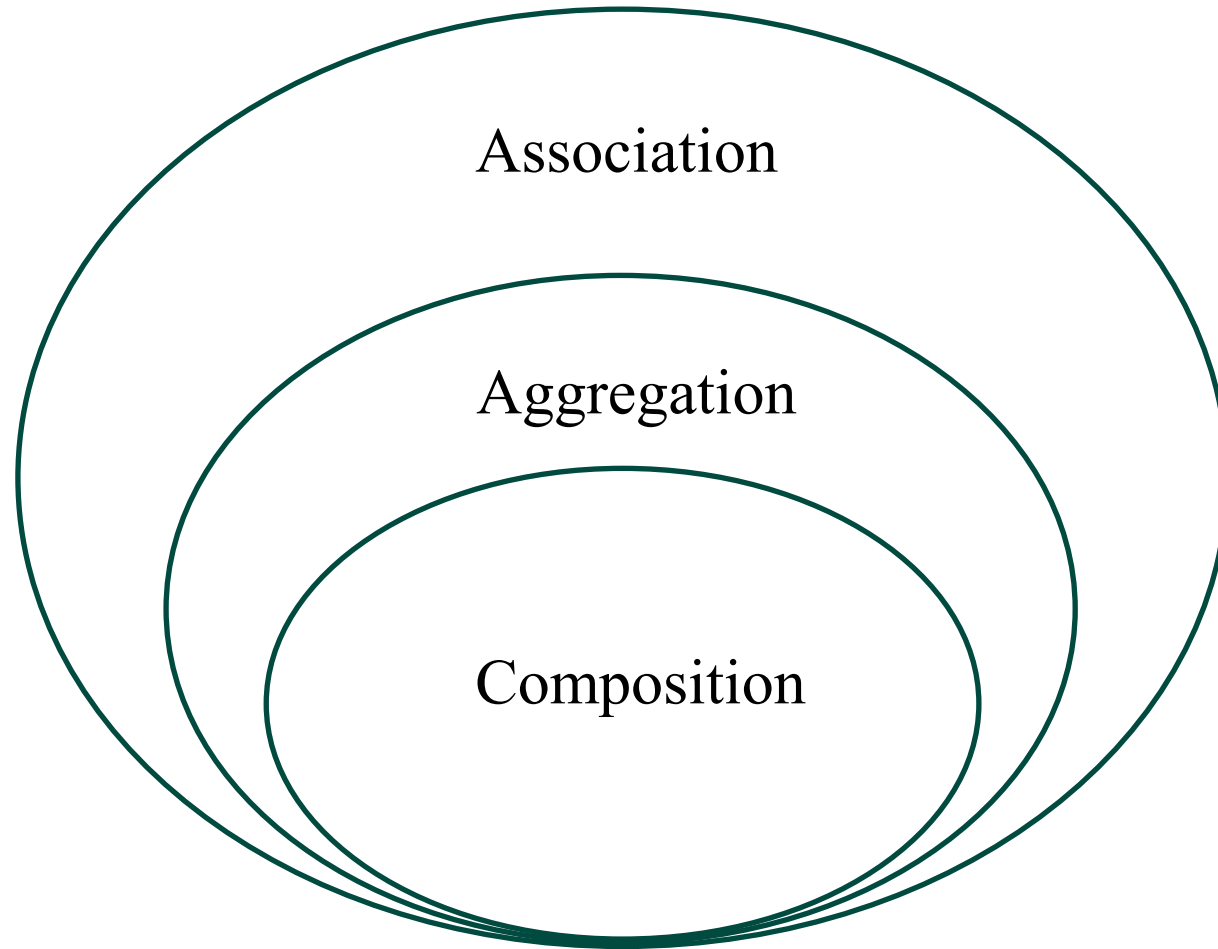


Abstraction

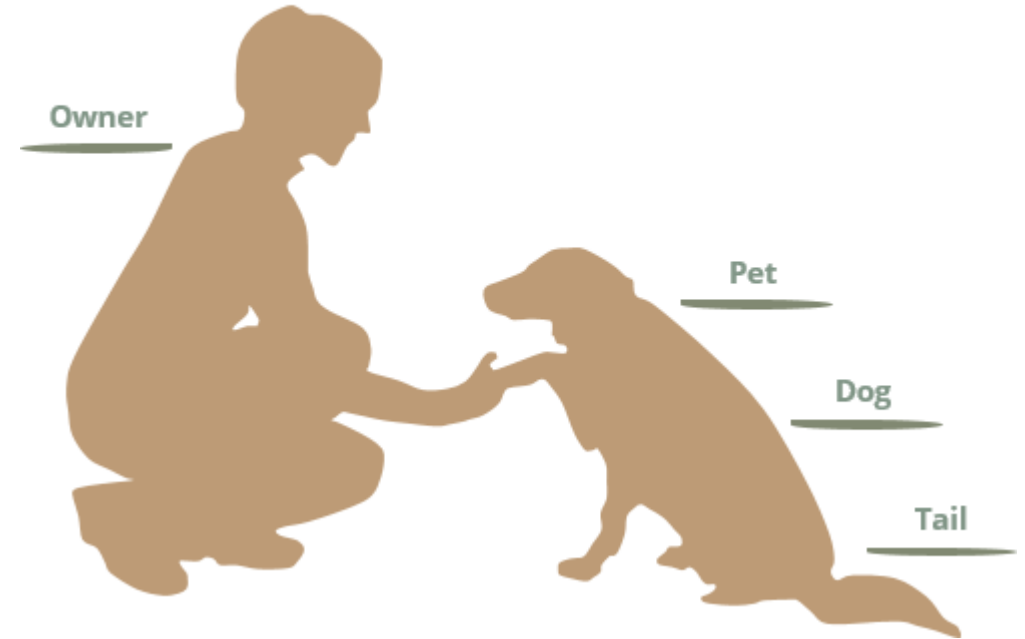
- Process of finding commonalities between different objects
 - Results in hierarchy of superclass-subclass
 - Inheritance
 - Also, defining an abstract behavior to represent common behavior of subclasses
 - Subclasses may implement the behavior in different ways
 - polymorphism



Relationship between Objects



Association • Aggregation • Composition

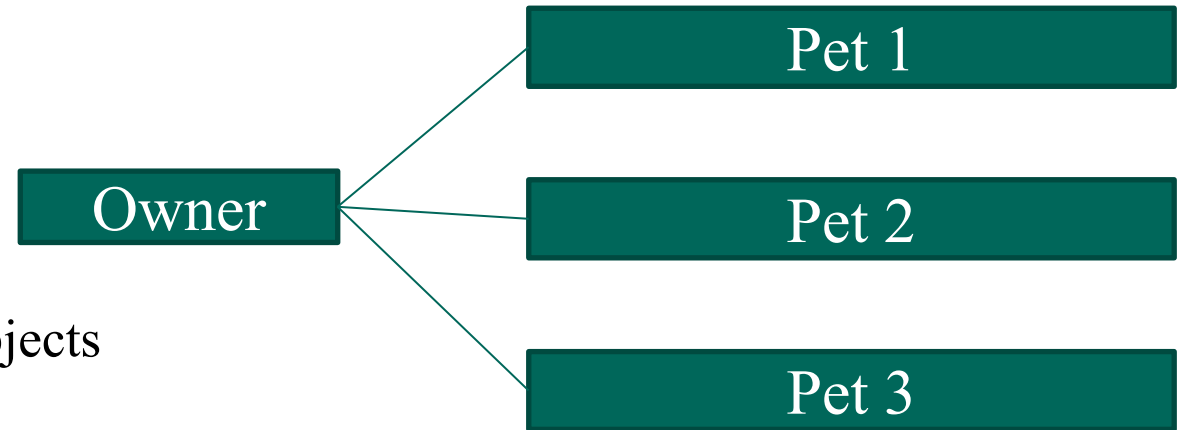


- owners feed pets, pets please owners (Association)
- a tail is a part of both dogs and cats (Aggregation / composition)
- a cat is a kind of pet (Inheritance / Generalization)

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

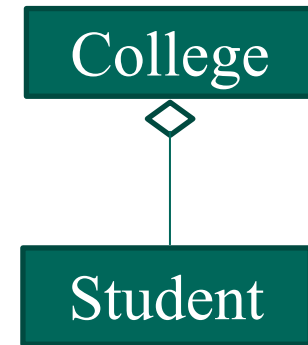


Relationships between Objects - Aggregation

- 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

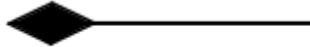


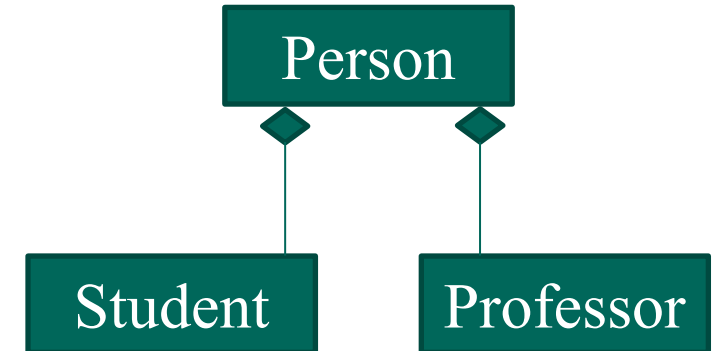
- Examples:

- 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

```
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;
    }
}
```

```
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();
    }
}
```



```

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



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

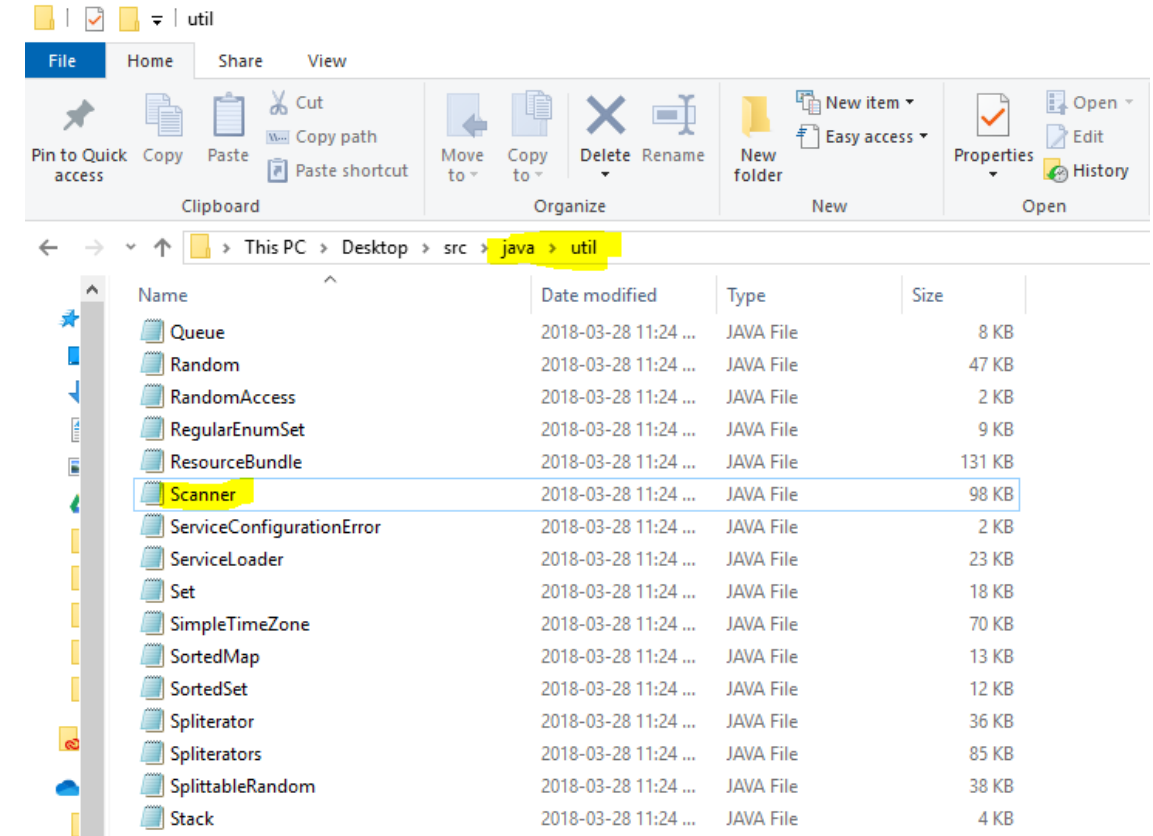
Package ↔ directory (folder)

Class ↔ file



Packages and directories

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



Package declaration

```
package shape;  
public class Rectangle{  
  
}
```

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

Importing a package

