Week 4 Object Oriented Design (OOD)

4009 DATA STRUCTURES & ALGORITHM (DSA)

Learning Outcomes

- Discuss the three principles of object-oriented design.
- Explain the concept of inheritance in Java
- Understand and implement object creation and referencing in Java
- Discuss the concept of Polymorphism
- Differentiate between method overriding and method overloading
- Differentiate between throwing exceptions and catching exceptions
- Understand the technique of casting and interfaces

Content Overview

- Object Oriented Design (OOD)
- Abstraction
- Encapsulation
- Modularity
- Inheritance
- Object creation and referencing
- Dynamic dispatch
- Polymorphism
- Method overriding
- Method overloading
- The keyword this
- Inheritance classes in Java (specialization)
- Inheritance classes in Java (extension)
- Throwing exceptions
- Catching exceptions
- Interfaces
- Abstraction classes & methods
- Casting
  - Widening conversions
  - Narrowing conversions

Object Oriented Design (OOD)

Object oriented programming deals with classes and objects.

The main “actors” in the object-oriented paradigm are called objects.

Each object is an instance of a class or an object comes from a class.

Each class presents to the outside world a concise and consistent view of the objects that are instances of this class.

The class definition typically specifies the data fields, also known as instance variables, that an object contains, as well as the methods (operations) that an object can execute.

```java
/* DSA Week 3 Lab 1 */

public class Week3Lab1 {

   //create a method called addition with a parameter called num
   static void addition(int num){
      System.out.println(num);
   }

   static void multiplication(int num){
      System.out.println(num);
   }

   static void subtraction(int num){
      System.out.println(num);
   }


   public static void main(String []args) {

      //create local variable names and align values to each
      String name = "Jerome";
      char gender ='M';
      int x = 5;
      int y = 2;

      int year = 2025;
      int num1 = (2*5);
      int num2 = (2+5);
      int num3 = (x - y);

      System.out.println("My Name is " + name + " my gender is " + gender);
      System.out.println("It's " + year);
      addition(num2); //prints aditional
      multiplication(num1); //prints multiplication
      subtraction(num3); //prints subtraction

   }
}
```

OODP

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals of: Abstraction, Encapsulation & Modularity.

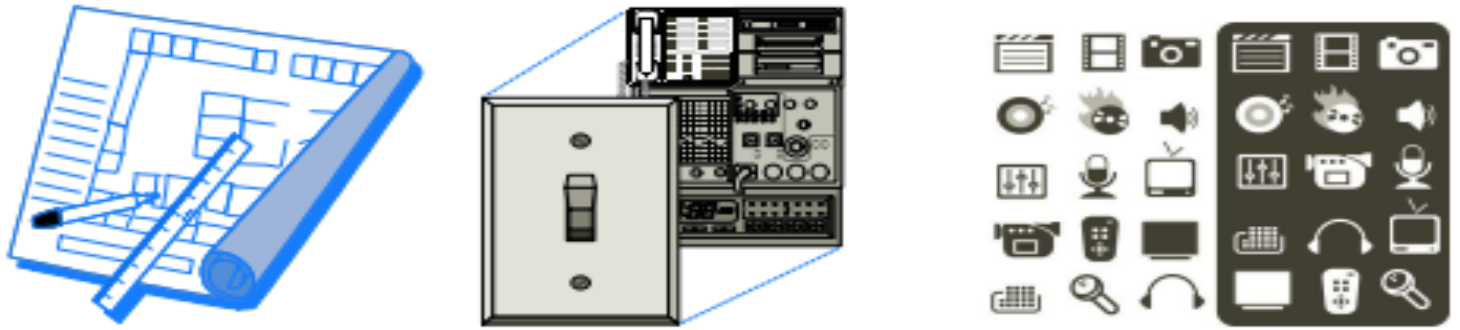


Figure 2.2: Principles of object-oriented design.

Abstraction

The notion of ***abstraction*** is to distil a complicated system down to its most fundamental parts.

Typically, describing the parts of a system involves naming them and explaining their functionality.

