ITSE424: Lec1 Programming in Java Review

Presented By Mai Elbaabaa

Degrees

- **50 percent**

35%	**Midterm**	**All topics covered until the midterm date.**
15%	**Assignments & project**	**Will explain the details later**

- **50 percent – final examination**

notes

- No postpones or make up midterm and final exam
- Assignments & project must be done and correctly turned in *on time* for full credit - no exceptions, no extensions, no excuses.

Virtual Classroom

- course materials, inquires or participations will be in classhub.

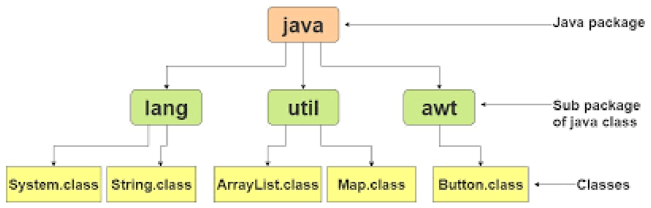
- **Books as reference**
  - Introduction to Java Programming, By: Y. Daniel Liang.
  - Gostling J., Java Language Specification, Addison-Wesley Professional, 2005.
  - Eckel B., Thinking in Java, Prentice Hall, 2006.
  - Schild H., Java 2: A Beginner's Guide, McGraw-Hill Professional, 2002.
- **Online tutorials**
  - https://www.tutorialspoint.com/java_technology_tutorials.htm
  - https://www.edureka.co/blog/advanced-java-tutorial/

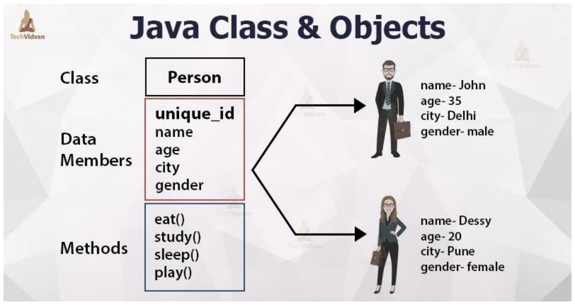
Course content

- Review on OOP principles & UML class diagrams.
- **Creational Patterns:** Factory, Abstract Factory, Builder, Singleton and Prototype design patterns.
- **Structural design patterns:** Adapter, Bridge, composition, decorator, façade, Flyweight and proxy design patterns
- **Behavioral Design Patterns:** Observer, Strategy and command design patterns.
- Anti Patterns.

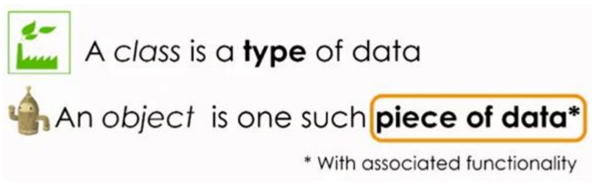
Packages, Classes, Methods…

- A package is a collection of related classes. Moreover, every package has a name.
- The term class is used to create Java programs; it is used to group a set of related operations; and it is used to allow users to create their own data types.
- method as a set of instructions designed to accomplish a specific task.





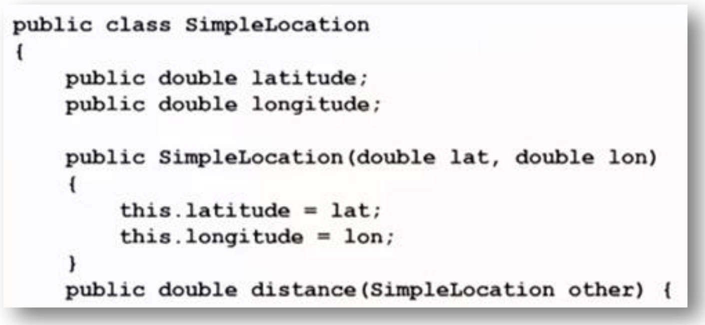
Class & Object


A class is a type of data
An object is one such piece of data*
* With associated functionality

Map
Shape
Location
Color
Window
… and plenty more objects

Example

- Lets say that I want to create a class to define my location in a map
- First what do I need to represent my location:
- Latitude and longitude coordinators.

