Session 4 (of 24)

PGR112 Objektorientert programmering

Yuan Lin / yuan.lin@kristiania.no

Today’s Main Goal

- Go through Github directory
- A Quick Recap of Encapsulation
  - In Step 3 we have learned Getters/Setters
- Package
- Import
- Modifiers
- Enums
- Constructor overload
- ArrayList
- Practice Time

A Quick Recap

- **Encapsulation in Java** is a process of wrapping code and data together into a single unit.
  - By providing only a **Getter** or **Setter** method, you can make the class **read-only or write-only**
  - It is a way to achieve **data hiding** because other classes are not able to access private data members.
  - In order to access private data members of a class, other classes must use Getters/Setters
  - The standard IDE provides a way of facilitating generation of Getters/Setters

- To achieve encapsulation in Java
  - Declare the variables in a class as private
  - Provide public Getter/Setter methods to view and modify variables

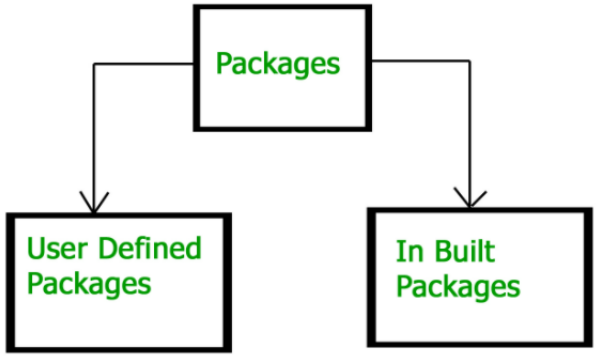
Java Packages

- Java package is used to encapsulate a group of classes.

- Java uses file system to store packages.

- Why use packages?
  - Preventing naming conflicts. i.e., package1.classname, package2.classname
  - Providing access protection. Default modifier have package level access control.
  - Write a better maintainable code

- Java packages are divided into two categories:
  - Built-in packages (Packages from Java API library)
  - User-defined Packages (create your own packages)



Create a Package

- Many IDEs provide a way to create a package, or add a class to a Package.
  - Include a package command followed by name of the package as the first statement in java source file
  - Example: package myPackage;

- You can also use Java Command (without using any IDE) to create a package
  - javac –d . myClass.java (keep the package within the same directory)
  - javac –d full_dir_path myClass.java (appoint specific directory)

Import

- There are three different ways of importing a class defined in a different package
  - Import the whole package

    import packageName.*
  - Import a specific class

    import packageName.className
  - Use fully qualified name of class

- What is a fully qualified class name?
  - Here pack.A is a fully qualified name for class A.

```java
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
class B {
    public static void main(String args[]) {
        pack.A obj = new pack.A();  //using fully qualified
        obj.msg();
    }
}
```

Modifier

- What is modifier?
  - We have learned that to achieve encapsulation, declare the variables in a class as private, and provide public Getter/Setter methods to view and modify variables
  - Private and Public are Java Access Modifiers
- Modifiers in Java are divided into Access Modifiers and Non-Access Modifiers
- Access Modifiers are used to control access mechanisms
  - There are 4: private, public, protected and default
  - For classes, you can use either public or default
  - For attributes, methods, or constructors, you can use all 4.
- Non-Access Modifiers provide information about their behaviors to JVM
  - There are 7: static, final, abstract, synchronized, transient, volatile, native
  - In this course you will learn static, final and abstract

Access Modifiers

- For classes
  - Public: The class is accessible by any other class
  - Default: The class is only accessible by classes in the same package. This is used when you don't specify a modifier. **You will learn more about packages later in this lecture**

- For attributes, methods and constructors
  - Public: The code is accessible for all classes
  - Private: The code is only accessible within the declared class
  - Protected: The code is accessible in the same package and outside the package through child classes
  - Default: The code is only accessible in the same package.

Access Modifiers

From Javatpoint

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Protected

- The protected modifier can be used to declare Fields, Methods and Constructors.
- To access protected fields, methods and constructors, you can
  - Access from any members of the same package
  - Access from subclasses from a different package
- Trying to access protected fields, methods and constructors from classes (not subclasses) of another package will produce compilation errors.

Protected

- Example
  - Protected variable
  - Protected method
  - Protected constructor

```java
/**
 * protected fields
 */
protected String ssn;

/**
 *    protected Getter/Setters for ssn
 */

protected String getSsn() {
    return ssn;
}
protected Person(String ssn) {
    this.ssn = ssn;
}
```

Non-Access Modifiers

- For classes
  - Final: The class cannot be inherited. **You will learn more when we talk about Inheritance**
  - Abstract: The class cannot be used to instantiate objects. To access it, it must be inherited from another class. **You will learn more when we talk about inheritance and abstraction**
  - Static: Java supports Static Class, but only in defining nested classes.

- For attributes and methods
  - Final: Attributes and methods cannot be overridden
  - Abstract: The method does not have a body. The body is provided by the child class. Abstract attribute and method can only be used in an abstract class. **You will learn more when we talk about inheritance and abstraction**
  - Static: Attributes and methods belongs to the class, rather than an object.

