Object Oriented Programming with Java II

Dr. Mohamed K. Hussein

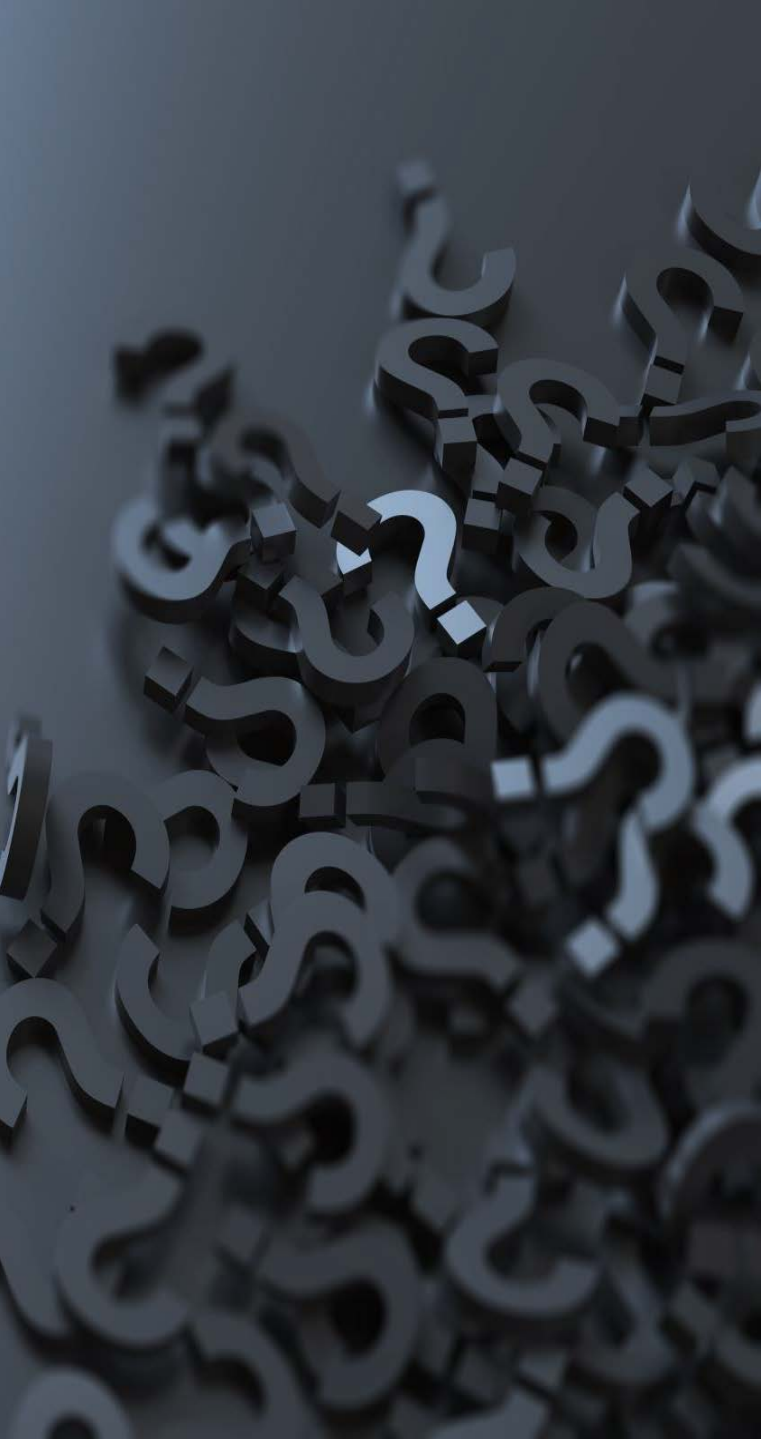
Lecture 5

Associate Prof., Computer Science department,

Faculty of Computers & Information Science,

Suez Canal University

Email: m_khamis@ci.suez.edu.eg



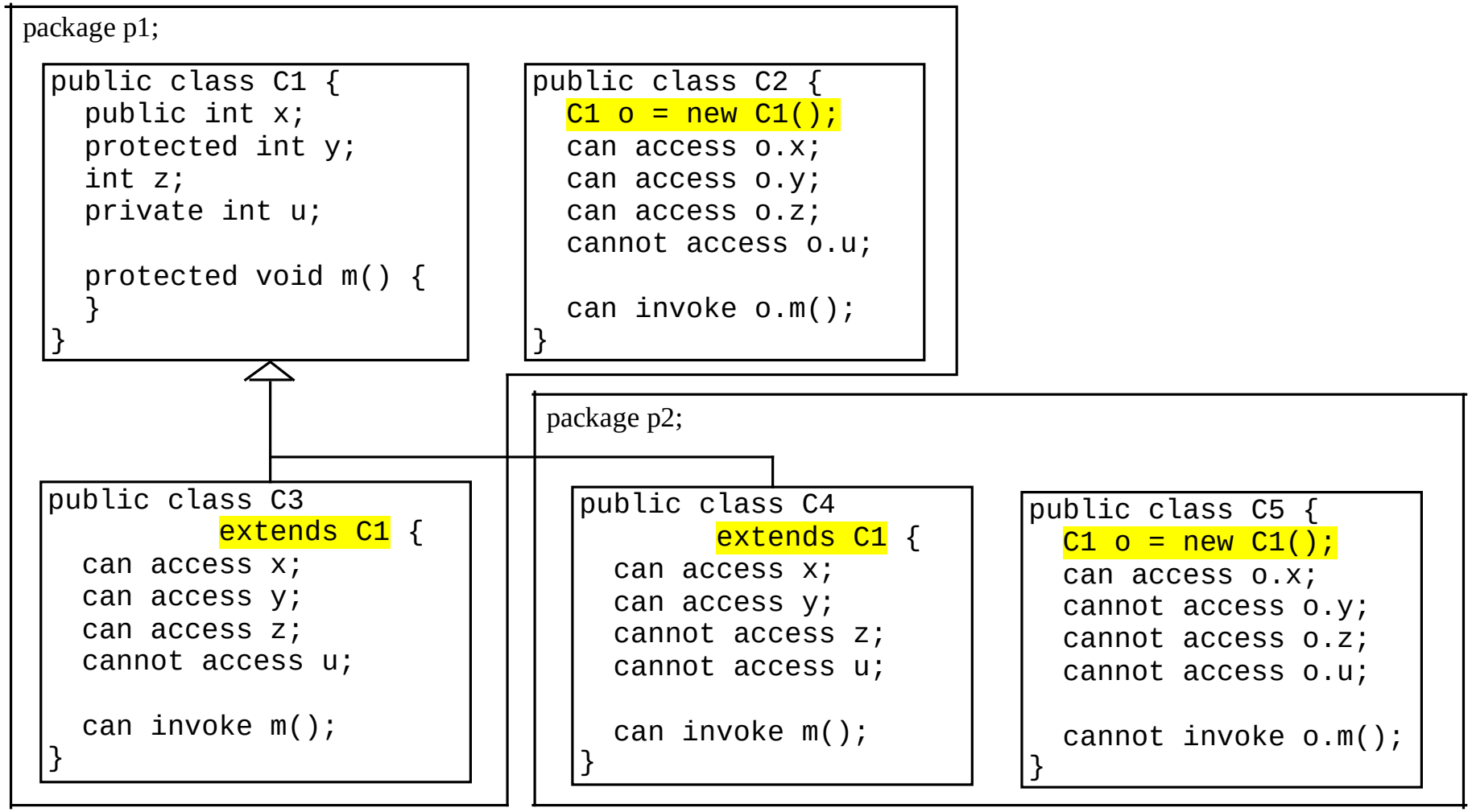
Lecture outcomes

- The protected Data and Methods
  - Visibility Modifiers
- Preventing Extending and Overriding
- Polymorphism
  - Dynamic Binding
- Casting Objects
  - The instanceof Operator
- The Object's equals Method
- The ArrayList Class

The protected Data and Methods

- *A protected member of a class can be accessed from a subclass.*
  - private members can be accessed only from inside of the class,
  - public members can be accessed from any other classes.
- protected allow subclasses to access data fields or methods defined in the superclass,
  - Do not to allow nonsubclasses to access these data fields and methods.