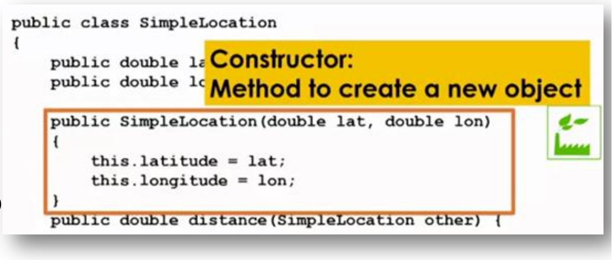


```
public class SimpleLocation
{
    public double latitude;
    public double longitude;

    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
    public double distance(SimpleLocation other) {
```

Constructor

- the constructor is a special method that gets called when my objects get created. So when I ask Java to give me a new object of type SimpleLocation, it's going to call this method here, which is called the constructor.
- And the reason you know it's the constructor, is that it doesn't have a return type, so it simply says public and then next word in the declaration of this method is just the name of the class.

- A constructor is automatically called on object creation
- A default constructor provided by the compiler initializes integer to 0, floating point numbers to 0.0, Boolean values to false and objects to null.



```
public class SimpleLocation
{
    public double la
    public double lo

    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
    public double distance(SimpleLocation other) {
```

Types of Variables

- There are three types of variables in Java:

1) Local Variable

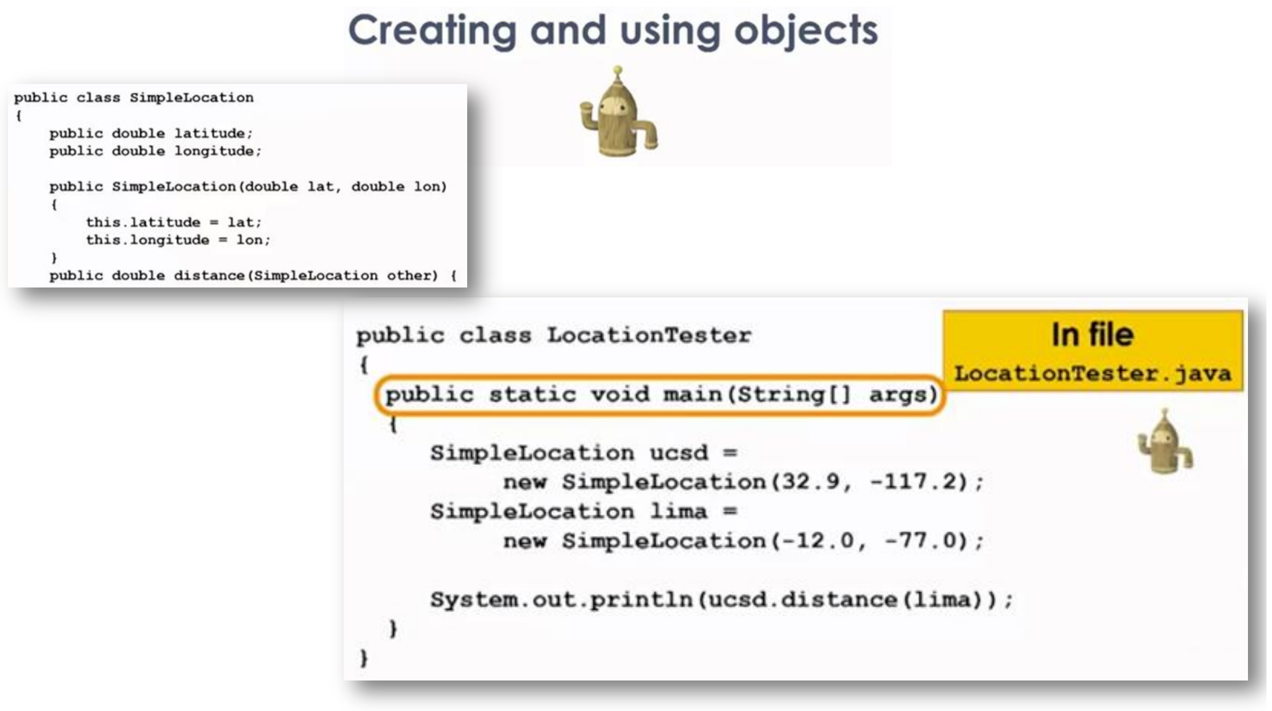
- A variable declared inside the body of the method is called local variable. You can use this variable only within that method

