Programming in Java

Dr. Nyein Aye Maung Maung
Dr. Eng (Ritsumeikan University, Japan)
Lecturer
Computer Engineering and Information Technology Dept.
Yangon Technological University

Course Schedule

Week	Topics
Week 1	Overview of JAVA, Data Types, Variables and Arrays
Week 2	Operators and Control Statements
Week 3	Classes and A closer look at Methods and Classes
Week 4	Inheritance, Polymorphism, Abstraction and Encapsulation
Week 5	Packages and Interfaces
Week 6	Exception Handling and Multi-threaded Programming
Week 7	String Handling
Week 8	Exploring java.lang and More utilities classes
Week 9	Java Collections Framework
Week 10	Java I/O
Week 11	Basic Graphical User Interface
Week 12	Event Handling
Week 13	Database Programming
Week 14	Applet and Networking

Lecture 5

Packages and Interfaces

Outline of Class (Lecture 5)

- Packages
- Interfaces

Lecture Objectives

- To simplify structuring the program using packages and interfaces
- To make searching and locating of classes with the help of packages
- To understand various access modifiers
- To define different access controls to the classes
- To define interfaces and define classes that implement interfaces

Topic 1

Packages

Packages

- Packages in Java are a way to encapsulate interfaces, Java classes and also subclasses. They are used for the following tasks –
  - Java Packages are used **to prevent the naming conflicts** which can occur between the classes.
  - Packages in Java make the **searching and locating** of classes or enumerations or annotations **much easier**.
  - Java Packages **provide access control** to the classes.
  - Packages in Java are used for **data encapsulation**.

Packages

- **Defining a package**
  - Choose a name for the package and simply include a package command as the first statement in a Java source file.

    package nameofpackage;

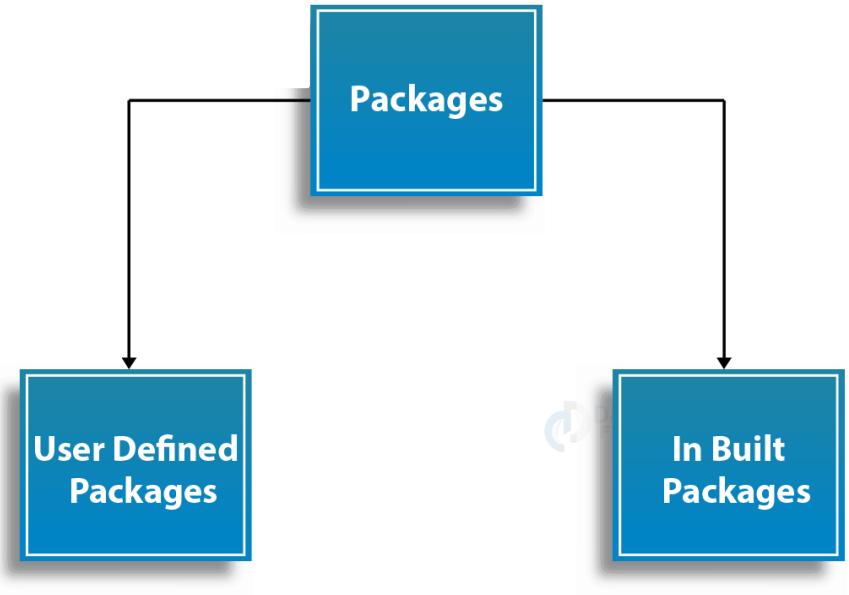
  - More than one file can include the same package statement.
  - A hierarchy of packages can also be created.
    - The general form of a multileveled package statement is shown here:

    package pkg1[.pkg2[.pkg3]];

    eg. java.util.Scanner;

Types of Packages in Java

- **Built-in**
  - Built-in Java packages are a part of Java API
    - **lang –** Automatically imported and it contains language support classes.
    - **io –** Contains classes for input and output operations.
    - **util –** Contains utility classes for implementing data structures.
    - **applet –** This package contains classes that create applets.
    - **awt –** Contain classes that implement compounds for GUI.
    - **net –** This package contains classes that support networking operations.
- **User Defined**
  - Java User creates these Java packages



Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
  - By default, the Java run-time system uses the current working directory as its starting point.
    - Thus, if your package is in a subdirectory of the current directory, it will be found.
  - Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example,

package MyPack

path to MyPack is

C:\MyPrograms\Java\MyPack

Then the class path to MyPack is

C:\MyPrograms\Java

Example 5.1. Design a class named Balance which has one string data field called 'name' and one double field called 'bal', two argument constructor tha t creates the Balance object with specified name and balance, one method na med 'show' to display the balance. Write a test program to create three balan ce objects and display their current balances. Create a 'MyPack' package and include the test program inside it.

```
package MyPack;
class Balance {
        String name;
        double bal;
        Balance(String n, double b) {
        name = n;
        bal = b;
        }
        void show() {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
}
}
```

```
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 1
23.23);
current[1] = new Balance("Will Tell", 157.02
);
current[2] = new Balance("Tom Jackson", -1
2.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Importing Packages

- Java includes the import statement to bring certain classes, or entire packages, into visibility.

  import pkg1[.pkg2].(classname|*);

  pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

- Once imported, a class can be referred to directly, using only its name.

  import package → import java.util.Scanner;

  Scanner input=new Scanner(System.in)

  Without import → java.util.Scanner input=new java.util.Scanner(System.in);

- Import statements occur immediately following the package statement (if it exists) and **before any class definitions**.

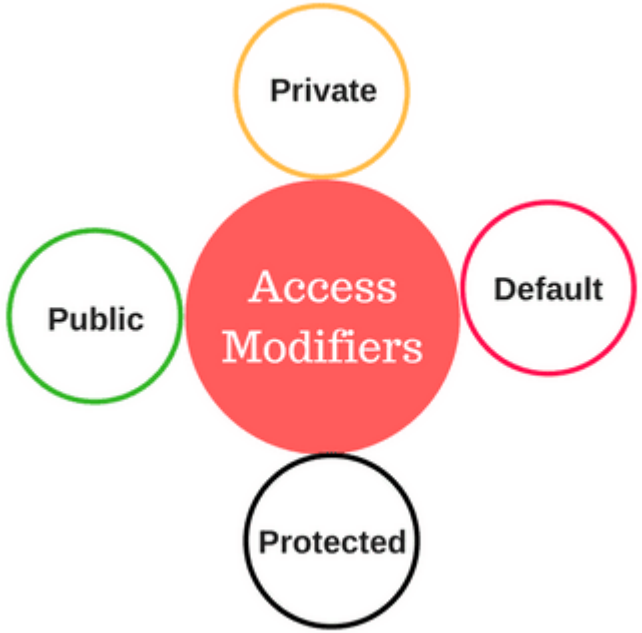
Example 5.2. An example demonstrating how to access class from different package.

```
package MyPack;
public class Balance {
        String name;
        double bal;
        public Balance(String n, double b) {
        name = n;
        bal = b;
        }
        public void show() {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
        }
}
```

```
package Pack2;
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may
use Balance class and call its
constructor. */
Balance test = new Balance("J. J.
Jaspers", 99.88);
test.show();}
}
```

Access Modifiers

- Modifiers control access to data fields, methods, and classes
- There are four types:
  - Default
  - Private
  - Protected
  - Public



Access Modifiers

- Default
  - Used to define access when no other modifier is specified for the class or function
  - With this modifier, they are accessible only within the same package
- Private (Keyword 'private')
  - Used to define access to a Java datatypes or method
  - Only accessible in the same class, i.e. any other class of the same package won't be able to access it.
  - Classes cannot be defined with this access type.
- Protected (Keyword 'protected')
  - Can be accessed within the same package or subclass of a different package
- Public (Keyword 'public')
  - There is no restriction, i.e. Java datatypes or methods can be accessed from anywhere in the program

Access Protection Rules

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection Rules

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

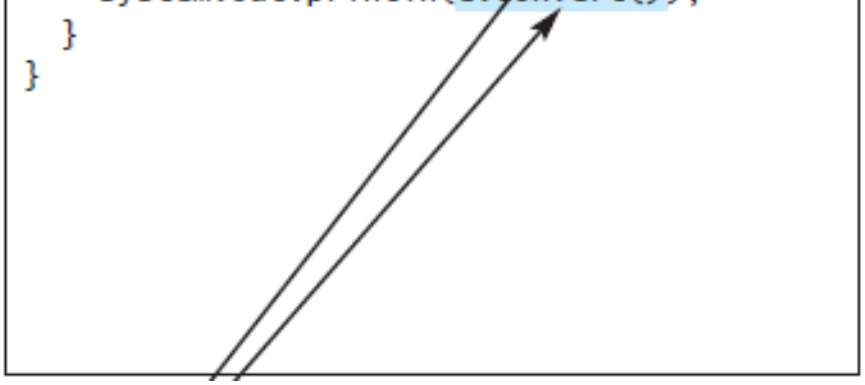
```java
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