```
import packageName.*;
```

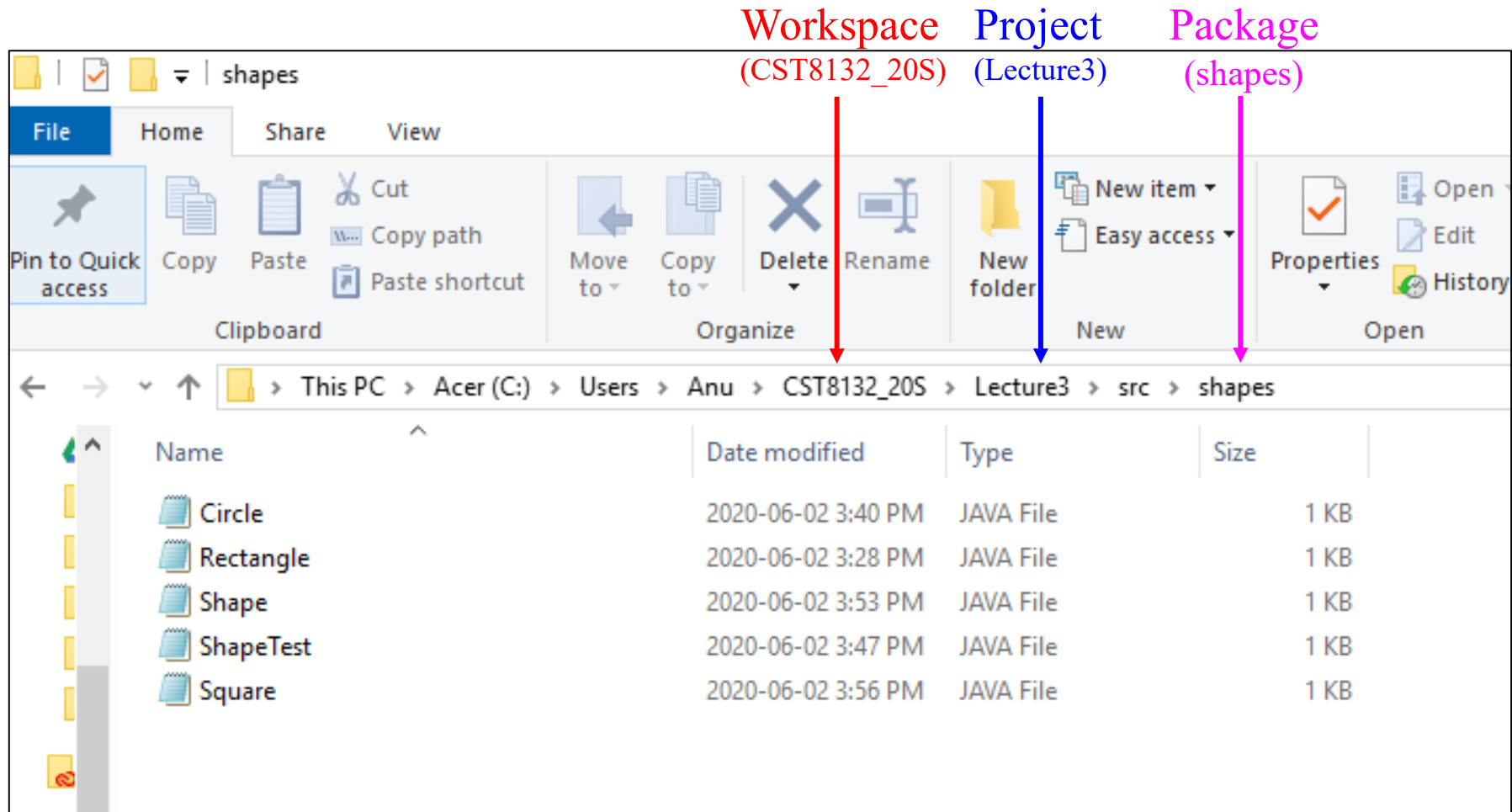
Example:

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

Rectangle will import util package



Packages



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.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



CST8132

OBJECT ORIENTED PROGRAMMING

Properties of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Professor : Dr. Anu Thomas

Email: thomasa@algonquincollege.com

Office: T314

Today's Topics

- Lab 2 – Store Management System I
- Features of OOP
 - Encapsulation
 - Polymorphism
 - Abstraction
 - Inheritance



Properties of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



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



Examples of Objects

Surgeon

1. Emp ID
2. Name
3. Address
4. Telephone
5. Base salary
6. Specialization
7. Hospital
8. Number of Surgeries

Family Doctor

1. Emp ID
2. Name
3. Address
4. Telephone
5. Base salary
6. Clinic name
7. Number of days working in the clinic

What do you see here?



Examples of Objects - Composition

Person

1. Name
2. Address
3. Telephone

Surgeon

1. Emp ID
2. **Person object**
3. Base salary
4. Specialization
5. Hospital
6. Number of Surgeries

Family Doctor

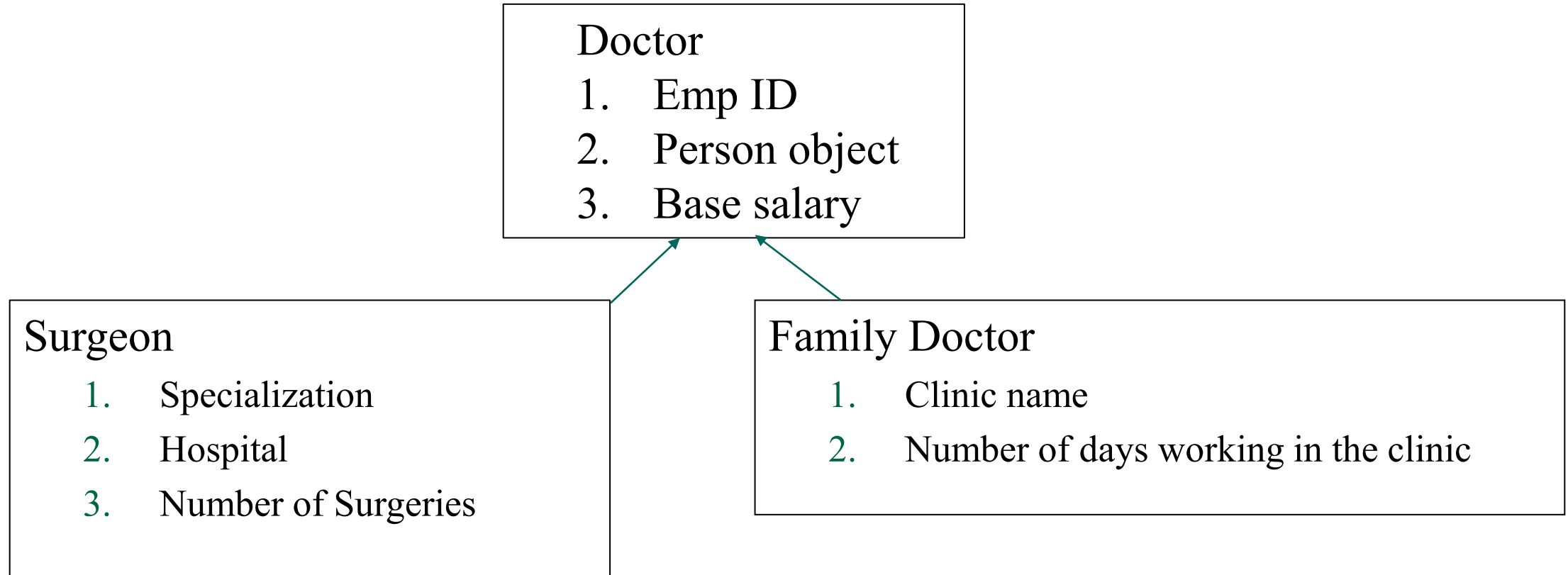
1. Emp ID
2. **Person object**
3. Base salary
4. Clinic name
5. Number of days working in the clinic

What do you see here?

Surgeon & Family Doctor “has” personal properties - Composition



Examples of Objects - Inheritance



What do you see here? **Inheritance**

Surgeon “is a” Doctor vs Surgeon has a Doctor(not true)

Family Doctor “is a” Doctor



Example

- Surgeon and Family Doctor share attributes
 - Surgeon and Family Doctor inherit from Doctor
 - Doctor is called the super-class
 - Surgeon and Family Doctor are called sub-classes



Inheritance

- When we are writing our classes, we can use an existing class as a starting point
- Suppose we have a `Doctor` class, and we want to add a `FamilyDoctor` class that is based on `Doctor`
- We would say `FamilyDoctor` “is-a” `Doctor`
- `FamilyDoctor` is a new class based on `Doctor`
- `FamilyDoctor` will inherit the attributes and methods of `Doctor`
- More `FamilyDoctor`-specific attributes and methods can be added
- Example:

```
public class FamilyDoctor extends Doctor{  
  
}
```



Inheritance (cont'd)

- With inheritance, we can create new classes from existing classes
- extends keyword: `public class B extends A`
- A is the superclass
- B is the subclass
- B inherits all of the members of A
 - The members are the fields and methods
 - The constructors of the superclass is invoked by the subclass constructors
 - Private members of A are not visible in B
 - B can add new fields and members
 - B can override methods of A
 - If a method in A named `doIt()` is overridden in B, then in B, `doIt()` invokes the B version, and `super.doIt()` invokes the A version.



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
 - Members of the same package can access all members



Access modifiers (contd.)

Access Modifier	Java keyword	UML symbol	Within class	Within package	Is subclass	Not subclass
Public	public	+	Y	Y	Y	Y
Protected	protected	#	Y	Y	Y	N
Package		~	Y	Y	N	N
Private	private	-	Y	N	N	N



Java Constructors and Inheritance

- If no constructor is defined for a class, a default constructor will be implicitly defined
- The default constructor for the super-class is automatically implicitly called by the constructor of a sub-class (unless at the beginning, super is called explicitly)
- If any non-default constructor is defined, there is no implicit default constructor which can cause compile time errors when a subclass needs the superclass to have a default constructor
- `this()` can be used in a constructor to refer to another constructor of the current class.

- Ex: a constructor `Square()` referring to `Square(int sideLength)` as in:

```
public Square() {  
    this(4);    //if the sideLength is not specified, a sideLength of 4 is used  
}
```



Inheritance and Overriding methods

- When one class extends another class, in the subclass, the programmer can override a method with the same signature in the superclass
- We use the `@Override` annotation on the method in the subclass
- The subclass version of the method will be the version used by the objects of the subclass
- If we want to use the superclass version, we can use
`super.methodName () ;`



Inheritance and Polymorphism

- Shapes Example
 - A circle “is-a” shape
 - A triangle “is-a” shape
 - We will make shape a parent class
 - Demonstration of calculating areas of different shapes
- Bicycles example of polymorphism
 - <http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>



Example – Shape, Rectangle & Square classes

```
public class Shape {  
    protected double area;  
    protected double perimeter;  
  
    public void findArea() {  
        area = 0;  
    }  
  
    public void findPerimeter() {  
        perimeter = 0;  
    }  
  
    public void printArea() {  
        System.out.println("Area is " + area);  
    }  
  
    public void printPerimeter() {  
        System.out.println("Perimeter is " + perimeter);  
    }  
}
```

```
public class Square extends Shape{  
    private double side;  
  
    Square(){}  
  
    Square(int s){  
        side = s;  
    }  
  
    @Override  
    public void findArea() {  
        area = side * side;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 4 * side;  
    }  
}
```

```
public class Rectangle extends Shape{  
    private double length;  
    private double width;  
  
    Rectangle(){  
    }  
  
    Rectangle(double l, double w){  
        length = l;  
        width = w;  
    }  
  
    @Override  
    public void findArea() {  
        area = length * width;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 2 * (length + width);  
    }  
}
```

```
public class ShapeTest {  
    public static void main(String[] args) {  
  
        //Shape object can take the form of a Shape, Rectangle or a Square  
        Shape s1 = new Shape();  
  
        Shape s2 = new Rectangle(3,4);  
  
        Shape s3 = new Square(5);  
  
        System.out.println("Shape s1 which is a Shape object");  
        s1.findArea();  
        s1.findPerimeter();  
        s1.printArea();  
        s1.printPerimeter();  
  
        System.out.println("\nShape s2 which is a Rectangle object");  
        s2.findArea();  
        s2.findPerimeter();  
        s2.printArea();  
        s2.printPerimeter();  
  
        System.out.println("\nShape s3 which is a Square object");  
        s3.findArea();  
        s3.findPerimeter();  
        s3.printArea();  
        s3.printPerimeter();  
    }  
}
```

Comments NOT
included to save
space.



Hospital System

Comments NOT
included to save
space.