2) Instance Variable

- A variable declared inside the class but outside the body of the method,

3)Static variable

- A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class.

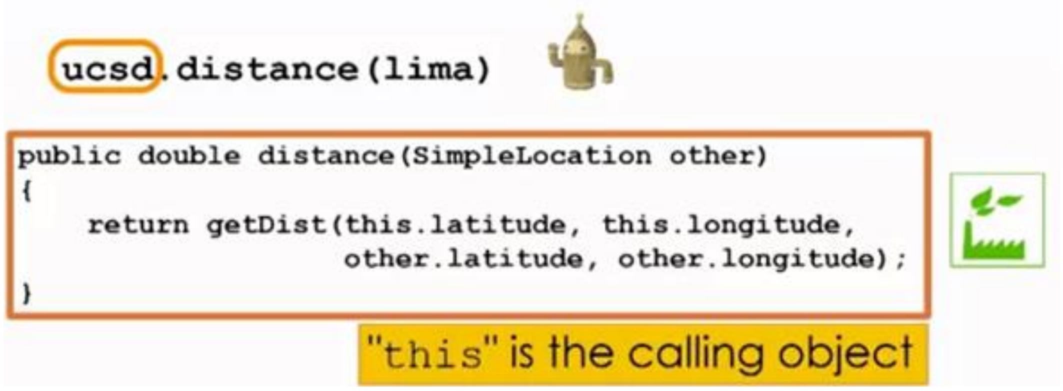


Creating and using objects

```
public class SimpleLocation
{
    public double latitude;
    public double longitude;

    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
    public double distance(SimpleLocation other) {
```

In file LocationTester.java

```
public class LocationTester
{
  public static void main(String[] args)
  {
     SimpleLocation ucsd =
           new SimpleLocation(32.9, -117.2);
     SimpleLocation lima =
           new SimpleLocation(-12.0, -77.0);

     System.out.println(ucsd.distance(lima));
  }
}
```

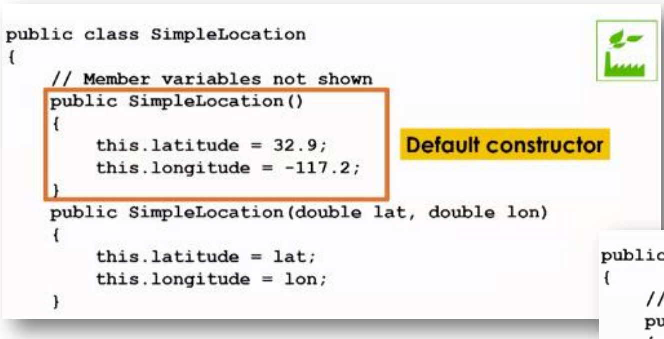


```
ucsd.distance(lima)

public double distance(SimpleLocation other)
{
    return getDist(this.latitude, this.longitude,
                   other.latitude, other.longitude);
}
```

"this" is the calling object

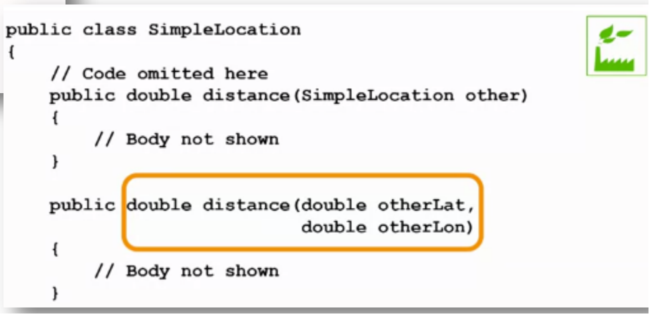
- We're comparing the distance between two SimpleLocation objects.
- the keyword, this refers to the calling object, which is the object that called the method, or on which the method was called. So in this line of code, ucsd dot distance and then pass in lima UCSD is called the calling object because it's the object that occurs before that dot.
- The result of running the code is 6567.659 KM.