This is okay because object **c** is used inside the class **C**.

```java
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```



This is wrong because **x** and **convert** are private in class **C**.

Example 5.3. This program demonstrates the difference between public and private

```
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
ob.a = 10;
ob.b = 20;
// ob.c = 100; // Error! Private Data
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
}
}
```

Quiz on Access Protection

Examine at which lines error may occur.

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub)
;
}
}
```

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package construc
tor");
System.out.println("n = " + p.n);
System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other
package constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package construc
tor");
System.out.println("n = " + p.n);
System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Topic 2

Interfaces

Interfaces

- An **interface in java** is a blueprint of a class.
- It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
    - The class that implements interface must implement all the methods of that interface

Interfaces

- An interface is similar to a class in the following ways:
  - An interface can contain any number of methods.
  - An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
  - The byte code of an interface appears in a .class file.
  - Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
- However, an interface is different from a class in several ways, including:
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - One class can implement multiple interfaces.
- **Interface allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.**

Defining an Interface

```
interface InterfaceName
{
    /* All the methods are public abstract by default
     * Note down that these methods are not having body
     */
  //Any number of final, static fields
 //Any number of abstract method declarations
    public void method1();
    public void method2();
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

Example 5.4. Simple Program implementing Interface

```
interface Callback {
void callback(int param);
}
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

Output
callback called with 42

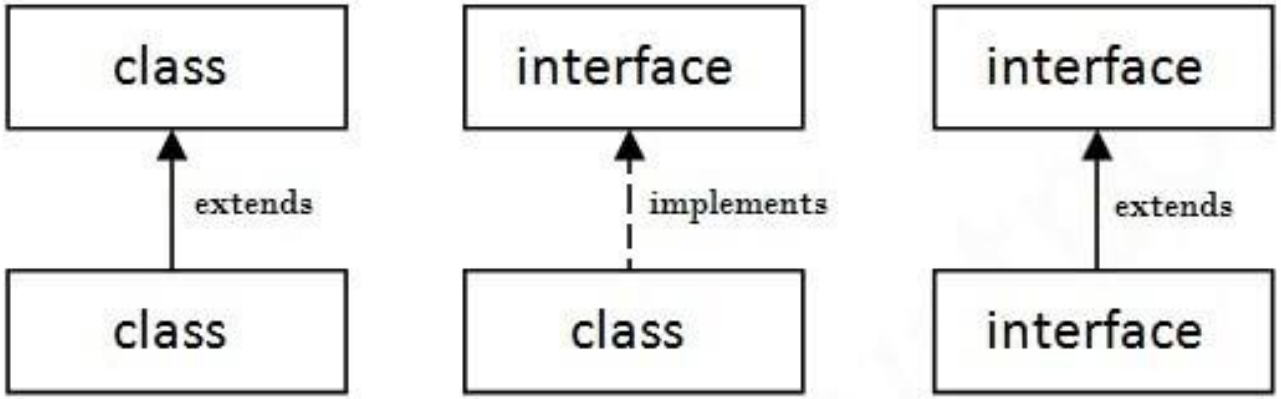
Example 5.5. Another implementation of Callback Interface

```
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
 class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

Output

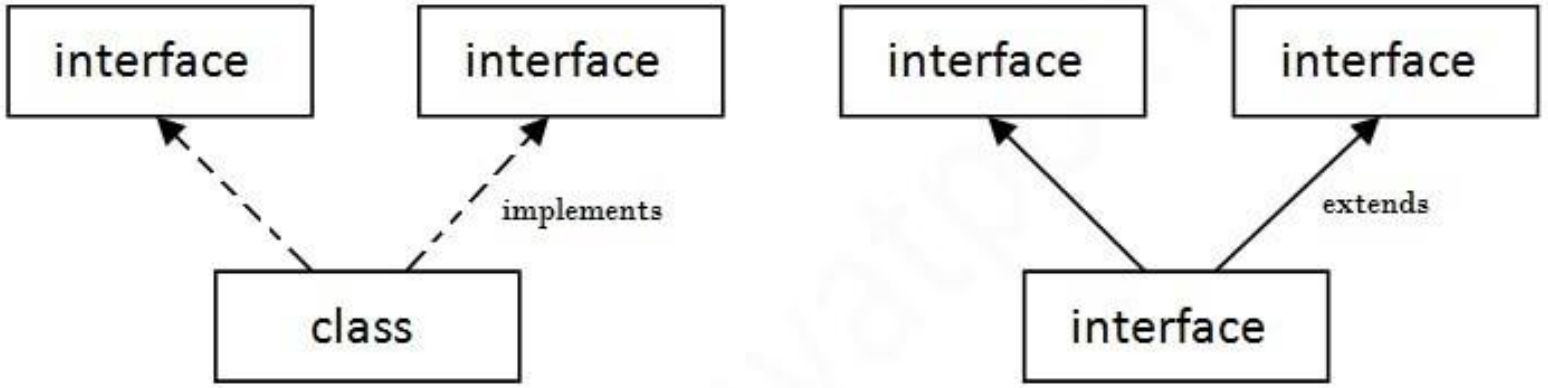
```
callback called with 42
Another version of callback
p squared is 1764
```

Relationship between classes and interfaces



- A class extends another class
- An interface extends another interface,
- But a **class implements an interface**.

Multiple inheritance in Java by interface



- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Example 5.6. Multiple inheritance by Interfaces

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface implements Printable, Showable{
        public void print(){System.out.println("Hello");}
        public static void main(String args[]){
        TestInterface obj = new TestInterface();
        obj.print();
         }
}
```