```
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;
    }
}
```

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

    Doctor() {}

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

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

    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: ");
        baseSalary = input.nextDouble();
    }
}
```

```
public class Surgeon extends Doctor{

    private String specialization;
    private String hospital;
    private int numSurgeries;

    @Override
    public void readDoctor() {
        Scanner input = new Scanner(System.in);
        super.readDoctor();
        System.out.print("Enter hospital name: ");
        hospital = input.nextLine();
        System.out.print("Enter specialization: ");
        specialization = input.next();
        System.out.print("Enter number of surgeries: ");
        numSurgeries = input.nextInt();
    }

    @Override
    public void printDoctor() {
        super.printDoctor();
        System.out.printf("%12s | %12s | %8.2f |\n", hospital, specialization, baseSalary + numSurgeries*1000);
    }
}
```

```
public class FamilyDoctor extends Doctor{
    private String clinicName;
    private int numDays;

    @Override
    public void readDoctor() {
        Scanner input = new Scanner(System.in);
        super.readDoctor();
        System.out.print("Enter clinic name: ");
        clinicName = input.next();
        System.out.print("Enter number of days working in this clinic: ");
        numDays = input.nextInt();
    }

    @Override
    public void printDoctor() {
        super.printDoctor();
        System.out.printf("%12s | %8.2f |\n", clinicName, baseSalary + numDays * 4 * 12 * 100);
    }
}
```

Hospital System (Contd.)

```
public class Hospital {  
    private Doctor []doctors;  
    Hospital(){}  
    Hospital(int n){  
        doctors = new Doctor[n];  
    }  
    public void readDoctors() {  
        Scanner input = new Scanner(System.in);  
        for(int i=0; i<doctors.length; i++) {  
            System.out.print("1. Surgeon \n2. Family Doctor \nEnter Doctor's type:");  
            int type = input.nextInt();  
            if(type == 1)  
                doctors[i] = new Surgeon();  
            else if (type == 2)  
                doctors[i] = new FamilyDoctor();  
            doctors[i].readDoctor();  
        }  
    }  
    public void printDoctors() {  
        for(int i=0; i<doctors.length; i++)  
            doctors[i].printDoctor();  
    }  
}
```

```
public class HospitalTest {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.print("How many doctors do you want to add to the system: ");  
        int num = input.nextInt();  
        Hospital h = new Hospital(num);  
        h.readDoctors();  
        h.printDoctors();  
    }  
}
```



static keyword

- Discussion of `static` versus instance
 - Static variables and static methods are a part of the class itself
 - All objects of the class, if any, use the SAME copy of them
 - Static variables and methods can be used BEFORE any objects of that class are instantiated
 - Static members should be referenced by the `ClassName` itself
 - Example of using static method “sort” of `Arrays` class:
 - `Arrays.sort(myArray)`
- (reference for `Arrays` class: <https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>)
- Instance members do not exist until an object has been created (with the **new** keyword)
- Instance members are accessed through a reference to the object



In-class Exercise – College System

- Create different classes and think about their relationships
- Student (Full time student, Part time Student), Professor (Full time Professor, Part time Professor)
- Student and Professor “has” personal properties
- Full time Student “is a” Student
- Part time Student “is a” Student
- Full time Professor “is a” Professor
- Part time Professor “is a” Professor



CST8132
OBJECT ORIENTED
PROGRAMMING

Properties of OOP

- Abstract class, Interface

Professor : Dr. Anu Thomas

Email: thomasa@algonquincollege.com

Office: T314

Today's Topics

- Features of OOP
 - Abstract Classes
 - Interfaces
- Access Modifiers
- Type Casting
- Lab 3 – College System I



Abstract Class - Example

```
public abstract class Shape {  
    private String name;  
    protected double area;  
    protected double perimeter;  
  
    Shape(){}  
  
    Shape(String n){  
        name = n;  
    }  
  
    public abstract void findArea();  
    public abstract void findPerimeter();  
  
    public void printDetails() {  
        System.out.println("Name: " + name + " Area: " + area +  
            " Perimeter: " + perimeter);  
    }  
  
    public void printNum() {  
        System.out.println("from Shape");  
    }  
}
```

```
public class Circle extends Shape{  
  
    private double radius;  
  
    Circle( int r){  
        super("Circle");  
        //name = "Circle";  
        radius = r;  
    }  
  
    @Override  
    public void findArea() {  
        area = 2*Math.PI * Math.pow(radius, 2);  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 2*Math.PI * radius;  
    }  
  
    @Override  
    public void printNum() {  
        System.out.println("From Circle");  
    }  
}
```

```
public class Rectangle extends Shape{  
  
    private int length;  
    private int width;  
  
    Rectangle(){  
    }  
  
    Rectangle( int len, int wid){  
        //name = "Rectangle";  
        super("Rectangle");  
        length = len;  
        width = wid;  
    }  
  
    @Override  
    public void findArea() {  
        area = length * width;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 2*(length + width);  
    }  
}
```

```
public class Square extends Shape{  
  
    private double side;  
  
    Square(int num){  
        super("Square");  
        //name = "Square";  
        side = num;  
    }  
  
    @Override  
    public void findArea() {  
        area = side * side;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 4 * side;  
    }  
}
```

Comments NOT included to save space.



Abstract Class - Example

```
public class ShapeTest {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        Rectangle r1 = new Rectangle(5,10);  
        r1.findArea();  
        r1.findPerimeter();  
        r1.printDetails();  
  
        Circle c1 = new Circle(3);  
        c1.findArea();  
        c1.findPerimeter();  
        c1.printDetails();  
  
        // Shape s1 = new Shape();  
        // s1.findArea();  
  
        Scanner input = new Scanner(System.in);  
        System.out.println("Number of shapes you want to create: ");  
        int num = input.nextInt();  
  
        Shape []shapes = new Shape[num]; // shapes is an array that can store Shape objects in it  
  
        for(int i=0; i<shapes.length; i++) {  
            System.out.println("1. Square \n2. Rectangle \nEnter option");  
            int option = input.nextInt();  
            if(option == 1)  
                shapes[i] = new Square(3); // each object should be created. otherwise, nullPointerException  
            else if (option == 2)  
                shapes[i] = new Rectangle(4,5);  
  
            shapes[i].findArea();  
            shapes[i].findPerimeter();  
        }  
  
        for(int i=0; i<shapes.length; i++)  
            shapes[i].printDetails();  
    }  
}
```



Abstract classes and methods

- Abstract classes cannot be instantiated
 - `public abstract class Shape{...`
- Abstract classes are used to form a basis for subclasses
 - E.g. Our Shape class is an abstract class
 - Provides information about how to deal with shapes of all kinds (e.g. can calculate area of shapes)
 - The “real” shapes like triangle and circle inherit from Shape (extends Shape)
 - The subclasses must provide method bodies for the abstract methods in the superclass
- Abstract methods are used to establish a behavior without writing the code to specify the details of how it should be carried out
 - Abstract methods are meant to be overridden (`@Override`)



Interfaces

- If an abstract class has nothing other than
 - Public static final properties
 - Abstract methods

Then it could be made into an interface by changing

```
public abstract class Shape{
```

into

```
public interface Shape{
```

- This is because an interface is a list of abstract methods (and if it has any properties, then they are `public static final`)
 - In fact, as all methods in an interface are abstract, there is no need to declare them as abstract
 - Because all properties (if present) are `public static final`, there is no need to declare them as `public static final`



Interfaces

- An interface tells us what we can do with an object, and how we do it
- An interface provides us with a set of method signatures
 - Signature: method name, number and type of parameters
- An interface provides us the return values of methods
- In other words, an interface tells us how to use the object that implements the interface



Interface – Example

```
public interface Shape {  
    public static final String name = "Shape";  
  
    public void findArea();  
    public void findPerimeter();  
    public void printDetails();  
}
```

```
public class Square implements Shape{  
  
    private double side;  
    protected String name;  
    protected double area;  
    protected double perimeter;  
  
    @Override  
    public void findArea() {  
        area = side * side;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 4 * side;  
    }  
  
    @Override  
    public void printDetails() {  
        System.out.println("Name: " + name + " Area: " + area  
            + " Perimeter: " + perimeter);  
    }  
}
```

```
public class Circle implements Shape{  
  
    private double radius;  
    protected String name;  
    protected double area;  
    protected double perimeter;  
  
    Circle (){}  
  
    Circle( int r){  
        name = "Circle";  
        radius = r;  
    }  
  
    @Override  
    public void findArea() {  
        area = Math.pow(radius, 2);  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 2*Math.PI * radius;  
    }  
  
    @Override  
    public void printDetails() {  
        System.out.println("Name: " + name + " Area: " + area  
            + " Perimeter: " + perimeter);  
    }  
}
```

```
public class Rectangle implements Shape{  
  
    private int length;  
    private int width;  
    protected String name;  
    protected double area;  
    protected double perimeter;  
  
    Rectangle( int len, int wid){  
        name = "Rectangle";  
        length = len;  
        width = wid;  
    }  
  
    @Override  
    public void findArea() {  
        area = length * width;  
    }  
  
    @Override  
    public void findPerimeter() {  
        perimeter = 2*(length + width);  
    }  
  
    @Override  
    public void printDetails() {  
        System.out.println("Name: " + name + " Area: " + area  
            + " Perimeter: " + perimeter);  
    }  
}
```

Comments NOT included to save space.



Interface – Example

```
public class ShapeTest {  
  
    public static void main(String[] args) {  
  
        Rectangle r1 = new Rectangle(5,10);  
        r1.findArea();  
        r1.findPerimeter();  
        r1.printDetails();  
  
        Circle c1 = new Circle(3);  
        c1.findArea();  
        c1.findPerimeter();  
        c1.printDetails();  
  
        //Shape s1 = new Shape();  
    }  
}
```



Multiple Inheritance

- In Java, a class can extend **at most** one super class – this is contrary to the notion of multiple inheritance
- In Java, interfaces provide an alternative to multiple inheritance
- A class can implement more than one interface
- Example:

```
public class Rock implements Hammer, PaperWeight, Weapon {  
  
}
```

- `Public class Employee extends Person, Hr{... } // will not work`
- Multilevel – `Employee extends Person, Regular extends Employee // possible`



static keyword

- Discussion of `static` versus instance
 - Static variables and static methods are a part of the class itself
 - All objects of the class, if any, use the SAME copy of them
 - Static variables and methods can be used BEFORE any objects of that class are instantiated
 - Static members should be referenced by the `ClassName` itself
 - Example of using static method “sort” of `Arrays` class:
 - `Arrays.sort(myArray)`
- (reference for `Arrays` class: <https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>)
- Instance members do not exist until an object has been created (with the **new** keyword)
- Instance members are accessed through a reference to the object



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
 - Members of the same package can access all members



Access modifiers (contd.)

Access Modifier	Java keyword	UML symbol	Within class	Within package	Is subclass	Not subclass
Public	public	+	Y	Y	Y	Y
Protected	protected	#	Y	Y	Y	N
Package		~	Y	Y	N	N
Private	private	-	Y	N	N	N



Lab 3 – College System I



CST8132
OBJECT ORIENTED
PROGRAMMING

ArrayList
UML 1

Professor : Dr. Anu Thomas
Email: thomasa@algonquincollege.com
Office: T314

What we learned so far?

- Arrays – 1-dim & multi-dim
- Composition
- Inheritance
- Polymorphism
- Abstraction
- Abstract class
- Interface
- Static
- This
- super

Overriding

Overloading

Access specifiers

Constructors – default, no-arg, parameterized
toString



ArrayList

- **ArrayList** is a resizable, ordered collection of elements.
- Internally, ArrayList implements a dynamically allocated array
- It is expressed as:
 - `ArrayList<T>;` meaning an ArrayList of type T



ArrayList

	array	ArrayList
Size after creation	Fixed	Dynamic
Elements	Primitive or objects	objects
Memory	Values or references are stored in contiguous locations	References are stored in contiguous locations
Get/Set	Assignment using []	Method calls
Added functionality	<ul style="list-style-type: none">• length	<ul style="list-style-type: none">• Add to end• Insert at position• Clear all elements• Remove element by position or value• Find element• ...



Method	Description
<code>add</code>	Adds an element to the end of the <code>ArrayList</code> .
<code>clear</code>	Removes all the elements from the <code>ArrayList</code> .
<code>contains</code>	Returns <code>true</code> if the <code>ArrayList</code> contains the specified element; otherwise, returns <code>false</code> .
<code>get</code>	Returns the element at the specified index.
<code>indexOf</code>	Returns the index of the first occurrence of the specified element in the <code>ArrayList</code> .
<code>remove</code>	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
<code>size</code>	Returns the number of elements stored in the <code>ArrayList</code> .
<code>trimToSize</code>	Trims the capacity of the <code>ArrayList</code> to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

ArrayList (Contd.)

- An `ArrayList`'s capacity indicates how many items it can hold without growing.
- When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
 - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
 - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

ArrayList (Contd.)

- Method **add** adds elements to the `ArrayList`.
 - One-argument version appends its argument to the end of the `ArrayList`.
 - Two-argument version inserts a new element at the specified position.
 - Collection indices start at zero.
- Method **size** returns the number of elements in the `ArrayList`.
- Method **get** obtains the element at a specified index.
- Method **remove** deletes an element with a specific value.
 - An overloaded version of the method removes the element at the specified index.
- Method **contains** determines if an item is in the `ArrayList`.