Visibility modifiers



Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Preventing Extending and Overriding

- *Neither a final class nor a final method can be extended.*
  - *A final data field is a constant.*
- The **final** modifier indicates that a class is final and cannot be a parent class.
  - The **Math** class is a final class.
  - The **String**, **StringBuilder**, and **StringBuffer** classes are also final classes.

```
public final class A {
        // Data fields, constructors, and methods
}
```

Preventing Extending and Overriding

- *A final method cannot be overridden by its subclasses.*
  - For example, the following method **m** is final and cannot be overridden:

```
public class Test {
    // Data fields, constructors, and methods
    public final void m() {
        // Do something
    }
}
```

Polymorphism

- *Polymorphism means that a variable of a supertype can refer to a subtype object.*
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
  - A subclass is a specialization of its superclass
  - Every instance of a subclass is also an instance of its superclass, but not vice versa.
    - For example, every student is a person object, but not every person object is a student.

```
public class Person{
    private int age;
    public Person(){age = 0;}
    public Person(int age){this.age = age;}
    public int getAge(){return age;}
    public void setAge(int age){this.age = age;}
    public String toString(){
        return “Age = “ + age;
    }
}
```

```
class Student extends Person{
    private int gpa;
    public Student(){
      Super();
      gpa = 0;
    }
    Public Student(int age, int gpa){
        Super(age);
        this.gpa = gpa;
    }
    public string toSTring(){
     return Super.toStrng() + “\n” + “gpa = “ + gpa;
    }
}
```

```
public class MyClass {
      /** Main method */
      public static void main(String[] args) {
            printData(new Student(5, 3.2));
            printData(new Person(10));
      }

      // Display object properties
      public static void printData(Person object) {
            System.out.println(“Data:\n" + object.toString());
      }
}
```

Dynamic Binding

- *A method can be implemented in several classes along the inheritance chain.*
  - *The JVM decides which method is invoked at runtime.*
- A method can be defined in a superclass and overridden in its subclass.
  - For example, the **toString()** method is defined in the **Object** class and overridden in **Person**.
  - Consider the following code:

```
Object o = new Person();
System.out.println(o.toString());
```

Which toString() method is invoked by o?

Dynamic Binding

- There are two terms: declared type and actual type.
  - A variable must be declared a type.
    - The type that declares a variable is called the variable's *declared type*.
      - **o**'s declared type is **Object**.
    - A variable of a reference type can hold a **null** value or a reference to an instance of the declared type.
    - The instance may be created using the constructor of the declared type or its subtype.
    - The *actual type* of the variable is the actual class for the object referenced by the variable.
  - Here **o**'s actual type is **Person**, because **o** references an object created using **new Person()**.
    - Which **toString()** method is invoked by **o** is determined by **o**'s actual type.
- This is known as ***dynamic binding***.

Casting Objects

- *One object reference can be typecast into another object reference.*
  - *This is called casting object*.
- The statement m(**new** Person());
  - Assigns the object **new Student()** to a parameter of the **Object** type.
- This statement is equivalent to Object o = **new** Person(); // Implicit casting m(o);
- The statement **Object o = new Person()**, known as *implicit casting*, is legal because an instance of **Person** is an instance of **Object**.

Casting Objects

- Consider the following statement: Person b = o;
  - In this case a compile error would occur.
  - Why does the statement **Object o = new Person()** work but **Person b = o** doesn't?
- The reason is that a **Person** object is always an instance of **Object**,
  - but an **Object** is not necessarily an instance of **Person**.
- Even though you can see that **o** is really a **Person** object, the compiler is not clever enough to know it.
  - To tell the compiler that **o** is a **Person** object, use *explicit casting*.
  - The syntax is similar to the one used for casting among primitive data types.
  - Person b = (Person)o; // Explicit casting

instanceof

- For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
  - If an object is not an instance of **person**, it cannot be cast into a variable of **object**.
  - It is a good practice to ensure that the object is an instance of another object before attempting a casting.
- This can be accomplished by using the *instanceof* operator.

```
Object myObject = new Person();
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Person) {
        System.out.println("The perso age is " + ((Person)myObject).getAge());
}
```

Example

```
public class CastingDemo {
	public static void main(String[] args) {
		// Create and initialize two objects
		Object object1 = new Student(21, 3);
		Object object2 = new Person(12);

		// Display Person and Student
		displayObject(object1);
		displayObject(object2);
	 }
	public static void displayObject(Object object) {
		if (object instanceof Student) {
			System.out.println("The Student age is " +  ((Student)object).getAge() + "The Student gpa is " +  ((Student)object).getGpa());
		}
		else if (object instanceof  Person) {
			System.out.println("The Person age is " + ((Person)object).getAge());
		}
	}
}
```

The Object's equals Method

- *Like the* **toString()** *method, the* **equals(Object)** *method is another useful method defined in the* **Object** *class*
- This method tests whether two objects are equal.
  - The syntax for invoking it is: **object1.equals(object2);**

The Object's equals Method

- The default implementation of the **equals** method in the **Object** class is:

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- This implementation checks whether two reference variables point to the same object using the **==** operator.
- You should override this method in your custom class to test whether two distinct objects have the same content.
- The **equals** method is overridden in many classes in the Java API,
  - such as **java.lang.String** and **java.util.Date**, to compare whether the contents of two objects are equal.