Overloading and overloading



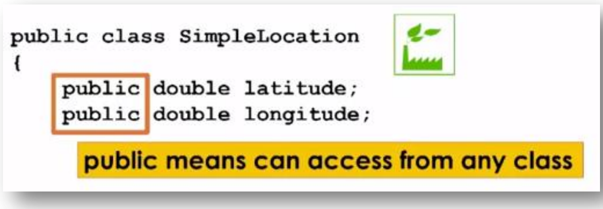
```
public class SimpleLocation
{
    // Member variables not shown
    public SimpleLocation()
    {
        this.latitude = 32.9;
        this.longitude = -117.2;
    }
    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
```

Default constructor



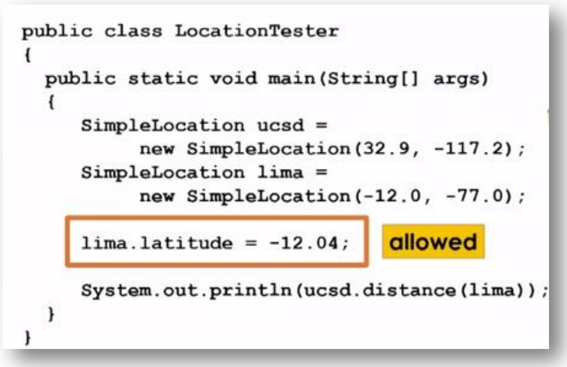
```
public class SimpleLocation
{
    // Code omitted here
    public double distance(SimpleLocation other)
    {
        // Body not shown
    }

    public double distance(double otherLat,
                           double otherLon)
    {
        // Body not shown
    }
```

Public vs. Private



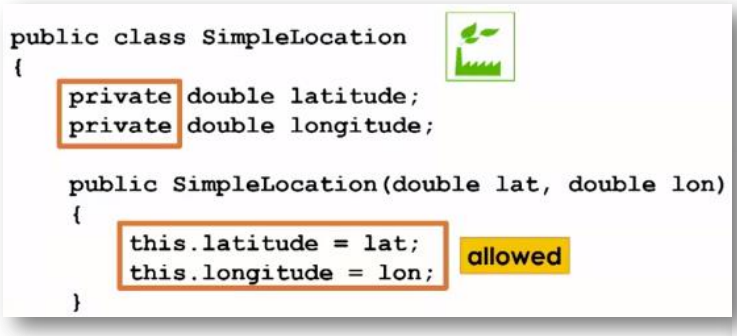
```
public class SimpleLocation
{
    public double latitude;
    public double longitude;
```

public means can access from any class



```
public class LocationTester
{
  public static void main(String[] args)
  {
     SimpleLocation ucsd =
          new SimpleLocation(32.9, -117.2);
     SimpleLocation lima =
          new SimpleLocation(-12.0, -77.0);

     lima.latitude = -12.04;

     System.out.println(ucsd.distance(lima));
  }
}
```

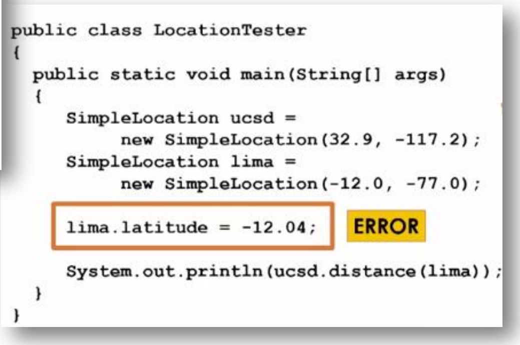
allowed



```
public class SimpleLocation
{
    private double latitude;
    private double longitude;

    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
```