ArrayList (Create and Add)

- Array

```
String studentNames[] = new String[100];  
int currentPosition = 0;  
studentNames[currentPosition++] = "Peter";  
studentNames[currentPosition++] = "Pauline";  
studentNames[currentPosition++] = "Robin";
```

- ArrayList

```
List<String> studentNames = new ArrayList<>();  
studentNames.add("Peter");  
studentNames.add("Pauline");  
studentNames.add("Robin");
```



ArrayList (Print names)

- Array

```
for(int i=0; i<=currentPosition; i++)  
    System.out.println(studentNames[i]);
```

- ArrayList

```
for(String s : studentNames)  
    System.out.println(s);
```

- `System.out.println(studentNames)`



ArrayList (Insert at first position)

- Array

```
for(int i=currentPosition; i>0; i--)  
    studentNames[i]=studentNames[i-1];  
studentNames[0]="John";
```

- ArrayList

```
studentNames.add(0, "John");
```



Type-Wrapper Classes

- Each primitive type has a corresponding **type-wrapper class** (in package `java.lang`).
 - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` and `Short`.
- Each type-wrapper class enables you to manipulate primitive-type values as objects.
- Collections cannot manipulate variables of primitive types.
 - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.
- Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`.
- The type-wrapper classes are `final` classes, so you cannot extend them.
- Primitive types do not have methods, so the methods related to a primitive type are in
- the corresponding type-wrapper class



Autoboxing and Auto-Unboxing

- A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions are performed automatically—called **autoboxing** and **auto-unboxing**.
- Example:

```
▪ // create integerArray
  Integer[] integerArray = new Integer[5];

  // assign Integer 10 to integerArray[ 0 ] integerArray[0] = 10;

  // get int value of Integer
  int value = integerArray[0];
```

Example – Hospital System

```
import java.util.Scanner;

public class HospitalTest {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.print("How many doctors do you want to add to the system: ");
        int num = input.nextInt();

        Hospital h = new Hospital(num);

        h.readDoctors();

        System.out.println("\n\nSummary of Doctors");
        System.out.println("*****");
        h.printDoctors();

    }
}
```

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

public class Hospital {

    //private Doctor []doctors;
    private int numDoctors;
    private ArrayList <Doctor> doctors;

    Hospital(){}

    Hospital(int n){
        //doctors = new Doctor[n];
        numDoctors = n;
        doctors = new ArrayList<Doctor>(numDoctors);
    }

    public void readDoctors() {
        Scanner input = new Scanner(System.in);
        //for(int i=0; i<doctors.length; i++) {
        for(int i=0; i<numDoctors; i++) {
            System.out.print("1. Surgeon \n2. Family Doctor \nEnter Doctor's type:");
            int type = input.nextInt();
            if(type == 1)
                //doctors[i] = new Surgeon();
                doctors.add(new Surgeon());
            else if (type == 2)
                //doctors[i] = new FamilyDoctor();
                doctors.add(new FamilyDoctor());

            //doctors[i].readDoctor();
            doctors.get(i).readDoctor();