Nested Interface

- An interface can have another interface which is known as a nested interface

```
interface printable{
        void print();
        interface MessagePrintable{
            void msg();
             }
}
```

Example 5.6. Demonstrate Nested Interface

```
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
```

```
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false : true;
}
}
```

```
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
}
}
```

Extending Interfaces

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
```

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Key Points of Interfaces

i. We can't instantiate an interface in java.

ii. Interface provides complete abstraction as none of its methods can have body. On the other hand, abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

iii. implements keyword is used by classes to implement an interface.

iv. While providing implementation in class of any method of an interface, it needs to be mentioned as public.

v. Class implementing any interface must implement all the methods, otherwise the class should be declared as "abstract".

vi. Interface cannot be declared as private or protected.

vii. All the interface methods are by default abstract and public.

viii. Variables declared in interface are public, static and final by default.

(ix) Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

```
interface Try
{
      int x; //Compile-time error
}
```

(x)Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final.

```
Class Sample implements Try
{
  public static void main(String arg[])
  {
     x=20; //compile time error
  }
}
```

Quiz on Interfaces

1. Which of the following is correct way of implementing an interface salary by class manager?
a) class manager extends salary {}
b) class manager implements salary {}
c) class manager imports salary {}
d) None of the mentioned.

2. What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue) {
        System.out.println("Hi Mom");
    }
}
```

3. Which of the following is a correct interface?

```
interface A {
  void print() { };
}
```

(a)

```
abstract interface A extends I1, I2 {
  abstract void print() { };
}
```

(b)

```
abstract interface A {
  print();
}
```

(c)

```
interface A {
  void print();
}
```

(d)

Practical Assignments

1. Design a class named **Account** that contains:
   - A private **int** data field named **id** for the account (default **0**).
   - A private **double** data field named **balance** for the account (default **0**).
   - A private **double** data field named **annualInterestRate** that stores the current interest rate (default **0**). Assume all accounts have the same interest rate.
   - A private **Date** data field named **dateCreated** that stores the date when the account was created.
   - A no-arg constructor that creates a default account.
   - A constructor that creates an account with the specified id and initial balance.
   - The accessor and mutator methods for **id**, **balance**, and **annualInterestRate**.
   - The accessor method for **dateCreated**.
   - A method named **getMonthlyInterestRate()** that returns the monthly interest rate.
   - A method named **getMonthlyInterest()** that returns the monthly interest
   - A method named **withdraw** that withdraws a specified amount from the account.
   - A method named **deposit** that deposits a specified amount to the account.

Draw the UML diagram for the class and then implement the class. (*Hint*: The method **getMonthlyInterest()** is to return monthly interest, not the interest rate.

Monthly interest is **balance * monthlyInterestRate**. **monthlyInterestRate** is **annualInterestRate / 12**. Note that **annualInterestRate** is a percentage,e.g.,like 4.5%. You need to divide it by 100.) Write a test program that creates an **Account** object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the **withdraw** method to withdraw $2,500, use the **deposit** method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

2. Design an interface named **Colorable** with a **void** method named **howToColor()**. Every class of a colorable object must implement the **Colorable** interface. Design a class named **Square** that extends **GeometricObject** and implements **Colorable**. Implement **howToColor** to display the message **Color all four sides**. Draw a UML diagram that involves **Colorable**, **Square**, and **GeometricObject**. Write a test program that creates an array of five **GeometricObject**s. For each object in the array, invoke its **howToColor** method if it is colorable.

Next Week Lecture

- **Exception Handling**
- **Multithreading**

Thank you!