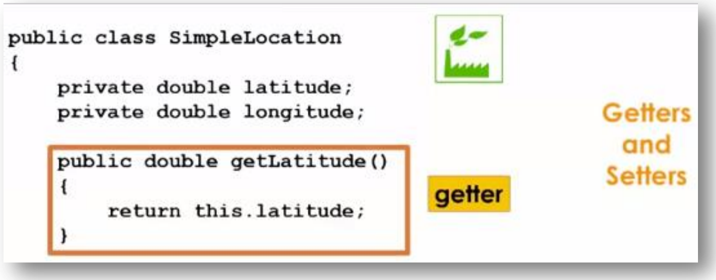
allowed

Rule of thumb: Make member variables private
(and methods either public or private)

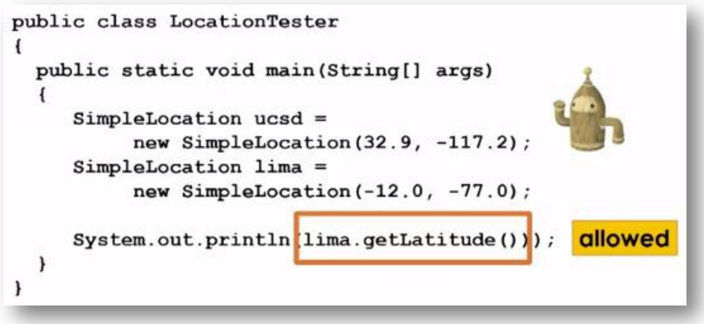


```
public class LocationTester
{
  public static void main(String[] args)
  {
     SimpleLocation ucsd =
          new SimpleLocation(32.9, -117.2);
     SimpleLocation lima =
          new SimpleLocation(-12.0, -77.0);

     lima.latitude = -12.04;

     System.out.println(ucsd.distance(lima));
  }
}
```

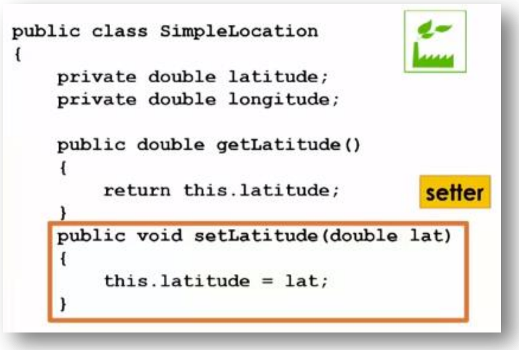
ERROR



```
public class SimpleLocation
{
    private double latitude;
    private double longitude;

    public double getLatitude()
    {
        return this.latitude;
    }
```



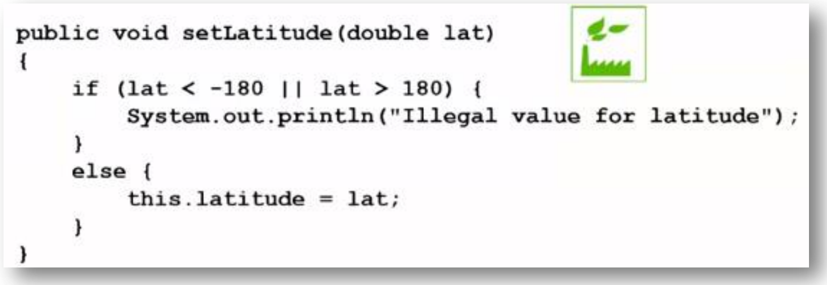
```
public class LocationTester
{
  public static void main(String[] args)
  {
     SimpleLocation ucsd =
          new SimpleLocation(32.9, -117.2);
     SimpleLocation lima =
          new SimpleLocation(-12.0, -77.0);

     System.out.println(lima.getLatitude());
  }
}
```



```
public class SimpleLocation
{
    private double latitude;
    private double longitude;

    public double getLatitude()
    {
        return this.latitude;
    }
    public void setLatitude(double lat)
    {
        this.latitude = lat;
    }
```

Why didn't we just make that member variable public?