        }
    }

    public void printDoctors() {
        //for(int i=0; i<doctors.length; i++)
        // doctors[i].printDoctor();
        for(int i=0; i< doctors.size(); i++)
            doctors.get(i).printDoctor();
    }
}
```



Final modifier

- With variable – becomes constant
- With method – cannot be overridden in subclasses
- With class – cannot be sub-classed

Eclipse demo



UML – Unified Modeling Language

- General-purpose developmental modeling language
- Standard way to visualize the design of a system
- References:
 - <https://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf>
 - https://en.wikipedia.org/wiki/Unified_Modeling_Language
 - <https://creately.com/blog/diagrams/class-diagram-tutorial/>

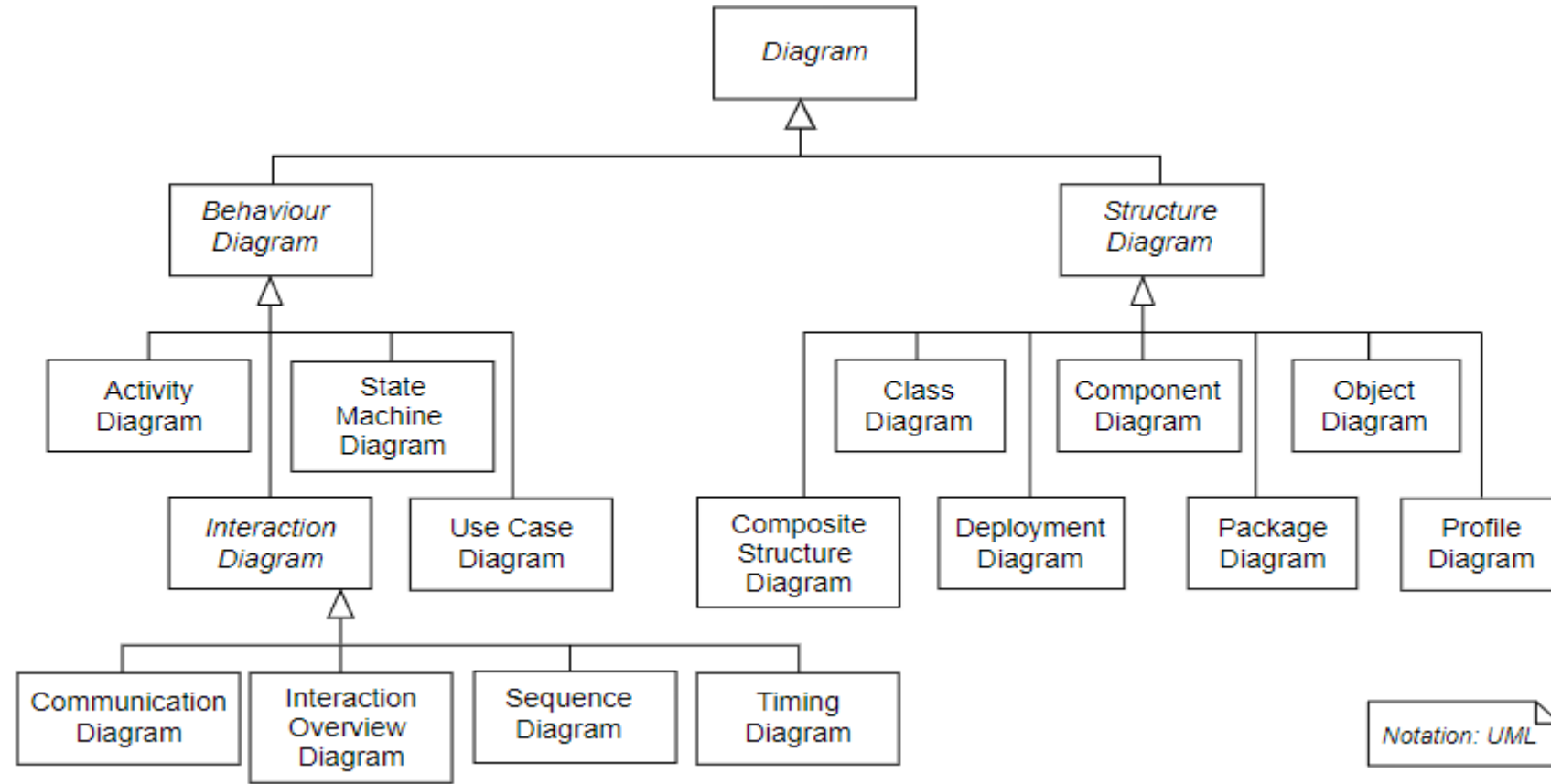


Uses of UML

- As a sketch: to communicate aspects of the system
 - Forward design: create UML before coding
 - Backward design: create UML after coding (for documentation)
 - Often done on whiteboard or on paper
 - Used in brainstorming
- As a blueprint: a complete design to be implemented
 - Done with professional tools like Visio
- As a programming language: tools available to auto-generate the structure of the code from the UML



UML diagrams



Structural : Class Diagram

- Top compartment – class name – Centered and **Bold**
- Middle compartment – State (attributes, properties, instance or class variables)
- Bottom compartment – Behaviors (methods)
- Access Level Modifiers:
 - + means public
 - # means protected
 - ~ means package protected
 - - means private
- Notice that in UML, types come after the member name, rather than before in Java

In UML:

-myInt: int
+factorial(n: int): int

In Java:

private int myInt and
public int factorial (int n)

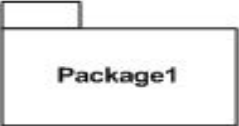
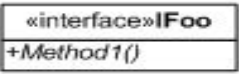
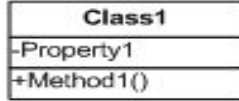


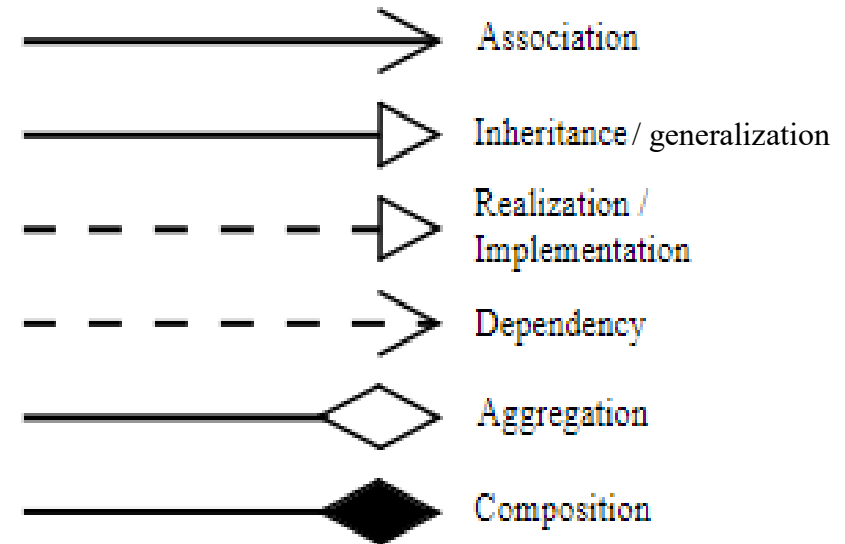
UML Relationships

- Lines drawn between class boxes indicate relationships between objects of the classes
 - Multiplicity: numbers at either end of an association line represent how many of each are involved
 - 0..1 – no instances or one instance - optional
 - 1 or 1..1 – one and only one
 - 1..n – one to a specific limit
 - 1..* - one or more
 - 0..* - zero or more
 - * - zero or more
 - 0..n – zero to a specific limit



UML Symbols

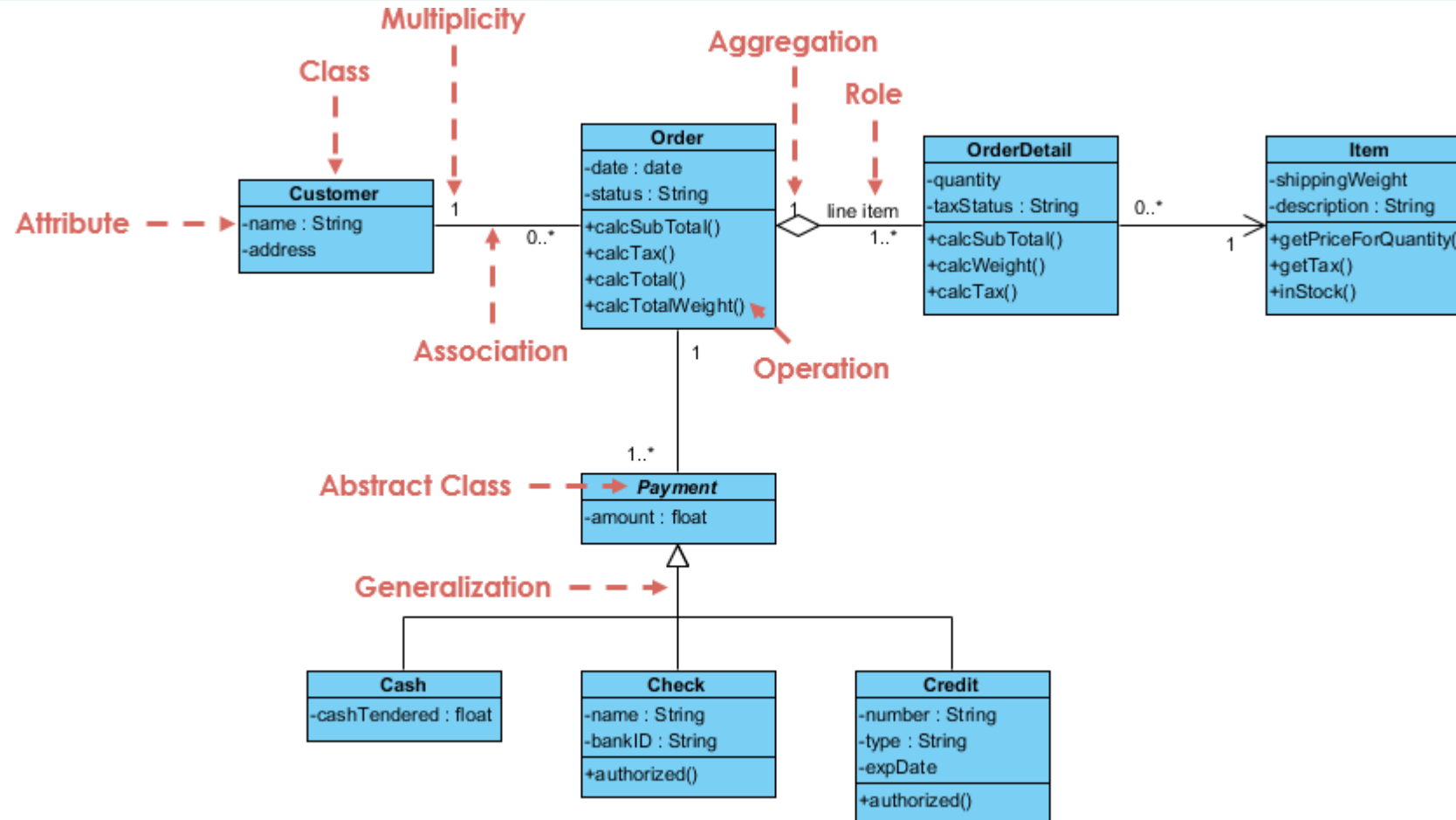
Shape	Description
 Package1	Package A collection of interfaces and classes.
 «interface»I Foo +Method1()	Interface Microsoft guidelines specify that interfaces should start with I. This graphic can also sometimes be used as an abstract class.
 Class1 -Property1 +Method1()	Class Properties or attributes sit at the top, methods or operations at the bottom. + indicates public and # indicates protected.



Research more!

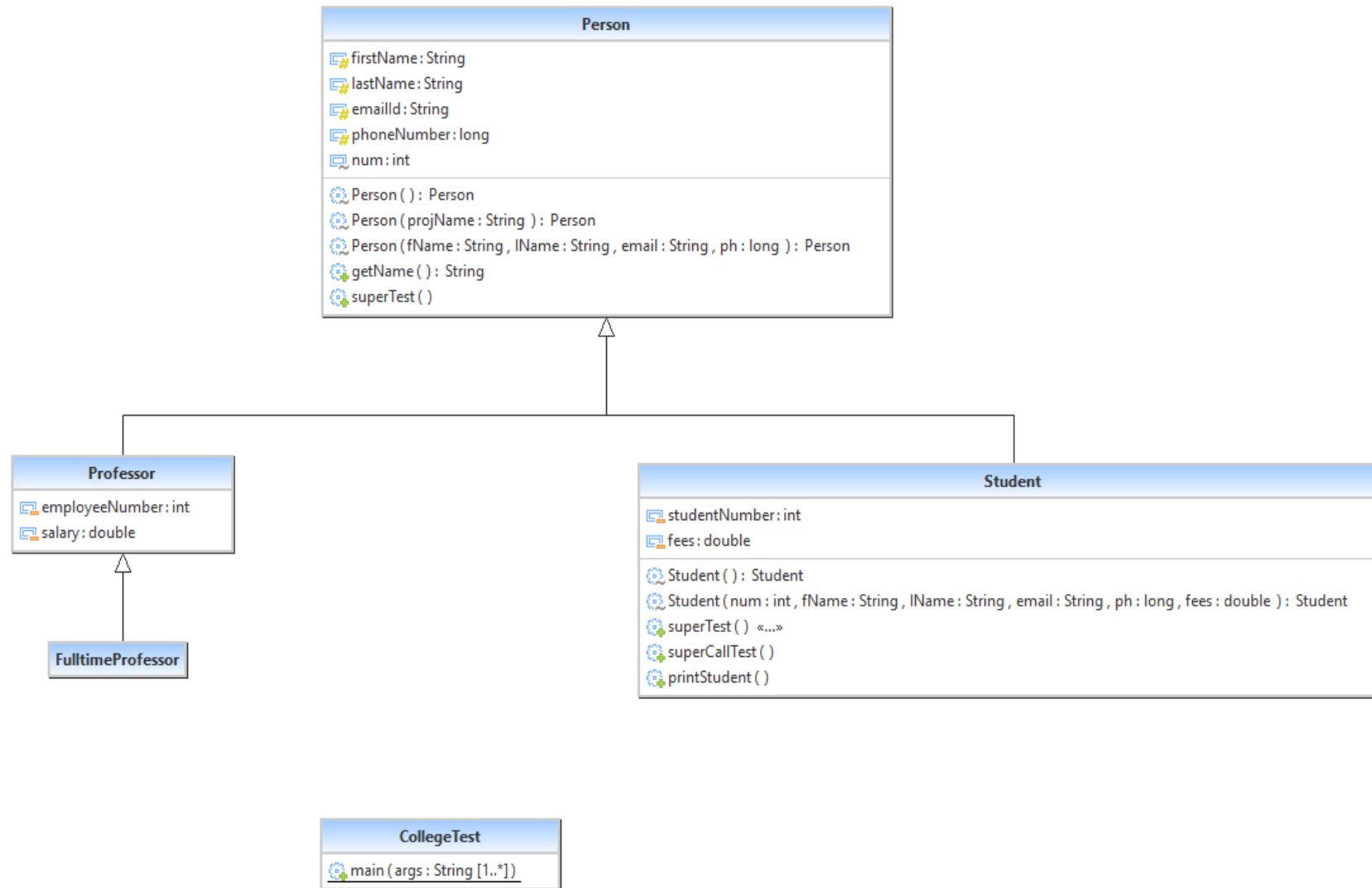


Example

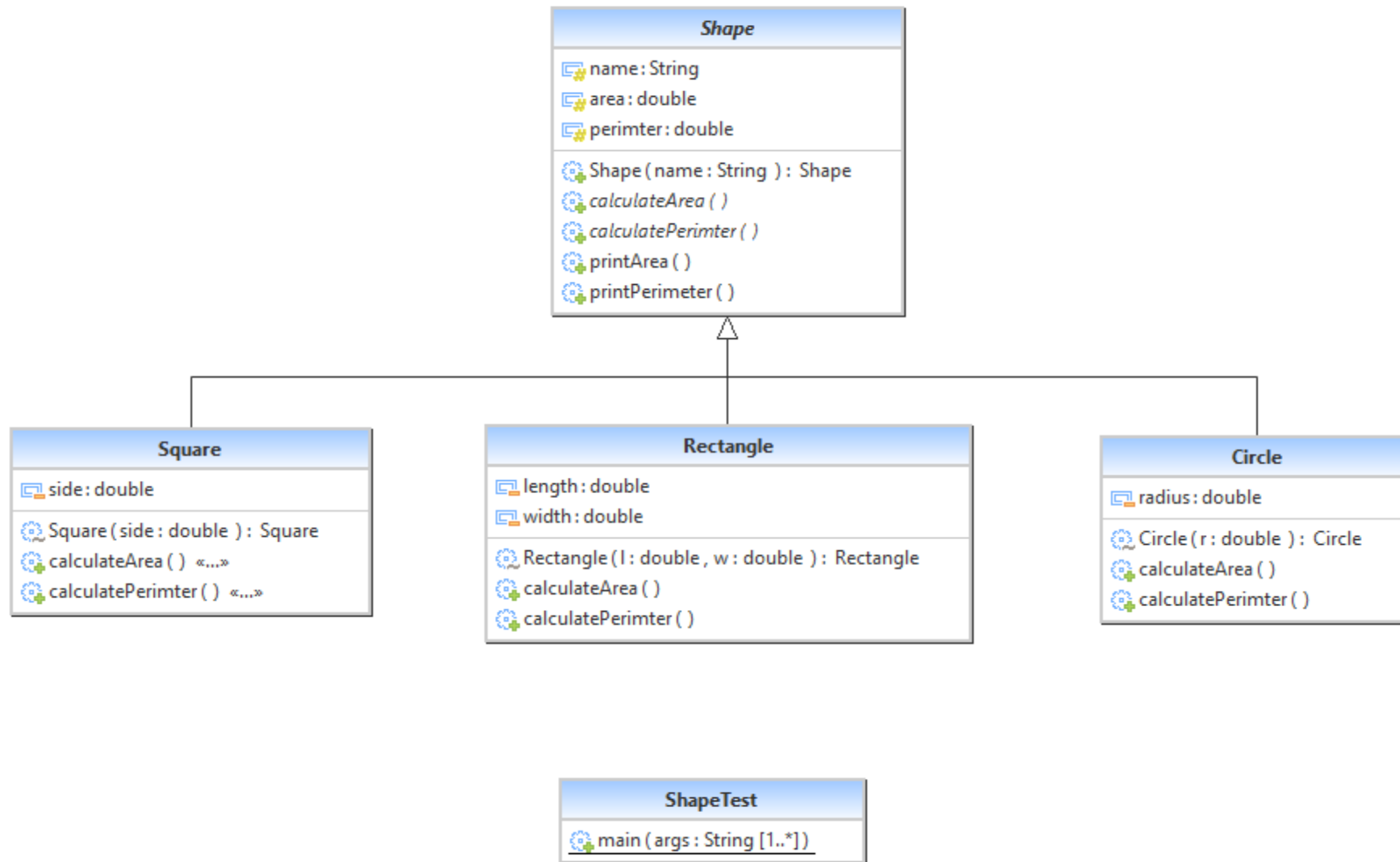


Taken from <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

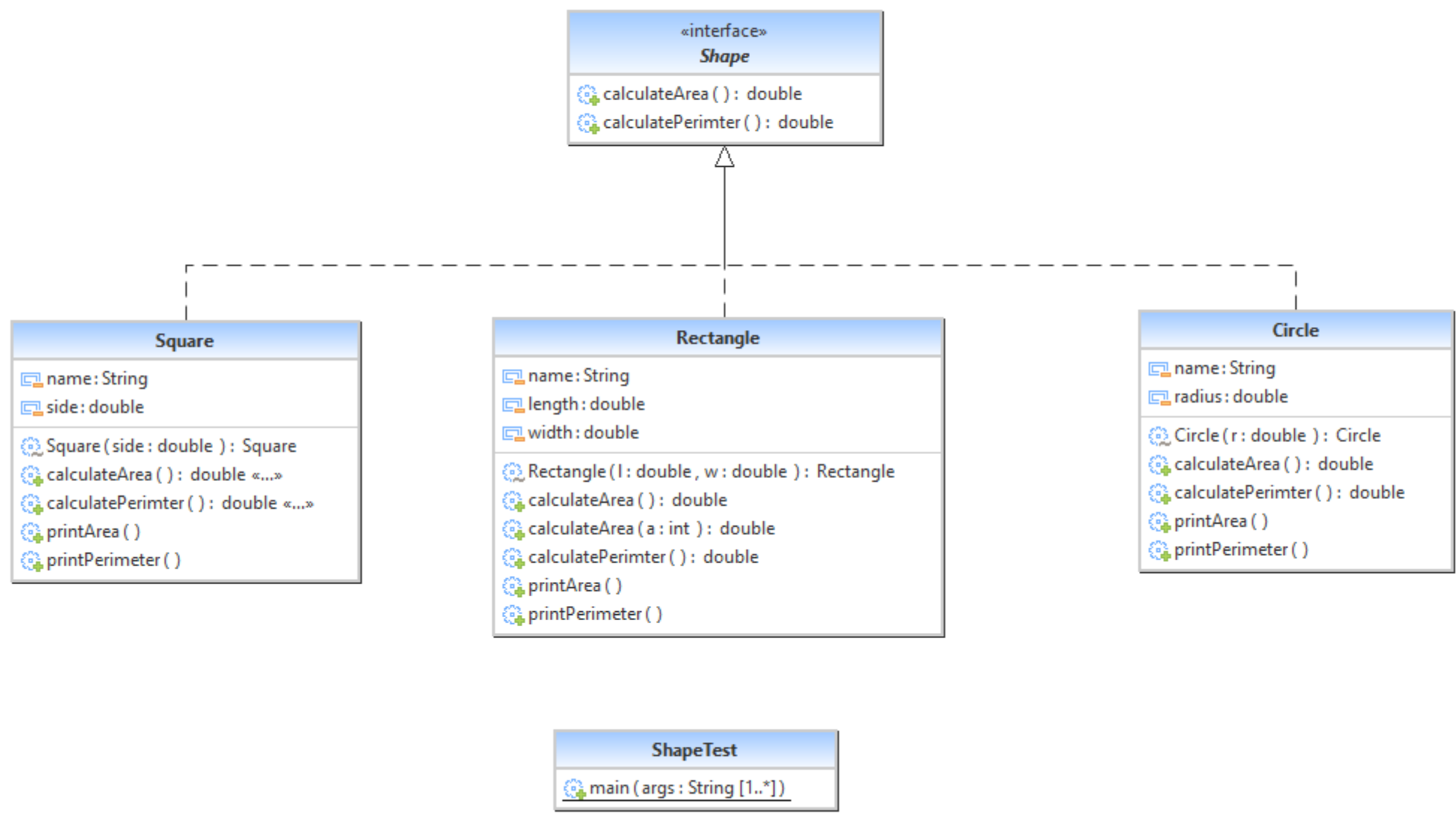
UML Class Diagram – College System



UML class diagram – Shape (abstract class)



Shape (interface)



- Final
 - Variables – makes them constant
 - Methods – cannot be overridden
 - Classes – cannot be subclassed



CST8132
OBJECT ORIENTED
PROGRAMMING

Testing
Javadoc
Midterm Preparation

Professor : Dr. Anu Thomas
Email: thomasa@algonquincollege.com
Office: T314

Today's Topics

- Labs
 - Lab 3 – Inheritance, abstract class, interface
- Hybrids
- Midterm
 - Week 7 – in class
 - MCQs
 - Worth 10% of term marks



Midterm Preparation

- On June 24 at 10:00 AM, during lecture
- You have 1.5 hours (90 minutes) to complete it
- There will not be a lecture, as soon as you are done you may leave
- There will be around 50 MCQ questions...
 - Multiple choice, multi-select, ordering, fill in the blanks etc.
- You need to join zoom by 9:50 AM.
- Webcam should be on for the full duration... I can see your activities in zoom and in Brightspace
- I will check IP address from zoom & Brightspace, and if they are not matching, you will get a 0



Midterm Preparation

- CAL Students : must send their CAL letters by Friday June 18, 11:59 PM.



What should you learn?

- Encapsulation
- Polymorphism
- Abstraction
- Inheritance
- Abstract class
- Overriding/Overloading
- Interface
- ArrayList
- Multi-dimensional arrays

- this keyword
- super keyword
- final keyword
- Static keyword
- Access specifiers
- Constructors
 - default,
 - no-arg
 - parameterized
- Relationships
 - Association
 - Aggregation
 - Composition
 - Cardinality

- Variables
 - Local
 - instance
- Packages
- Testing
- Javadoc



Recap

1. Inheritance

- Super-class, sub-class
- Super keyword – `super()`, `super.method()`

2. Polymorphism

- Taking multiple forms
- Static polymorphism – compile time (ex. Method overloading)
- Dynamic polymorphism – runtime (ex. Method overriding)

3. Abstract class

- Providing abstract of methods.... Will be overridden in child classes
- Can have non-abstract methods – they can have their definitions in the abstract class

4. Interface

1. Only abstract methods and public static final variables
2. Abstract keyword not required – by default, methods are abstract

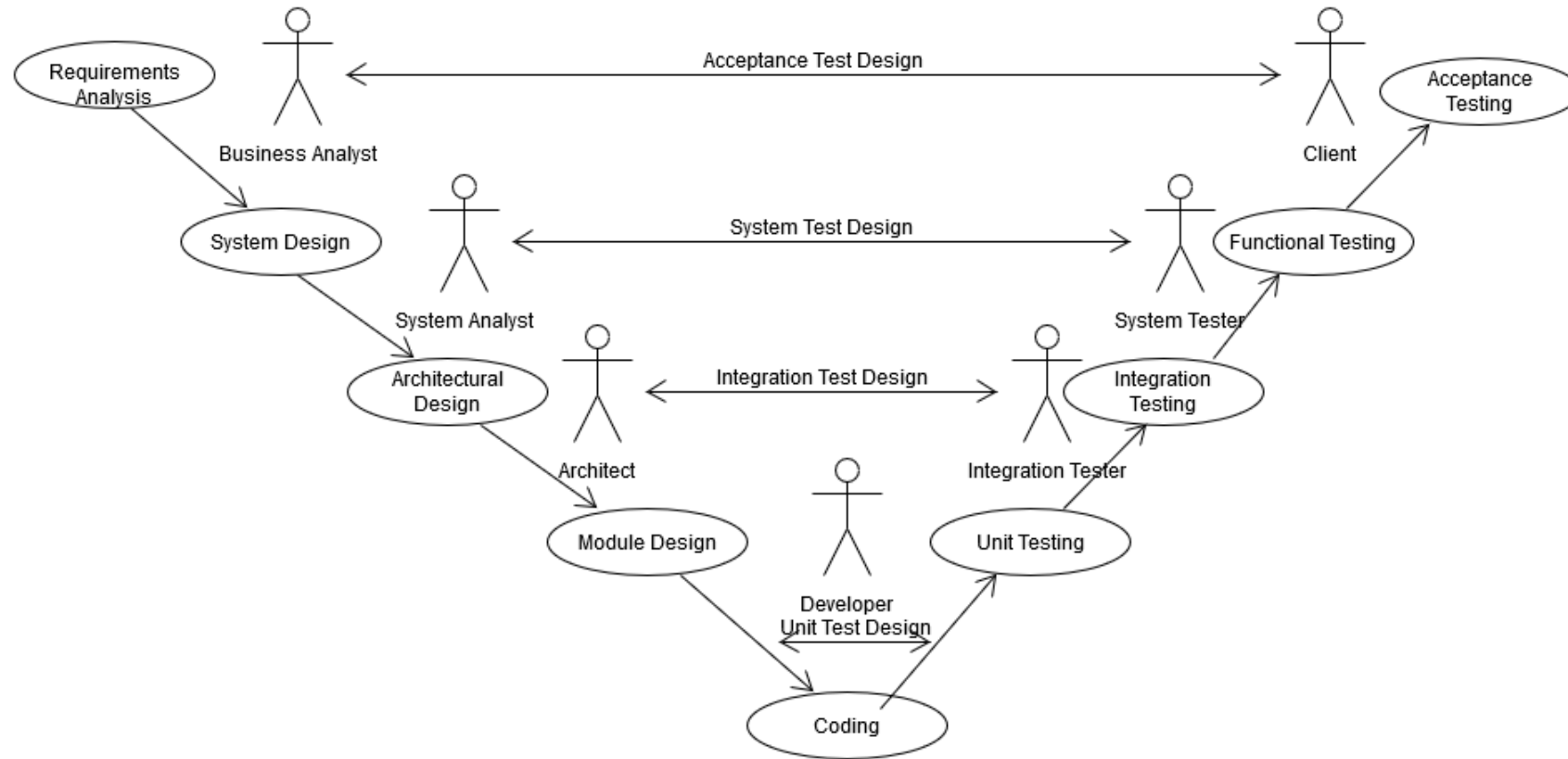


Test Plans

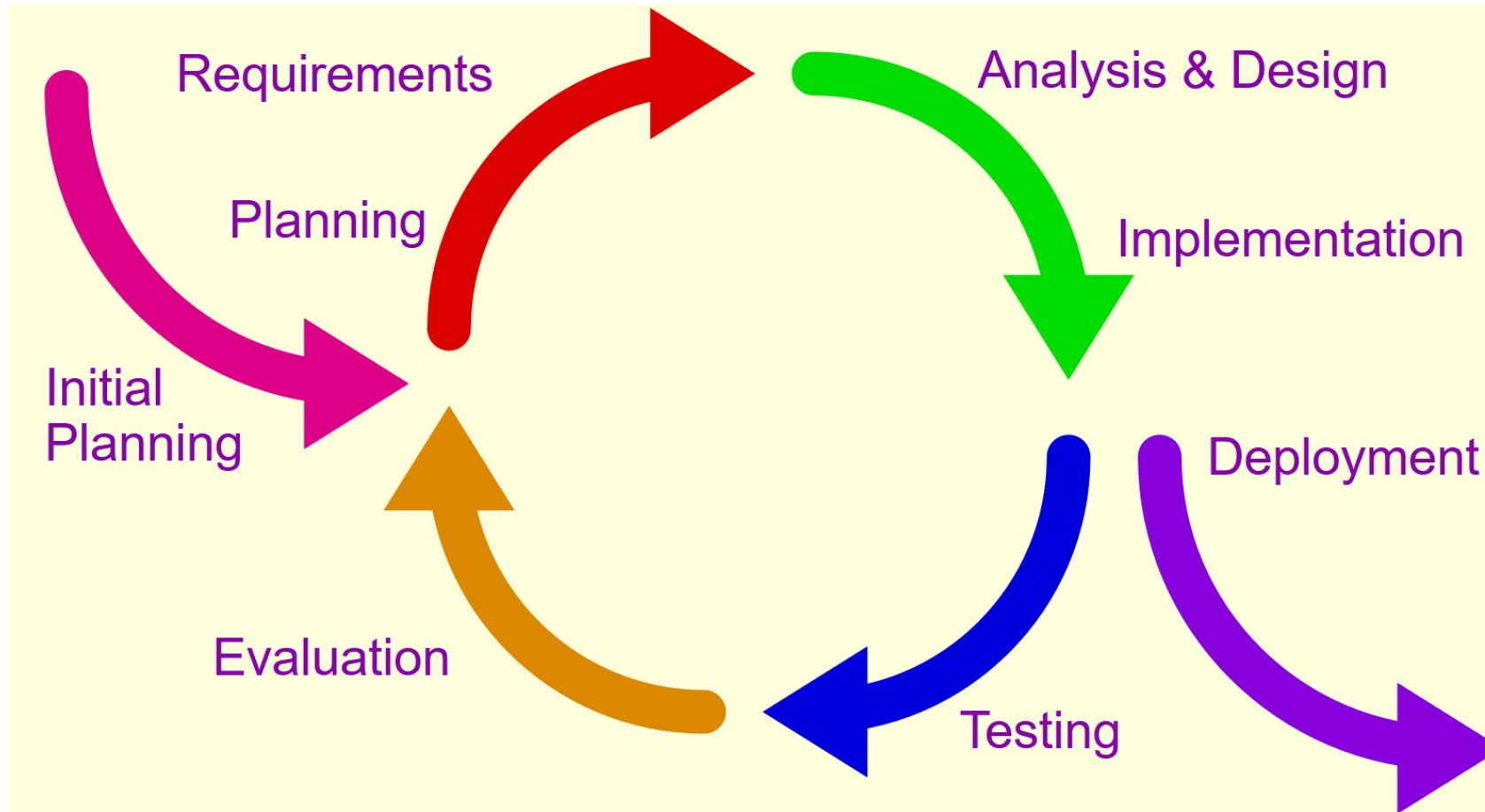
- A test plan is a comprehensive strategy for ensuring that the system does not contain errors:
 - Unit testing:
 - testing the individual units of our programs (classes, methods)
 - Integration testing:
 - putting the units together, and testing their use of each other's interfaces etc
 - Functional testing:
 - tests whether the overall system matches the behaviour defined in the system requirements
 - Acceptance testing
 - Evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.



V model (Waterfall)



Iterative Development Model (Agile)



Boundary Conditions

- In developing good tests for your code, you will need to identify all the boundary conditions
- You want to test with data that probes the boundary conditions
- What are the boundaries in an array?

