15-440
Distributed Systems
Recitation 1

Tamim Jabban
Office Hours

Office 1004

Sunday, Tuesday: 9:30 - 11:59 AM

Appointment: send an e-mail

Open door policy
Java: Object Oriented Programming

• A programming paradigm based on objects
• An example of an Object template:

```java
public class Student {
}
```
Java: Object Oriented Programming

• A programming paradigm based on objects
• An Object can contain data/attributes:

```java
public class Student {
    String name;
    int age;
    ...
}
```
Java: Object Oriented Programming

• A programming paradigm based on objects
• An Object can contain methods (behavior):

```java
public class Student {
    ...
    String name;
    public String getName() {
        return name;
    }
}
```
Java: Object Oriented Programming

- A programming paradigm based on objects
- To create a Student Object:

   `Student Ahmad = new Student();`
Constructors

• Constructors take in zero or more variables to create an Object:

```java
public class Student {
    String name;
    int age;
    public Student() {
    }
}

Student Ahmad = new Student();
```
Constructors

• Constructors take in zero or more variables to create an Object:

```java
public class Student {
    String name;
    int age;
    public Student(String name, int sAge) {
        this.name = name;
        age = sAge;
    }
}

Student Ahmad = new Student("Ahmad","21");
```
Inheritance

• Enables one object to inherit methods (behavior) and attributes from another object.

• For example, an Alumni class can extend a Student class:

```java
public class Alumni extends Student {
    int graduationYear;
    String jobTitle;
}
```

• Alumni inherits name, age & getName from Student.
Class Hierarchy

- This introduces **subclasses** and **superclasses**.
- A class that *inherits* from another class is called a **subclass**:
  - Alumni *inherits* from Student, and therefore Alumni is a **subclass**.
- The class that is *inherited* is called a **superclass**:
  - Student is *inherited*, and is the **superclass**.
Inheritance

• Organizes related classes in a hierarchy:
  • This allows reusability and extensibility of common code

• Subclasses extend the functionality of a superclass

• Subclasses inherit all the methods of the superclass (excluding constructors and privates)

• Subclasses can override methods from the superclass (more on this later)
Java Workspace Hierarchy

- **Project name**: BMW
- **Package name**: BMW
- **Class name**: BMW_i8

```java
package BMW;

public class BMW_i8 {
}
```
Access Control

Access modifiers describe the accessibility (scope) of data like:

- Attributes:
  ```java
  public String name;
  ```

- Methods:
  ```java
  protected String getName() { ... }
  ```

- Constructors
  ```java
  private Student(String name, int sAge) { ... }
  ```
Access Control

• Access modifiers include:
  • Public
  • Protected
  • Private
Access Control

• Access modifiers include:
  • Public
  • Protected
  • Private
Access Control

• Access modifiers include:

  • Public:

    Allows the access of the object/attributes/methods from any other program that is using this object:

    ```java
    public class Student {
        ...
        public void setName(String newName) {
            this.name = newName;
        }
    }
    ```

    ```java
    public class Test {
        public static void main(String[] args) {
            Student Ahmad = new Student();
            Ahmad.setName("Ahmad");
        }
    }
    ```
Access Control

- Access modifiers include:
  - Public
  - Protected
  - Private
Access Control

• Access modifiers include:

  • **Protected**:
    • You can use this only in the following
      • Same class as the variable,
      • Any subclasses of that class,
      • Or classes in the same **package**.
    • A **package** is a group of related classes that serve a common purpose.
Access Control

• Access modifiers include:
  • Public
  • Protected
  • Private
Access Control

- Access modifiers include:
  - Private:
    Restricted even further than a protected variable: you can use it **only in the same class**:

```java
public class Student {
    private void setName(String newName) {
        this.name = newName;
    }
    public Student(String name) {
        setName(name);
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Student Ahmad = new Student();
        Ahmad.setName("Ahmad");  // Not accessible anymore!
    }
}
```
Object & Class Variables

• Each Student object has its own name, age, etc...
  • name and age are examples of Object Variables.

• When an attribute should describe an entire class of objects instead of a specific object, we use Class Variables (or Static Variables).
Object & Class Variables

• A Class Variable Example:

```java
public class Student {
    public static String University = "CMU";
}
```

```java
public class Test() {
    public static void main(String[] args) {
        Student Ahmad = new Student();
        String uni = Ahmad.University;
    }
}
```
Object & Class Variables

• A Class Variable Example:

```java
public class Student {
    public static String University = "CMU";
}
```

```java
public class Test() {
    public static void main(String[] args) {
        String uni = Student.University;
    }
}
```
Encapsulation

- Encapsulation is restricting access to an object’s components.
- How can we change or access name now?:

```java
public class Student {
    private String name;
    private int age;
}

Student Ahmad = new Student();
```
Encapsulation

• Encapsulation is **restricting access to an object’s components**.
• Using **getters** and **setters**:
  ```java
  public class Student {
      private String name;
      private int age;
      public void setName(String newName) {
          this.name = newName;
      }
  }
  
  Student Ahmad = new Student();
  Ahmad.setName("Ahmad");
  ```
Overloading Methods

• Methods overload one another when they have the same method name but:
  • The number of parameters is different for the methods
  • The parameter types are different

• Example:
  
  public void changeDate(int year) {
      // process date change
  }

  public void changeDate(int year, int month) {
      // process date change
  }
Overloading Methods

• Methods overload one another when they have same method name but:
  • The number of parameters is different for the methods
  • The parameter types are different