Applying the abstraction paradigm to the design of data structures gives rise to abstract data types (ADTs).

An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations.

An ADT specifies what each operation does, but not how it does it.

In Java, an ADT can be expressed by an interface, which is simply a list of method declarations, where each method has an empty body.

An ADT is realized by a concrete data structure, which is modelled in Java by a class. A class defines the data being stored and the operations supported by the objects that are instances of the class.

Also, unlike interfaces, classes specify how the operations are performed in the body of each method. A Java class is said to implement an interface if its methods include all the methods declared in the interface, thus providing a body for them. However, a class can have more methods than those of the interface.

Encapsulation

Another important principle of object-oriented design is encapsulation; different components of a software system should not reveal the internal details of their respective implementations.

One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions.

The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface.

Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Modularity

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized.

Modularity refers to an organizing principle in which different components of a software system are divided into separate functional units. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system.

Inheritance

OOD provides a **modular and hierarchical structure** for **reusing code through a technique** called **inheritance**.

This technique allows the design of general classes that can be specialized to more particular classes with the specialized classes reusing the code of the general class**. The general class (base class) or superclass can define standard instance variables and methods** that apply in a multitude of situations.

Inheritance

A **class that specializes or extends or inherits from a superclass need not give new implementations of the general methods for it inherits them**. It should only define those methods that are specialized for this particular subclass. **To make class a subclass of superclass we use the keyword extends.**

Subclass (child) & superclass (parent)

```java
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Illustration: The car class (subclass) inherits the attributes and methods from the Vehicle class

Object creation and Referencing

When an object ***O*** is created, memory is allocated for its data fields and these same fields are initialized to specific beginning values.

Typically, one associates the new object ***O*** with a variable which serves as a “link” to object ***O*** and is said to **reference *O***.

When accessing object ***O*** *we can either request execution of one of* ***O***’s methods or look up one of the fields of ***O***.

```
class ParentClass{
   public void show(){
      System.out.println("Parent class");
   }
}

class ChildClass extends ParentClass{

   public void show(){
      System.out.println("Child class");
   }

   public static void main(String[] args){
      ParentClass parentClass = new ParentClass();
      ChildClass childClass = new ChildClass();
      parentClass.show();
      childClass.show();
   }
}
```

From example: parentClass object is created that references the ParentClass;

Dynamic Dispatch

When a program wishes to invoke a certain method a() of some object ***O,*** it sends a message to ***O*** which is usually denoted, using the dot-operator syntax as ***O.a();***

```
class ParentClass{
   public void show(){
      System.out.println("Parent class");
   }
}

class ChildClass extends ParentClass{

   public void show(){
      System.out.println("Child class");
   }

   public static void main(String[] args){
      ParentClass parentClass = new ParentClass();
      ChildClass childClass = new ChildClass();
      parentClass.show();
      childClass.show();
   }
}
```

From example: invoke method show() using dot-operator to link objects parentClass and childClass.

Polymorphism

The word ***polymorphism*** literally means “many forms.”

In OOD, it refers to the ability of a reference variable to take different forms.

For example: the declaration of a variable having ParentClass as its type:

ParentClass parent;

Because this is a reference variable, the statement declares the new variable which does not yet refer to any card instance. While we have already seen that we can assign it to a newly constructed instance of the ParentClass class, Java also allows us to assign that variable to refer to an instance of the ChildClass subclass.

ParentClass parent = **new** ChildClass();

We say that the variable ***parent*** is polymorphic, it may take one of many forms depending on the specific class of the object to which it refers.

Method overriding Vs. Method overloading

METHOD OVERRIDING

This is a method of superclass that is re-defined in the subclass.

For example:

```
class s {
        public void a(){
        }
}
class t extends s {
        Public void a(){
        }
}
```

METHOD OVERLOADING

In method overloading, a method name is used by more than one method in the same class.

For example:

```
class T {
        public int a(){
        }
        Public void a (int x){
        }
}
```

The keyword this

Java provides a keyword called this to reference the current instance of a class. The reference this is useful in cases where we would like to pass the current object as a parameter to some method.
To reference a field inside current object that has a similar name with a variable defined in the current block of code.

For example:

```
public class Tester{
        public int dog = 2;
public void clobber(){

    int dog = 5;
    System.out.println("The dog local variable =  " + dog);
    System.out.println("The dog field = " + this.dog);
}
public static void main(String [] args){
    Tester t = new Tester();
    t.clobber();
}
}
```

```
----jGRASP exec: java Tester
The dog local variable =  5
The dog field = 2

----jGRASP: operation complete.
```

Java inheritance

There are two primary ways of using inheritance of classes in Java:

1. Specialization
2. Extension

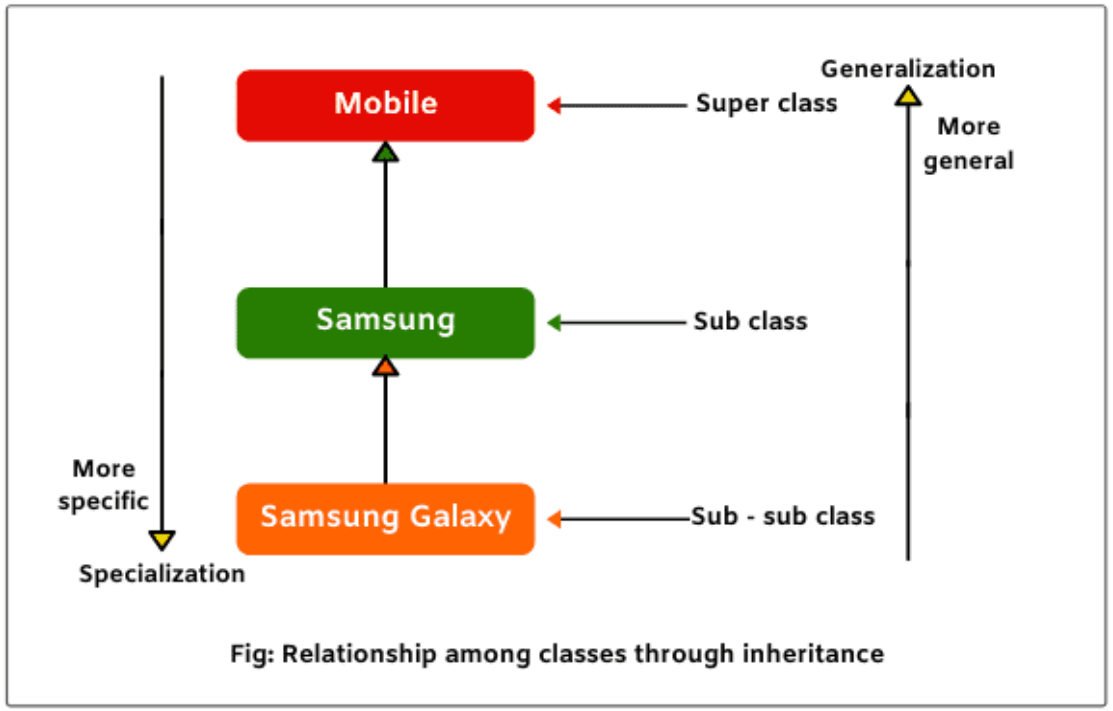
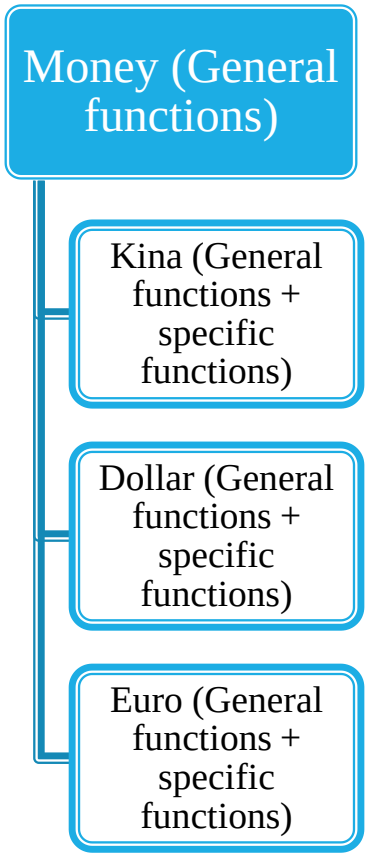


Fig: Relationship among classes through inheritance

Inheritance classes in Java (specialization)

In using specialization, we are specializing a general class to particular subclasses. Such subclasses typically possess an “is a” relationship to their superclass. A subclass then inherits all the methods of the superclass.

For each inherited method, if that method operates correctly independent of whether it is operating for a specification, no additional work is needed. If, on the other hand, a general method of the superclass would not work correctly on the subclass, then we should override the method to have the correct functionality for the subclass.



General class: Money, **Specialized class:** Kina, Dollar & Euro

Inheritance classes in Java (extension)

In extension, we utilize inheritance to reuse the code written for methods of the superclass, but we then add new methods that are not present in the superclass., so as to extend it's functionality.

In Java, each class can extend exactly one other class. Refer to the example in this slide.