- let's say we want to allow the user to change the value of the latitude. But we're little bit unsure that the user of our class will know what they're doing
- rather than just blindly accepting whatever argument was passed in, we would have some checks, some logic inside of that method that said okay, if it's out of range of -180 to 180 then that's not a legal value.



```
public void setLatitude(double lat)
{
     if (lat < -180 || lat > 180) {
          System.out.println("Illegal value for latitude");
     }
     else {
          this.latitude = lat;
     }
}
```

Main Tenets of OO Programming

- **Encapsulation**
  - abstraction, information hiding
- **Inheritance**
  - code reuse, specialization "New code using old code."
- **Polymorphism**
  - do X for a collection of various types of objects, where X is *different* depending on the type of object
  - "Old code using new code.“
- **Abstraction**

Things and Relationships

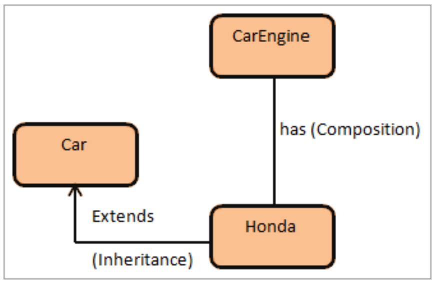
- Object oriented programming leads to programs that are models
  - sometimes models of things in the real world
  - sometimes models of contrived or imaginary things
- There are many types of relationships between the things in the models
  - chess piece has a position
  - chess piece has a color
  - chess piece moves (changes position)
  - chess piece is taken
  - a rook is a type of chess piece

The “has-A” Relationship

- Objects are often made up of many parts or have sub data.
  - chess piece: position, color
  - die: result, number of sides
- This “has-a” relationship is modeled by composition
  - the instance variables or fields internal to objects
- Encapsulation captures this concept

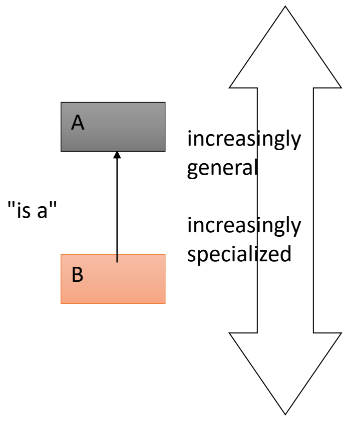
The “is-a” relationship

- Another type of relationship found in the real world
  - a rook is a chess piece
  - a queen is a chess piece
  - a student is a person
  - a faculty member is a person
  - an undergraduate student is a student
- “is-a” usually denotes some form of specialization
- it is not the same as “has-a”



Inheritance

- The "is-a" relationship, and the specialization that accompanies it, is modeled in object oriented languages via *inheritance*
- Classes can inherit from other classes
  - base inheritance in a program on the real world things being modeled
  - does "an A is a B" make sense? Is it logical?



21

Inheritance

- Person is said to be
  - the (parent, super, base, ancestor) class of Student
- Student is said to be
  - (a child, a sub, a derived, a descendant) class of Person
- the sub class inherits (gains) all instance variables and instance methods of the super class, **automatically**
- additional methods can be added to class B (specialization)
- the sub class can replace (redefine, override) methods from the super class

Polymorphism

- Another feature of OOP
- literally "having many forms"
- object variables in Java are polymorphic
- object variables can refer to objects or their declared type AND any objects that are descendants of the declared type

```
ClosedShape s = new ClosedShape();
s = new Rectangle(); // legal!
s = new Circle(); //legal!
Object obj1; // = what?
```

23

Abstract Classes & Interfaces

The abstract Method and abstract class

- An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method.

