



University of Kelaniya

Kesavan Selvarajah

BSc (UOK), MSc (Reading, UCSC)

Temporary Lecturer

Software Engineering Department

Faculty of Computing and Technology

skesa231@kln.ac.lk

Object-Oriented Programming

CSCI 21052 / ETEC 21062



Programming paradigms

- All computer programs consist of two elements: **code** and **data**.
- Some programs are written around “**what is happening**” and others are written around “**who is being affected.**”
- These are the two paradigms that govern how a program is constructed.
- The first way is called the **process-oriented model**.
- This approach characterizes a program as a series of linear steps (code).
- The **process-oriented model** can be thought of as code acting on data.
- Procedural languages such as C employ this model to considerable success.
- However, problems with this approach appear as programs grow larger and more complex.



OO paradigm

- To manage increasing complexity, the second approach, called **Object-oriented programming**, was conceived.
- **Object-oriented programming** organizes a program around its data (objects) and a set of well-defined interfaces to that data.
- An **Object-oriented program** can be characterized as data controlling access to code.
- As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.



OOP Principles

- **Encapsulation**
 - Encapsulation can be defined as the binding of data and attributes or methods and data members in a single unit.
- **Inheritance**
 - Inheritance is the method of acquiring features of the existing class into the new class.
- **Polymorphism**
 - Polymorphism is the ability of something to have or to be displayed in more than one form.
- **Abstraction**
 - Abstraction can be defined as hiding internal implementation and showing only the required features or set of services that are offered.



Encapsulation

- **Encapsulation** is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
- Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- In Java, the basis of encapsulation is the class.



Encapsulation

- A class defines the structure and behavior (data and code) that will be shared by a set of objects.
- Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class.
- For this reason, objects are sometimes referred to as instances of a class.
- Thus, a class is a logical construct; an object has physical reality.



Encapsulation

- In properly written Java programs, the methods define how the member variables can be used.
- This means that the behavior and interface of a class are defined by the methods that operate on its instance data.
- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know, or may know.



Encapsulation

```
class Person {  
    private String name;  
    private int age;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        if (age >= 0) {  
            this.age = age;  
        }  
    }  
}
```



Encapsulation

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.setName("Alice");  
        person.setAge(30);  
        System.out.println("Name: " + person.getName());  
        System.out.println("Age: " + person.getAge());  
    }  
}
```



Encapsulation

- The above code demonstrates how to encapsulate the internal state of an object and provide controlled access to that state through getter and setter methods.
- We have a **Person** class with two private fields: *name* and *age*. These fields are made private to encapsulate them, preventing direct access from outside the class.
- Public getter methods (*getName* and *getAge*) allow controlled access to retrieve the values of the private fields. These methods provide read-only access to the encapsulated data.



Encapsulation

- Public setter methods (**setName** and **setAge**) allow controlled modification of the private fields.
- The **setAge** method includes validation to ensure that the age is non-negative.
- In the Main class, we create a **Person** object, set its name and age using the setter methods, and then retrieve and display the data using the getter methods.
- This demonstrates how encapsulation helps in maintaining data integrity and providing a controlled interface for interacting with the object's state.



Inheritance

- **Inheritance** is the process by which one object acquires the properties of another object.
- For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal.
- Without the use of inheritance, each object would need to define all of its characteristics explicitly.
- However, by use of inheritance, an object need only define those qualities that make it unique within its class.
- It can inherit its general attributes from its parent.
- Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.



Inheritance

```
class Person {  
    private String name;  
    private int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public int getAge() {  
        return age;  
    }  
}
```



Inheritance

```
class Student extends Person {
    private String studentId;

    // Subclass inheriting from the Person class
    public Student(String name, int age, String studentId) {
        super(name, age); // Call the superclass constructor using
        'super'
        this.studentId = studentId;
    }

    public String getStudentId() {
        return studentId;
    }
}
```



Inheritance

```
public class Main {  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.setName("Bob");  
        student.setAge(25);  
        student.setStudentId("S12345");  
        System.out.println("Name: " + student.getName());  
        System.out.println("Age: " + student.getAge());  
        System.out.println("Student ID: " + student.getStudentId());  
    }  
}
```



Inheritance

- The above code demonstrates the concept of inheritance in Java, which allows a class to inherit properties and methods from another class.
- **Person** is the base class (or superclass).
- It defines common attributes such as name and age that are shared among different individuals.
- **Student** is a subclass that inherits from the **Person** class.
- It extends the base class to add specific attributes and behaviors related to students.



Inheritance

- In the constructor of the **Student** class, we use the super keyword to invoke the constructor of the superclass (**Person**).
- This ensures that the name and age fields are properly initialized by the base class constructor.
- The **Student** subclass inherits the *getName* and *getAge* methods from the **Person** superclass.
- This demonstrates the reusability of methods through inheritance.



Inheritance

- The **Student** class introduces a new field, *studentId*, which is specific to students.
- This field is not present in the **Person** class.
- In this code, we do not override any methods.
- However, you can override methods from the superclass in the subclass to provide specialized behavior.
- This is a common practice in inheritance.



Inheritance

- In the Main class, we create an instance of the **Student** class.
- Even though the **Student** class inherits attributes and methods from the **Person** class, it has its own specific attributes, such as *studentId*.
- We can access these attributes and methods using the student object.
- Inheritance allows you to create a hierarchy of classes, with more specialized classes inheriting from more general ones.
- This promotes code reuse and helps model real-world relationships between objects.



Polymorphism

- **Polymorphism** (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation.
- More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”
- This means that it is possible to design a generic interface to a group of related activities.
- This helps reduce complexity by allowing the same interface to be used to specify a general class of action.
- It is the compiler’s job to select the specific action (that is, method) as it applies to each situation.



Polymorphism