```
Int [] num = new int[100];
```

- Array indices: -1, 0 , 1
- Array indices: 99, 100
- Boundaries on int values? Integer.MAX_VALUE, Integer.MIN_VALUE, int values change sign: -1, 0, 1



Sample Test Plan

CST8132 – Object-Oriented Programming

Test Plan < – write project name >

Prepared By - <Student Name >

<add more tables if you have more features to test. Add more rows if you have more test cases>

Feature: Menu Processing

Test #	Description	Input	Expected output/result	Actual Output	Status (Pass/Fail)
T1	Invalid selection – negative number	-1	Invalid Option—please try again...	Invalid Option—please try again...	Pass
	Invalid input - string	n	Invalid Option—please try again...	InputMismatch – program crashed	Fail
T7	Valid selection	1			

Feature: <Name of the feature >

Test #	Description	Input	Expected output/result	Actual Output	Status (Pass/Fail)



- JUnit is a facility for automated unit testing
- It supports test classes that run your methods and “look for the unexpected”
- Your tests will use the static methods of the Assert class to assert the condition you are testing for
 - Example
- See sections 1 – 4 of <http://www.vogella.com/tutorials/JUnit/article.html>



Summary

- Testing – How you are going to test if the program is working properly
 - Unit – lowest level testing (Class and methods)
 - Integration - test the integration between a set of components
 - Functional/System – test if the system works as a whole
- JUnit – Automated tool to unit test your code



Javadoc

- documentation generator created for the Java language
- intended for other developers, and represents the API of your Java class.
- produces formatted HTML documentation easily for your code.
- can produce automatic documentation from your code, however you can supplement this using special comments throughout your code.



Javadoc

- Javadoc comments must be enclosed by
 - `/** */`
 - They **MUST** begin with `/**`
- Javadoc comments **MUST** appear on the line(s) immediately preceding the code they describe.
- Wherever a description is required, a proper descriptive sentence or sentence fragment is expected.
- Javadoc is **REQUIRED** on all assignments going forward.
- Javadoc is not a replacement for comments.



Javadoc

- consist of descriptive text, and tagged descriptors.
- tags begin with an @ symbol, and always use lower-case names.
- Some tags are required, and some are optional. Some may be conditionally repeated.
- conditionally required whether you are documenting classes, properties or methods.
- Class documentation does not replace the required header as described in requirements.



- Java Class Documentation

- Every class requires the following Javadoc to appear immediately above the class declaration (below any package or import statements).

```
/**  
 * The purpose of this class is ...  
 * @author   John Doe  
 * @version  1.0  
 * @since    1.8  
 **/  
public class MyClass {
```



- Java Property Documentation

- Every non-private property (instance or class variable) requires a Javadoc description to appear immediately above the variable declaration.