Static

- Static is applicable for Blocks, Variables, Methods, Classes
- When a member is declared static, it can be accessed before any objects of its class are created, or without reference to any object.
- Static variables:
  - All instances of the class share the same static variable. Static variable can be accessed by non-static method.
  - In Java, static variables can only be created at class level. Not allowed in local methods.
- Static methods:
  - They can only directly call other static methods
  - They can only directly access static variables
  - They cannot refer to **this** – why?

Static

- Example
  - Static variable
  - Static Getters/Setters
  - Static method

```java
/**
 * static variables and Getters/Setters
 */
static String subject;
public static String getSubject() {
    return subject;
}
public static void setSubject(String subject) {
    Teacher.subject = subject;
}
/**
 * static method cannot access non-static variables
 * To access non-static variables, you need to create an object instance
 */
public static void printTeacherSubject() {
    Teacher teacher = new Teacher();
    teacher.setSubject("IT Technology");
    System.out.println("Teacher subject:"+teacher.subject);
}
```

Java Enums

- What is Enum?
  - Enum is a constant
  - The major purpose of enum is to define our own data type
  - In Java, we can add variables, methods and even constructors to enum.
- All enums implicitly extend java.lang.

- Declaration of Enum
  - Enum declaration can be done outside a class or inside a class ut not inside a method

- Enum constant is implicitly **public static final**
  - Static means that we can access it by using enum Name
  - Final means we can not create child enums

Java Enums

- Enum can be implemented as a class
- We can define a main method inside enums
- Enum can contain a constructor
  - The constructor executes separately for each enum constant at enum class loading time
- Enum can contain methods
  - Enum can contain abstract method, but in such case the abstract method must be implemented by each instance of the enum class.
- Enum has values(), ordinal() and valueOf() methods
  - values() returns all values present inside enum
  - ordinal() can be used to find each enum constant index, just as an array index
  - valueOf() returns the enum constant of the specified string value, **if exists**
  - valueOf() will cause IllegalArgumentException if enum constant of the specified value does not exist

Java Enums

```java
public enum Genre {
    CRIME( name: "CRIME"), ACTION( name: "ACTION"), FANTASY( name: "FANTASY"),
    CLASSIC( name: "CLASSIC"), OTHER( name: "OTHER");
    private String name;
    /**
     * Enum can contain a construtor
     * @param name
     */
    Genre(String name) { this.name = name; }
    /**
     * Enum can contain methods
     * @return
     */
    public String getModifiedName() {
        return "Category-"+this.name;
    }
}
```

Constructor Overloading

- Constructor overloading means that we can write more than one constructor for a class
  - Each constructor can have different parameters
  - Constructor can be default or parametrized
- Why do we need constructor overloading?
  - So that each constructor can perform a different task when we initiate an object
- If we do not specify any constructor, java will invoke a default constructor (non-parametrized)
  - But if we specify a constructor, the default constructor is omitted.

Constructor Overloading

```java
public Book(String title, String author, int pages) {
    this.title = title;
    this.author = author;
    if (pages > 0) {
        this.numberOfPages = pages;
    }
    this.genre = Genre.OTHER;
}

public Book(String title, String author, int pages, Genre genre) {
    this(title, author, pages);
    this.genre = genre;
}
```

The Role of This in Constructor Overloading

- We can use **this** keyword in constructor overloading
  - this() is used to invoke the default constructor.
  - this(…args) is used to invoke the parametrized constructor

- Note that we use **this()** method as the first statement inside a constructor

A Recap of *this* keyword

- The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name.

- Example:

Field and parameter have the same name…

```
private String name;  //Field

public void setName(String name) { #what shall we do?}  // Setter-method
```

- We use this to refer to the object itself:

This object's name attribute is set equal to the value of the name parameter in Setter method.

```
private String name;

public void setName(String name) { this.name = name;}
```

ArrayList

- In Step 3, we used an array to hold the books in BookRegister class.
- The disadvantage of arrays is that they have a pre-defined size
- By using ArrayList we can remove this limitation. We can use ArrayList to
  - Insert objects
  - Get objects
  - Travel through
  - Remove objects

Create an ArrayList

- Below code shows how to create an ArrayList (example from w3schools):

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Clarify type: MUST

Not a MUST

ArrayList

```
ArrayList<Chapter> chapters1 = new ArrayList<>();
Chapter chapter1 = new Chapter( title: "chapter1",  pages: 20);
Chapter chapter2 = new Chapter( title: "chapter2",  pages: 50);
chapters1.add(chapter1);
chapters1.add(chapter2);
```

```
ArrayList<Book> books = bookRegister.GetRegisteredBooksByGenre(Genre.CRIME);
for(Book book: books) {
    book.printState();
}
```

Before we end

- Goals for this session:
  - I can import/create a package
  - I know how to use most commonly applied Modifiers
  - I understand Protected & Static
  - I know how constructors overloading works
  - I can use Java Enums
  - I can use both Array[] and ArrayList

Remember, do not just read code, play with it.
Good luck with the tasks!

Remember, there is help available all week, use Mattermost or GitHub.