The Object's equals Method

- You can override the **equals** method in the **Person** class
  - To compare whether two Person are equal based on their age as follows:

```
public boolean equals(Object o) {
    if (o instanceof Person)
            return age == ((Person)o).age;
    else
            return false;
}
```

The ArrayList Class

- *An* **ArrayList** *object can be used to store a list of objects.*

- You can create an array to store objects.
  - But, once the array is created, its size is fixed.
  - Java provides the **ArrayList**
- The following statement creates an **ArrayList** and assigns its reference to variable **cities**.
  - This **ArrayList** object can be used to store strings.
    - ArrayList<String> cities = **new** ArrayList<String>();

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	

ArrayList Example

```
import java.util.ArrayList;
public class TestArrayList {
        public static void main(String[] args) {
                ArrayList<String> cityList = new ArrayList<String>();              // Create a list to store cities
                cityList.add("Cairo");                                             // Add some cities in the list
                cityList.add("Alexandria");
                cityList.add("Ismilia");
                cityList.add("Mansoura");
                System.out.println("List size? " + cityList.size());
                System.out.println("Is Cairo in the list? " +  cityList.contains("Cairo"));
                System.out.println("The location of Alexandria in the list? "  + cityList.indexOf("Alexandria"));
                System.out.println("Is the list empty? " +  cityList.isEmpty());     // Print false
                cityList.add(2, "Giza");                                                    // Insert a new city at index 2
                cityList.remove("Cairo");                                          // Remove a city from the list
                cityList.remove(1);                                                // Remove a city at index 1
                System.out.println(cityList.toString());                           // Display the contents in the list
                for (int i = cityList.size() - 1; i >= 0; i--)      // Display the contents in the list in reverse order
                 System.out.print(cityList.get(i) + " ");
                 System.out.println();
        }
}
```

Array List Example

```
import java.util.ArrayList;
public class TestArrayList {
        public static void main(String[] args) {
                ArrayList<Person> People = new ArrayList<Person>();              // Create a list to store Persons
                People.add( new Person(18));                                     // Add some cities in the list
                People.add( new Person(22));

                System.out.println(People.toString());                           // Display the contents in the list

                for (int i = People.size() - 1; i >= 0; i--)                     // Display the contents in the list in reverse order
                 System.out.println( “The age of the person? " + People.get(i).getAge() );

                People.clear();
        }
}
```

Assignment 5

- The **Account** class is defined to model a bank account. An account class has:
  - Add a field name of the String type to store the name of the customer.
  - Other attributes include account number, balance, annual interest rate, and date created,
  - A no-arg constructor that creates a default account.
  - A constructor that constructs an account with the specified name, id, and balance.
  - Add a data field named **transactions** whose type is **ArrayList** that stores the transaction for the accounts. Each transaction include.
    - The date of this transaction.
    - The type of the transaction, such as 'W' for withdrawal, 'D' for deposit.
    - The amount of the transaction.
    - The new balance after this transaction.
    - Construct a Transaction with the specified date, type, balance, and description.
  - Methods to deposit and withdraw funds that adds the transaction to the arrayList.
- Create two subclasses for checking and saving accounts.
  - A checking account has an overdraft limit, but a savings account cannot be overdrawn.

Assignment 5

1. Draw the UML diagram for the classes and then implement them.
2. Write a test program that creates objects of **Account**, **SavingsAccount**, and **CheckingAccount** and invokes their **toString()** methods.
3. Creates an **Account** with annual interest rate **1.5%**, balance **1000**, id **1122**, and name **Sarah** .
4. Deposit $30, $40, and $50 to the account and withdraw $5, $4, and $2 from the account.
5. Print an account summary that shows account holder name, interest rate, balance, and all transactions.

Thank you