```
/** The current account balance. */  
protected double balance;
```

```
/** The unique account number. */  
protected long accountNum;
```

```
/** The client who is the account holder. */  
public Client client;
```



- Java Property Documentation
 - Declare every property on a new line.
 - Private properties do not require Javadoc.
 - Do not simply repeat the variable name – this is not sufficient. Javadoc gets the name and data type automatically. Your job is to write qualitative information about the property.
 - The valid range of values is a good thing to include in Javadoc documentation. For example:

```
/** The hour of day, 0 – 23. */  
protected int hour;
```



- Java Method Documentation
 - Every non-private method requires a Javadoc description to appear immediately above the method declaration.
 - Constructors and the main method require Javadoc!!!
 - Methods with parameters require the Javadoc to contain one **@param** tag for each parameter.
 - Methods with a return type require the Javadoc to contain one and only one **@return** tag for the return value.
 - When we begin exception handling, methods which throw exceptions should have an **@throws** tag for each exception type thrown.



- Java Method Documentation

- Example:

```
/** This method attempts to withdraw a
 *  specified amount from the account.
 *  @param amt
 *          The amount requested to withdraw.
 *  @return True if the withdrawal is
 *          successful, or false if the
 *          balance is insufficient for the
 *          amount requested.
 */
public boolean withdraw ( double amt ) {
```



- Javadoc Requirements
 - [description]
 - @author [author name]
 - If more than one author, each author has their own @author tag. Order from most recent author at the top of the list to the oldest at the end of the list.
 - @version [software version]
 - For your assignments, use a sensible version such as 1.0.
 - @since [jdk version]
 - We are using version 1.8. Could provide full version.
 - @param [name] [description]
 - The name of the parameter must match the Javadoc name.
 - @return [description]



- Javadoc Details
 - Javadoc descriptions may contain HTML mark-up for additional formatting opportunities.
 - Javadoc does NOT replace the need for comments in your code.
 - Code blocks such as control structures still require comments.
 - Javadoc should not appear inside of any methods – only comments belong inside of methods.
- To enable Javadoc validation in Eclipse:
 - Window > Preferences > Java > Compiler > Javadoc

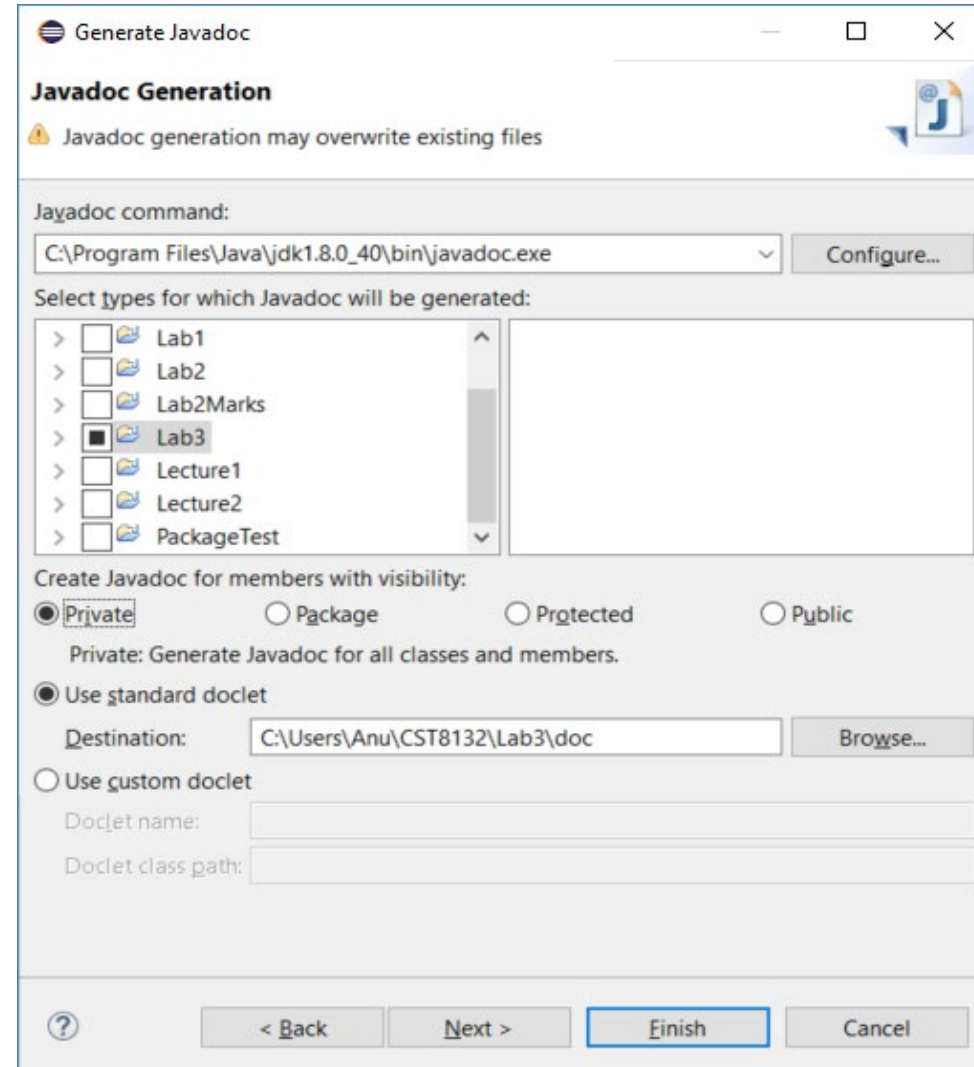
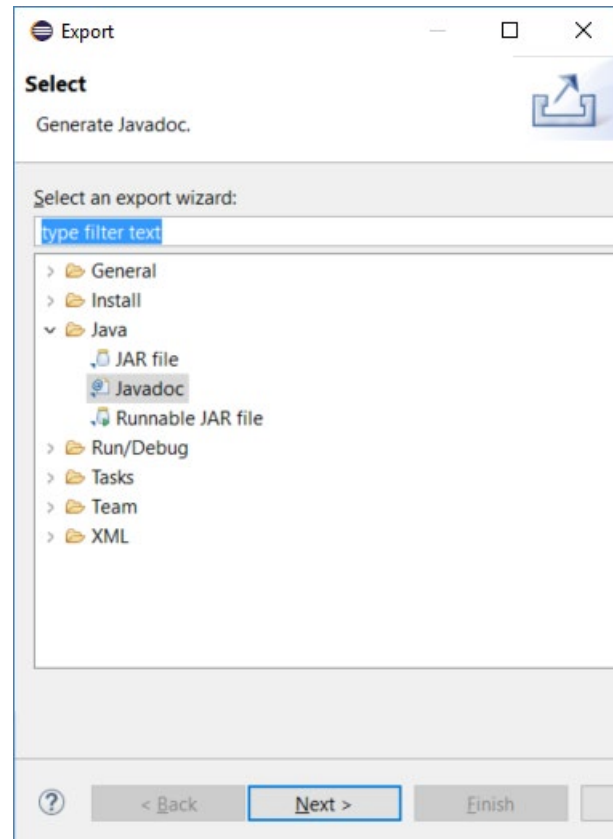


Generating Javadoc

- Javadoc will only be generated based on comments within `/** */` frame.
- To generate Javadoc in Eclipse:
 - Select the **File → Export → Java → Javadoc** menu option.
 - Click the **Configure...** button next to the Javadoc Command text box, and **browse** to your installed javadoc.exe file. For example:
 - C:\Program Files\Java\jdk1.8.0_40\bin\javadoc.exe.
 - Ensure that the project and ALL java files within are selected.
 - From **Create Javadoc for members with visibility**, select the **Package** option.
 - Leave the **Destination** with the default value.
 - Click **Finish**.



Generating Javadoc



What does JavaDoc output look like?

- API Documentation - <https://docs.oracle.com/en/java/javase/11/docs/api/index.html#>
- String - <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>



CST8132
OBJECT-ORIENTED
PROGRAMMING

Overview & Review

Professor : Dr. Anu Thomas

Email: thomasa@algonquincollege.com

Office: T314

Today's Topics

- Brightspace
- Course Outline
- Reflections on CST8116
- Intro to Object-Oriented Programming
- Multi-dimensional arrays
- Details on first lab: Lab 1



- Professor Information

- Dr. Anu Thomas
- T314
- thomasa@algonquincollege.com
- Office Hours: by appointment

- Course Information

- Course Outline
 - What this course is supposed to be
 - Details on evaluation: test, quizzes, labs, assignments, exam & final mark
- Course Section Information (Under Course Information)
 - A summary of what's going to happen and when throughout this term
 - Specific details on evaluation: test, quizzes, labs, assignments, exam & final mark



Brightspace (Cont'd)

- Weekly Learning Materials
 - Lecture Notes, PowerPoint slides etc.
 - before joining the class, check slides
- Hybrid Assignments
 - Materials and links that you will work on your own, with online quizzes
- Lab Assignments
 - This is the area where you will retrieve documents and related files for labs
 - Labs should be submitted before the due date and should be demoed to your lab professor during the lab sessions
 - Both submission AND demo are required to get grades
- **Survey - Location**



Brightspace (Cont'd)

- Discussion Board
 - We can use these to discuss details of lab requirements
 - Questions about intended purpose or satisfactory strategies etc.
 - There may not be much activity, but it is a resource available to us
 - Can have discussions on various course topics, and you can help each other by posting answers
 - Don't post code for any assignments!
- Announcements
 - When a student asks a question, and if it is relevant for the entire class, I will post an announcement



Resources

- **Required Textbook:**
 - Java, How to Program, 11th Edition by Deitel and Deitel, Published by Pearson Education Inc. in 2015 ISBN: 9780134743356
- **Recommended Textbook:**
 - Effective Java, 3rd Edition, Joshua Bloch, Addison-Wesley Professional, ISBN: 9780134686097
 - Big Java Early Objects, Seventh Edition, Cay Horstmann, Wiley, ISBN: 978-1-119635659
- **Software:**
 - Java Platform (JDK) 8uNN (latest Java 8 update)
 - - <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
 - Eclipse (as used in CST8116)
 - <http://www.eclipse.org/downloads>



Teaching / Learning Methods

- Lectures

- 2 hours per week
- some (but not all) notes/material posted on Brightspace
- DO NOT MISS LECTURES
- 1 hour hybrid per week; activity will be listed in Brightspace
- Bonus quizzes in class (from week 2 onward)

- Labs

- 2 hours per week – DO NOT MISS LABS
- 6 lab questions will be completed and demoed during the lab time – (that emphasize key concepts)
- Lab requirements will be posted in Brightspace



Your weekly schedule

- Lecture - Thursdays
 - Lecture from 10:00 M – 12:00 PM
- Lab – Mondays, Tuesdays & Fridays
 - Lab submissions are due on Sundays 11:59 PM
 - Labs should be submitted in Brightspace **AND** should be demoed during lab hours
 - CSI has detailed information of labs and its due dates
- Hybrid – due on Sundays
 - Released on Mondays, due on Sundays.
 - Multiple attempts are allowed, and will IGNORE all but the most recent submission
 - For hybrids 1, 5, and 9, only 3 attempts, but you need to do the questions that you got wrong in the previous attempt
- Lecture – Dr. Anu Thomas
- Labs: Dr. James Mwangi, Karan Kalsi



Evaluation

- **Midterms** 20%
 - Midterm 1 (10%)
 - Midterm 2 (10%)
- **Final Exam** 30%
- **Lab Assignments** 40%
 - 3 labs (5% for lab1, 7% for remaining labs)
- **Hybrid Exercises** 10%
 - 10 exercises worth 1% each
- **NOTE:** In order to pass this course, at least 50% (i.e., 25/50) must be achieved in the theory component (midterm and final). Additionally, at least 50% (i.e., 20/40) must be achieved in the applied component (labs). Even if your combined grade exceeds 50% for the entire course, if you fail either the theory component or the applied component you will not achieve a passing grade in the course.



Plagiarism

- **The School of Advanced Technology's Standard Operating Procedure on Plagiarism and Academic Honesty** defines plagiarism as an **attempt to use or pass off as one's own idea or product, work of another without giving credit**. Plagiarism has occurred in instances where a student either directly copies another person's work without acknowledgement; or, closely paraphrases the equivalent of a short paragraph or more without acknowledgement; or, borrows, without acknowledgement, any ideas in a clear and recognizable form in such a way as to present them as one's own thought, where such ideas, if they were the student's own would contribute to the merit of his or her own work.
- Plagiarism is one of the most serious academic offences a student can commit. Anyone found guilty will, on the first offence, be given a written warning and an "F" on the plagiarized work. If the student commits a second offence, an "F" will be given for the course along with a written warning. A third offence will result in suspension from the program and/or the college.



Reflections on CST8116

- Discussion on
 - What did you like about CST8116?
 - What did you dislike about CST8116?
 - Was it more difficult or easier than expected? How?
 - Was the content of CST8116 as you expected? If not, what did you expect?



CST8116 Review – JDK, JVM, JRE

- JVM – Java Virtual Machine
 - interprets Java bytecode on native hardware (HW)
 - Just In Time (JIT) compiler optimizes frequently used bytecode
 - Browser plug-in is JVM with security restrictions
- JRE – Java Runtime Environment
 - implementation of JVM, libraries & other binaries
- JDK – Java Development Kit
 - JRE plus compiler, debugger & other tools
 - JDK & JRE specific to each combination of HW & OS



CST8116 Review – JDK, JVM, JRE

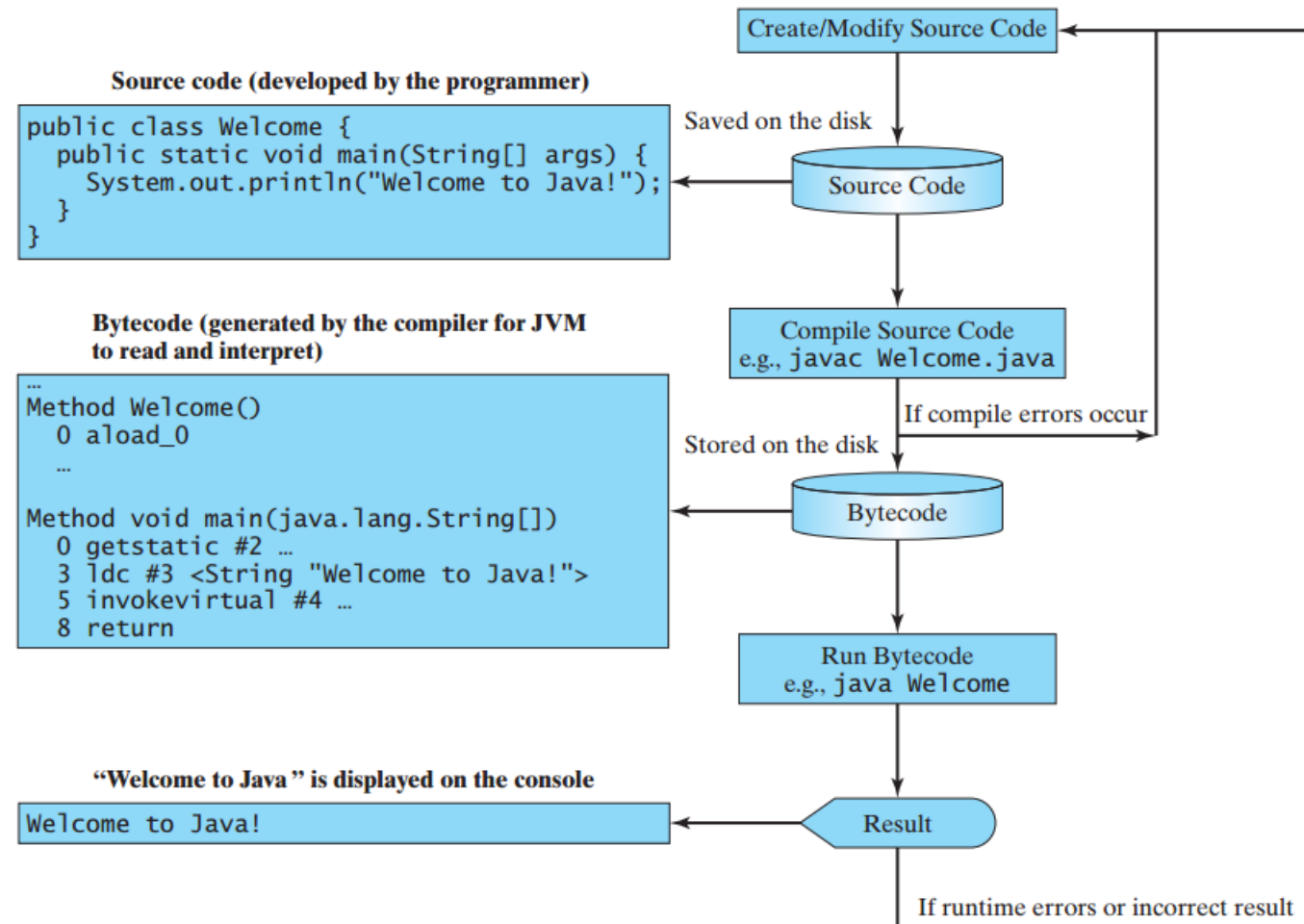


FIGURE 1.6 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.

Development Cycle for Java

- Step 1 - Editor
 - Type in our instructions/statements according to rules of Java Language
 - File from editor is called a source file and will have extension **.java**
- Step 2 - Compiler
 - Use compiler to translate the statements in source file to bytecode (which are portable – not dependent on hardware platform)
 - File from compiler is called bytecode file and will have extension **.class**
 - Use command> **javac ProgramName.java**
- Step 3 – Class Loader
 - class loader loads bytecode files into memory (includes all the .class files that you have written/produced and those provided by Java that you are using)
 - Use command> **java ProgramName** to invoke Loader, Verification, Execution
- Step 4 – Bytecode verifier
 - Use bytecode verifier to ensure bytecodes are valid and do not violate Java's security restrictions
- Step 5 – Execution
 - Use JVM (Java Virtual Machine) to execute program's bytecodes (to perform the actions specified by your program)



CST8116 Review – Data Types

- Primitive Data Types

- You should know sizes and literals for each

boolean	int
char	long
byte	float
short	double

- Reference Data Types

- Object
 - String
 - Arrays



CST8116 Review – Operators

- Relational and Logical Operators
 - `<` `>` `==` `!=` `<=` `>=` `&&` `||`
- Arithmetic, Compound Assignment, Increment/Decrement
 - `+` `-` `*` `/` `%` `+=` `-=` `*=` `/=` `%-`
 - `++` `--`
 - Integer arithmetic (including rounding and truncation)
- Operator precedence



CST8116 Review – Output

- Standard Output
 - `System.out.print()`
 - `System.out.println()`
 - `System.out.printf()`
- Special / escaped characters
 - `\n` `\t` `\\` `\”` `\’` `\r`
- Format specifications
 - Including width, precision, justification, and leading zeros
 - `%s` `%d` `%x` `%X` `%f` etc

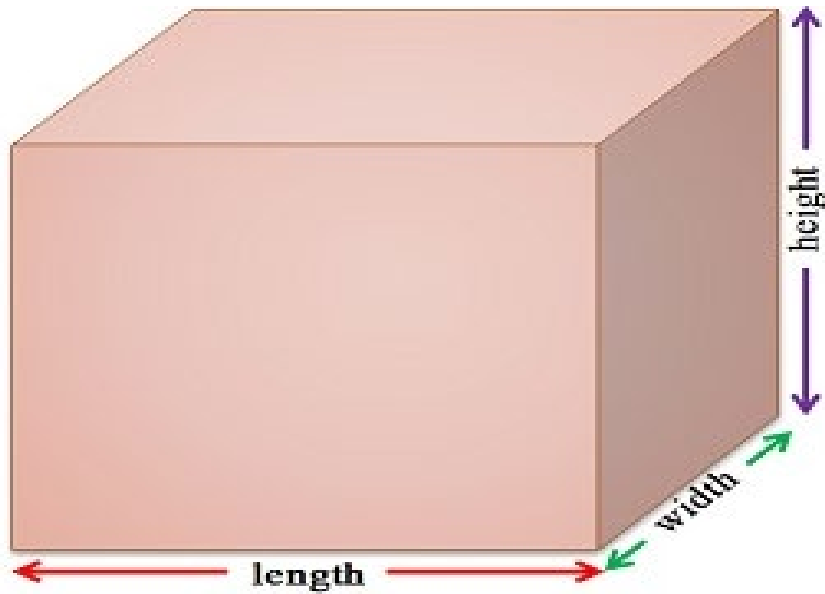


Object-Oriented Programming (OOP)

- In Object-Oriented programming, everything is an Object
- In Object-Oriented programming, everything **is-an** Object
- We now have a new OOP term: **is-a**
 - More on this later
- Every object is an instance of the Object Class
- What is the difference between an object and a class?



Box



What can we do with this data?

- Data (Attributes)
 - Length
 - Width
 - Height
- Find Volume?
- Find Surface Area?
- Print Volume?
- Print Surface Area?

Picture taken from:
<https://keydifferences.com/wp-content/uploads/2017/01/length-vs-height.jpg>



Box of Mouse

- Length = 16 cm
- Width = 8 cm
- Height = 3 cm



- Volume = length * width * height = 384



Box of Pencils

- Length = 16 cm
- Width = 7 cm
- Height = 2.5 cm
- Volume = length * width * height = 280



Class – An example