• Another Example:
  ```java
  public void addSemesterGPA(float newGPA) {
    // process newGPA
  }
  
  public void addSemesterGPA(double newGPA) {
    // process newGPA
  }
  ```
Overloading Methods

• Methods overload one another when they have same method name but:
  • The number of parameters is different for the methods
  • The parameter types are different

• Another Example:
  ```java
  public void changeDate(int year) {
    // process date change
  }
  
  public void changeDate(int month) {
    // process date change
  }
  ```
Overloading Methods

• Methods overload one another when they have same method name but:
  • The number of parameters is different for the methods
  • The parameter types are different

• Another Example:

```java
public void changeDate(int year) {
    // process date change
}

public void changeDate(int month) {
    // process date change
}
```

We can’t overload methods by just changing the parameter name!
Overriding Methods

• Example:

```java
public class Parent {
    public int someMethod() {
        return 3;
    }
}

public class Child extends Parent {
    // this is method overriding:
    public int someMethod() {
        return 4;
    }
}
```
Overriding Methods

- Any class extends the Java superclass "Object".
- The Java "Object" class has 3 important methods:
  - `public boolean equals(Object obj);`
  - `public int hashCode();`
  - `public String toString();`
- The `hashCode` is just a number that is generated by any object:
  - It shouldn’t be used to compare two objects!
  - Instead, **override** the equals, hashCode, and toString methods.
Overriding Methods

• Example: Overriding the `toString` and `equals` methods in our `Student` class:

```java
public class Student {
    ...
    public String toString() {
        return this.name;
    }
}
```
Overriding Methods

- Example: **Overriding** the `toString` and `equals` methods in our `Student` class:

```java
public class Student {
    ...
    public boolean equals(Object obj) {
        if (obj.getClass() != this.getClass())
            return false;
        else {
            Student s = (Student) obj;
            return (s.name == this.name);
        }
    }
}
```
Abstract Classes

• A class that is **not completely implemented**.
• Contains one or more **abstract** methods (methods with no bodies; **only signatures**) that subclasses must implement
• Cannot be used to instantiate objects
• Abstract **class** header:
  
  ```java
  accessModifier abstract class className
  public abstract class Car
  ```
• Abstract **method** signature:
  
  ```java
  accessModifier abstract returnType methodName (args);
  public abstract int max_speed ();
  ```
• **Subclass** signature:
  
  ```java
  accessModifier class subclassName extends className
  public class Mercedes extends Car
  ```
Interfaces

- A **special abstract class** in which *all the methods are abstract*
- Contains only abstract methods that subclasses **must implement**
- Interface header:

  ```java
  accessModifier  interface  interfaceName
  public          interface  Car
  ```

- Abstract method signature:

  ```java
  accessModifier  abstract  returnType  methodName ( args );
  public          abstract  String      CarType    ( args );
  ```

- Subclass signature:

  ```java
  accessModifier  class  subclassName  implements  someInterface
  public          class  BMW          implements  Car
  ```
Generic Methods

• *Generic* or *parameterized* methods receive the data-type of elements as a parameter

• E.g.: a generic method for sorting elements in an array (be it *Integers, Doubles, Objects* etc.)
A Simple Box Class

• Consider this non-generic Box class:

```java
public class Box {
    private Object object;
    public void set(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
```
A Simple Box Class

• A *generic class* is defined with the following format:

```cpp
class my_generic_class <T1, T2, ..., Tn>
{
    /* ... */
}
```

Type parameters
A Simple Box Class

• Now to make our Box class generic:

```java
public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

• To create, for example, an Integer “Box”:

```java
Box<Integer> integerBox;
```
Java Generic Collections

• Classes that represent data-structures
• *Generic or parameterized* since the elements’ data-type is given as a parameter*
• E.g.: LinkedList, Queue, ArrayList, HashMap, Tree

• Provide methods for:
  • Iteration
  • Bulk operations
  • Conversion to/from arrays

*The data-type passed as parameter to a collection’s constructor can not be of the type Object, the unknown type ?, or a primitive data-type. The data-type must be a Class.*
Why Generic Functions?

• Consider writing a method that takes an array of objects, a collection, and puts all objects in the array into the collection

```java
static void fromArrayToCollection(Object[] arr, Collection<?> coll) {
    for (Object o : arr) {
        coll.add(o); // compile-time error
    }
}
```

```java
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```
ArrayList Class

• Is a subclass of Collection
• Implements a resizable array
• Provides methods for array manipulation
• Generic or parameterized
• Declaration and Instantiation:
  ```java
  ArrayList<ClassName> arrayListName = new ArrayList<ClassName>();
  ArrayList<Student> students = new ArrayList<Student>();
  ```
ArrayList Methods

• Add, Get, Set, Clear, Remove, Size, IsEmpty, Contains, IndexOf, LastIndexOf, AsList etc.

• Basic Iterator:

  ```java
  for (int i = 0; i < arrayListName.size(); i++) {
    // Get object at index i
    ClassName obj = arrayListName.get(i)
    // Process obj …
  }
  ```

• Advanced Iterator:

  ```java
  for (ClassName obj : arrayListName) {
    // Process obj
  }
  ```
Generic Classes with Wildcards

• Wildcards $?\rangle$ denote “unknown” or “any” type (resembles $<\text{T}>\rangle$)

```java
public void summAll(ArrayList<? extends Number> listOfNumbers) {}
```