```
class ParentClass{
   public void show(){
      System.out.println("Parent class");
   }
}

class ChildClass extends ParentClass{

   public void show(){
      System.out.println("Child class");
   }

   public static void main(String[] args){

      //create a parentClass object
      ParentClass parentClass = new ParentClass();
      //create a childClass object
      ChildClass childClass = new ChildClass();

      //call the show() method (from the Parent class) on the parentClass object
      parentClass.show();

      childClass.show();
   }
}
```

Exceptions

Exceptions are unexpected events that occur during the execution of a program.

An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.

In Java, exceptions are objects that can be thrown by code that encounters an unexpected situation, or by the Java Virtual Machine, for example, if running out of memory.

An exception may also be caught by a surrounding block of code that “handles” the problem in an appropriate fashion. If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.

In this section, we discuss common exception types in Java, as well as the syntax for throwing and catch exceptions within user-defined blocks of code.

Throwing exceptions

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Catching exceptions

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

Interfaces

In order for two objects to interact, they must “know” about the various messages that each will accept, that is, the methods each object supports.

To enforce this “knowledge,” the object-oriented design paradigm asks that classes specify the application programming interface (API), or simply interface, that their objects present to other objects.

An interface defining an ADT is specified as a type definition and a collection of methods for this type, with the arguments for each method being of specified types.

The main structural element in Java that enforces an API is an interface. An interface is a collection of method declarations with no data and no bodies. That is, the methods of an interface are always empty; they are simply method signatures. Interfaces do not have constructors and they cannot be directly instantiated. When a class implements an interface, it must implement all of the methods declared in the interface. In this way, interfaces enforce requirements that an implementing class has methods with certain specified signature

Abstraction classes & methods

In Java, an abstract class serves a role somewhat between that of a traditional class and that of an interface. Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies; such methods are known as abstract methods. However, unlike an interface, an abstract class may define one or more fields and any number of methods with implementation (so-called concrete methods). An abstract class may also extend another class and be extended by further subclasses.

Casting

In this section, we discuss casting among reference variables, as well as a technique, called generics, that allows us to define methods and classes that work with a variety of data types without the need for explicit casting.

Widening conversions

A widening conversion occurs when a type T is converted into a “wider” type U. The following are common cases of widening conversions:

- T and U are class types and U is a superclass of T.
- T and U are interface types and U is a superinterface of T.
- T is a class that implements interface U.

Narrowing conversions

A narrowing conversion occurs when a type T is converted into a “narrower” type S. The following are common cases of narrowing conversions:

- T and S are class types and S is a subclass of T.
- T and S are interface types and S is a subinterface of T.
- T is an interface implemented by class S.

Topic next week

Week 5 Arrays

DSA Week 4 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
  - Refer to the print out, answer all week 4 questions.
  - Refer to the print out, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 1, test 2, Major Assignment, Mid Semester Exam & Final Exam.