```
class Person {  
    private String name;  
    private int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public void displayDetails() {  
        System.out.println("Name: " + name);  
        System.out.println("Age: " + age);  
    }  
}
```



Polymorphism

```
class Student extends Person {  
    private String studentId;  
    public Student(String name, int age, String studentId) {  
        super(name, age);  
        this.studentId = studentId;  
    }  
    @Override  
    public void displayDetails() {  
        super.displayDetails();  
        System.out.println("Student ID: " + studentId);  
    }  
}
```



Polymorphism

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Student("Bob", 25, "S12345");  
        person.displayDetails();  
    }  
}
```



Polymorphism

- The above code demonstrates the concept of polymorphism in Java, where objects of different classes can be treated as objects of a common superclass and provide different implementations for the same method.
- **Person** is the base class (or superclass) that defines a method *displayDetails*.
- This method displays general information about a person, including their name and age.



Polymorphism

- **Student** is a subclass that inherits from the Person class. It overrides the *displayDetails* method to provide specific information related to students, including their student ID.
- In the **Student** class, we override the *displayDetails* method.
- This allows the **Student** class to provide its own implementation of the method, in addition to invoking the superclass method using *super.displayDetails()* to include general information.



Polymorphism

- In the Main class, we create an instance of the **Student** class and assign it to a reference of the **Person** class.
- This demonstrates polymorphism, where the reference variable person is of the base class type, but it refers to an object of the derived class.
- Despite the reference variable being of type Person, when we call ***person.displayDetails()*** , the overridden method in the Student class is invoked.
- This is known as dynamic method dispatch and is a key feature of polymorphism.



Polymorphism

- The output of this code will display the details of the **Student** object, including both the general information from the **Person** class and the specific student information from the **Student** class.
- This demonstrates how polymorphism allows you to treat objects of different subclasses as objects of a common superclass, making your code more flexible and extensible.



Abstraction

- An essential element of object-oriented programming is **abstraction**.
- Humans manage complexity through abstraction.
- For example, people do not think of a car as a set of tens of thousands of individual parts.
- They think of it as a well defined object with its own unique behavior.
- This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the individual parts.
- They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.



Abstraction

- The data from a traditional process-oriented program can be transformed by abstraction into its component **objects**.
- A sequence of process steps can become a collection of messages between these objects.
- Thus, each of these objects describes its own unique behavior.
- You can treat these objects as concrete entities that respond to messages telling them to do something.
- This is the essence of object-oriented programming.



Abstraction

```
abstract class Person {  
    public String name;  
    public int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public abstract void displayDetails();  
}
```



Abstraction

```
class Student extends Person {  
    public Student(String name, int age) {  
        super(name, age);  
    }  
    @Override  
    public void displayDetails() {  
        System.out.println("Name: " + name);  
        System.out.println("Age: " + age);  
        System.out.println("Student Details");  
    }  
}
```



Abstraction

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Student("Bob", 25);  
        person.displayDetails();  
    }  
}
```



Abstraction

- The above code demonstrates the concept of abstraction in Java, which involves defining an abstract class with one or more abstract methods that must be implemented by concrete subclasses.
- **Person** is an abstract class.
- An abstract class is a class that cannot be instantiated on its own but can be extended by concrete (non-abstract) subclasses.
- It is used to define a common interface or set of methods that must be implemented by its subclasses.



Abstraction

- The **Person** class contains an abstract method called *displayDetails()*.
- Abstract methods are declared without a body and are marked with the **abstract** keyword.
- Subclasses of **Person** are required to provide concrete implementations for this method.
- **Student** is a concrete subclass of the abstract class **Person**.
- It extends **Person** and provides a concrete implementation of the *displayDetails()* method.



Abstraction

- The **Person** and **Student** classes have constructors to initialize the name and age fields.
- The **super** keyword is used to call the constructor of the superclass.
- In the **Student** class, we implement the ***displayDetails()*** method to provide specific details related to a student.
- The method overrides the abstract ***displayDetails()*** method declared in the **Person** class.



Abstraction

- In the **Main** class, we create an instance of the **Student** class and assign it to a reference of the **Person** class.
- This demonstrates the concept of polymorphism, where a reference of the base class can refer to an object of a derived class.
- We then call the ***displayDetails()*** method on this reference, which invokes the overridden method in the **Student** class, displaying the student's details.
- Abstraction allows you to define a common interface and enforce the implementation of specific methods in subclasses while providing flexibility and extensibility in your code.



Thank you