```
public class Person{  
    private int age;  
    public Person(){  
        age = 4;  
    }  
}
```



*we put this in a file called
Person.java*

- Everything is an object
- Here, Person can have many attributes or properties like age, name, dob, weight, height etc.
- If we want to manipulate these attributes or values, we need to write methods around it
- When we wrap attributes and its corresponding methods into one single entity, that entity is called Class



Objects in OOP

- Class
 - A class is like a blueprint/drawings of a house
 - A class is the definition of a type of object
 - You cannot live in a blueprint – a class is not an object
- Object
 - An object is like the actual house
 - You can make as many houses as you want from one blueprint
 - Those houses will look the same, but there are several of them
 - Objects are created at run-time by instantiating instances of classes
 - For the class Person, we can create its object with this java code:

```
Person person1 = new Person();
```
- Instance
 - Objects of the same type are called instances of a class



- We use classes in OOP to specify two things:
 1. What are the attributes of the objects we have in mind
 - ✓ The attributes of an object define its state
 2. What can those objects do (methods)
 - ✓ The methods of an object define its behavior
- So, we have learned to think of objects (Person, Object, Scanner etc) as having **state** and **behavior**
- Our Person class on the previous slide is rather silly:
 - ✓ What comprises the state of a person? As we've written it, just age.
 - ✓ What are the behaviors of a person? As we've written it, none.



public static void main(String [] args)

- The main method is a *static* method (notice the keyword static)
 - Static methods are associated with the class itself, not the object instances of the class (more on static later)
 - For our purposes right now, this means we don't need an object to exist before we call a static method
 - The main method is the starting point for Java programs
 - When the program starts, no objects exist, so main must be static
 - In CST8116, you may have done substantial programming in the main method
 - In OOP, we use the main method just to create the first objects and start them *talking* to each other



Discussion on methods

- Print age of `person1`
- Can we have a `person2`?
- Can `person2` have a different age?

Let's implement the complete solution for review



Features of OOP

- Encapsulation
- Abstraction
- Polymorphism
- Inheritance

We will learn about these features in this term!



CST8116 Review -Control Structures

- Branching Control Structures
 - if (*condition*) { }
 - if (*condition*) { } else { }
 - if (*condition*) { } else if (*condition*) { } else { }
 - switch (*var*) { case *value*: default: }



- Looping Control Structures
 - while (*condition*) { }
 - do { } while (*condition*);
 - for (*initialization; termination; increment*) { }
 - for (Object o : [collection|array]) { }
 - Enhanced for loop
 - For-each loop



CST8116 Review – Arrays

- An array is a container object which holds a specified number of values of a single data type.
- The length of the array is specified when the array is initialized.
 - Once created, the length of an array is fixed.
- Each item in an array is called an **element**
- Each element is accessed by its **index**.
- Index numbering always begins with **0**.

98
95
87
96
92

marks



CST8116 Review – Arrays

- An array declaration has two required parts:
 - The array's type, written as *type*[]
 - The array's name
- An array must be both declared and initialized before the elements of the array may be accessed.
- Arrays are often accessed using looping constructs.

```
int []marks= new int[5];
```

marks	
marks[0]	95
marks[1]	93
marks[2]	98
marks[3]	92
marks[4]	97



CST8116 Review

This course builds on CST8116, please review this material so you don't fall behind.

- Questions?
- Comments?
- Feedback?



Multidimensional Arrays

What if you need to express your information in more than one dimension?

Example:

- Desk row/seat number
- Building/floor/room
- Year/month/day/hour/minutes

Seating Arrangement

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	John	Doe	Bill	Tom	Sam
Row 2	Nora	Lily	Emma	Sophia	Maya
Row 3	Thomas	Leo	Jacob	James	Jack

15 students to be seated in 3 rows. Each row has 5 seats.

Could we use a single dimensional array?



Multidimensional Arrays

Declaration would be

```
maxRows = 3; maxSeats = 5;  
String[][] studentName = new String[maxRows][maxSeats];
```

To get/set student name:

```
String name = studentName[row][seat];  
studentName[row][seat] = "John";
```

A method declaration would be:

```
public void printName(String[][] names) {}
```



Multidimensional Arrays

- `names.length` returns the number of rows
- `names[0].length` returns the number of seats
- `names[3][4]` only mean row 3 seat 4
- `name[2][25]` would create a run time error
- **What about the different sized rows?**



Multidimensional Arrays

For example if each row has twice as many seats as the previous.

```
String studentName[][] = new String[5][];  
studentName[0] = new String[1];  
studentName[1] = new String[2];  
studentName[2] = new String[4];  
studentName[3] = new String[8];  
studentName[4] = new String[16];
```



Multidimensional Arrays

- We can even initialize an array

```
String wordLength[][] = {  
    {}, // one letter numbers  
    {}, // two letter numbers  
    {"one", "two", "six", "ten"}, // three letter numbers  
    {"four", "five", "nine"}, // four letter numbers  
    {"three", "seven", "eight"}, // five letter numbers  
};
```



2-dimensional Arrays

	Column 1 index 0	Column 2 index 1	Column 3 index 2	Column 4 index 3	Column 5 index 4
Row 1 index 0	0,0	0,1	0,2	0,3	0,4
Row 2 index 1	1,0	1,1	1,2	1,3	1,4
Row 3 index 2	2,0	2,1	2,2	2,3	2,4

	Column 1 index 0	Column 2 index 1	Column 3 index 2	Column 4 index 3	Column 5 index 4
Row 1 index 0	0,0	0,1	0,2	0,3	0,4
Row 2 index 1	1,0	1,1	1,2	1,3	1,4
Row 3 index 2	2,0	2,1	2,2	2,3	2,4

72
21
98

Two-dimensional array: One-dimensional array with each element as a one-dimensional array



3-dimensional Arrays

- One-dimensional array with two-dimensional arrays
- Example: Multi-level Parking lot
 - Parking lot with different levels, each level with rows and columns of parking spaces
 - Car parked on 5th spot of 3rd row of level 4 ➔ 4th level, 3rd row, 5th position
 - `carpark[4][3][5]`



Multidimensional Arrays

- Questions
- Comments
- Feedback



In-class exercise (solution will be provided next week)

- Write a class named Numbers with the following:
 - `public void generateNumberTable()`
 - `public void pascalTriangle(int size)`
 - https://en.wikipedia.org/wiki/Pascal%27s_triangle
 - `public void printTable()`
- Write a class named NumbersTest with the following:
 - “main” to call the three methods



In-class exercise

```
public static void main(String[] args) {  
    System.out.println("\n\nTable of Integers");  
    Numbers n1 = new Numbers(5,10);  
    n1.generateNumberTable();  
    n1.printTable();  
  
    System.out.println("\n\nPascal Triangle");  
    Numbers n2 = new Numbers();  
    n2.pascalTriangle(5);  
    n2.printTable();  
}
```

Output:

Table of Numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Pascal Triangle

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1