- For example, in the Shape class, we can declare abstract methods getArea(), draw(), etc, as follows:

```
abstract public class Shape {
   ......
   ......
   abstract public double getArea();
   abstract public double getPerimeter();
   abstract public void draw();
}
```

Abstract Class EG. 1: Shape and its Subclasses

- To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods.

The abstract Superclass Shape.java

```java
/*
 * This abstract superclass Shape contains an abstract method
 * getArea(), to be implemented by its subclasses.
 */
abstract public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

   // All Shape subclasses must implement a method called getArea()
   abstract public double getArea();
}
```

In summary

- an abstract class provides a template for further development. The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses. For example, in the abstract class Shape, you can define abstract methods such as getArea()

```java
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an abstract class
      Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

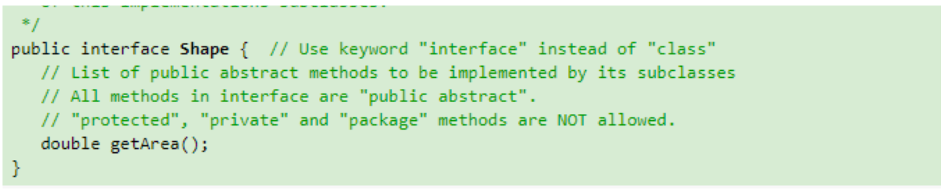
- An abstract method cannot be declared final, as final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendant before it can be used.
- An abstract method cannot be private (which generates a compilation error). This is because private method are not visible to the subclass and thus cannot be override.

What is interface in java

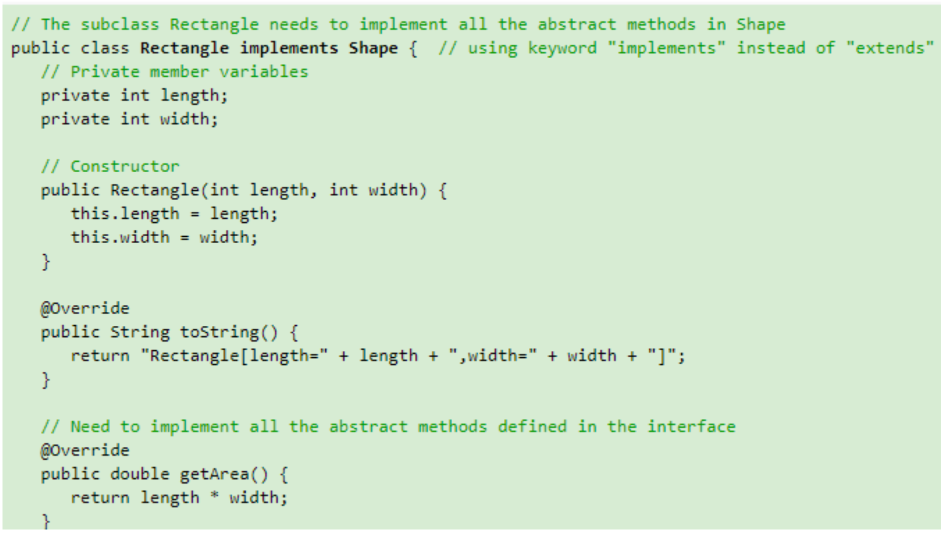
- Since multiple inheritance is not allowed in Java, interfaces only way to implement multiple inheritance at Type level
- A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support

Interface EG. 1: Shape Interface and its Implementations

We can re-write the `abstract` superclass Shape into an `interface`, containing only `abstract` methods, as follows:



```
 */
public interface Shape {  // Use keyword "interface" instead of "class"
   // List of public abstract methods to be implemented by its subclasses
   // All methods in interface are "public abstract".
   // "protected", "private" and "package" methods are NOT allowed.
   double getArea();
}
```



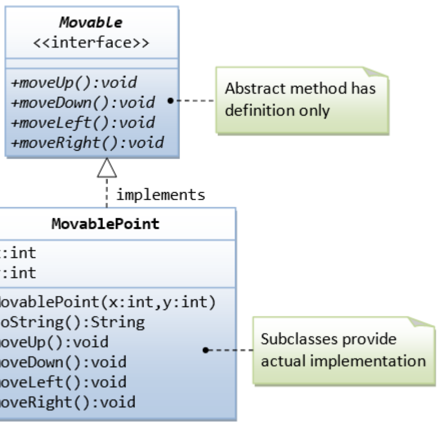
```
// The subclass Rectangle needs to implement all the abstract methods in Shape
public class Rectangle implements Shape {  // using keyword "implements" instead of "extends"
   // Private member variables
   private int length;
   private int width;

   // Constructor
   public Rectangle(int length, int width) {
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle[length=" + length + ",width=" + width + "]";
   }

   // Need to implement all the abstract methods defined in the interface
   @Override
   public double getArea() {
      return length * width;
   }
```

```
// The subclass Triangle need to implement all the abstract methods in Shape
public class Triangle implements Shape {
   // Private member variables
   private int base;
   private int height;

   // Constructor
   public Triangle(int base, int height) {
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle[base=" + base + ",height=" + height + "]";
   }

   // Need to implement all the abstract methods defined in the interface
   @Override
   public double getArea() {
      return 0.5 * base * height;
   }
}
```

A test driver is as follows:

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle(1, 2);  // upcast
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle(3, 4);  // upcast
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());
```

Interface EG. 2: `Movable` Interface and its Implementations.



- Suppose that our application involves many objects that can move. We could define an interface called `movable`, containing the signatures of the various movement methods

Interface `Moveable.java`

```
/*
 * The Movable interface defines a list of public abstract methods
 * to be implemented by its subclasses
 */
public interface Movable {  // use keyword "interface" (instead of "class") to define an interface
   // An interface defines a list of public abstract methods to be implemented by the subclasses
   public void moveUp();    // "public" and "abstract" optional
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

- MovablePoint.java
- To derive subclasses from an interface, a new keyboard "implements" is to be used instead of "extends" for deriving subclasses from an ordinary class or an abstract class. It is important to note that the subclass implementing an interface need to override ALL the abstract methods defined in the interface; otherwise, the subclass cannot be compiled. For example,

```java
public class MovablePoint implements Movable {
   // Private member variables
   private int x, y;   // (x, y) coordinates of the point

   // Constructor
   public MovablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   @Override
   public String toString() {
      return "(" + x + "," + y + ")";
   }

   // Need to implement all the abstract methods defined in the interface Movable
   @Override
   public void moveUp() {
      y--;
   }
   @Override
   public void moveDown() {
      y++;
   }
   @Override
   public void moveLeft() {
      x--;
   }
   @Override
   public void moveRight() {
      x++;
   }
}
```

- TestMovable.java
- We can also upcast subclass instances to the Movable interface, via polymorphism, similar to an abstract class.

```java
public class TestMovable {
   public static void main(String[] args) {
      MovablePoint p1 = new MovablePoint(1, 2);  // upcast
      System.out.println(p1);
      p1.moveDown();
      System.out.println(p1);
      p1.moveRight();
      System.out.println(p1);

      // Test Polymorphism
      Movable p2 = new MovablePoint(3, 4);  // upcast
      p2.moveUp();
      System.out.println(p2);
      MovablePoint p3 = (MovablePoint)p2;   // downcast
      System.out.println(p3);
   }
}
```

Implementing Multiple Interfaces

- As mentioned, Java supports only *single inheritance*. That is, a subclass can be derived from one and only one superclass. Java does not support *multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses. Multiple inheritance, however, does have its place in programming.
- A subclass, however, can implement more than one interfaces.
- In other words, Java indirectly supports multiple inheritances via implementing multiple interfaces. For example,

```
public class Circle extends Shape implements Movable, Adjustable {
          // extends one superclass but implements multiple interfaces
   .......
}
```

interface Formal Syntax

- The formal syntax for declaring interface is:

```
[public|protected|package] interface interfaceName
[extends superInterfaceName] {
   // constants
   static final ...;

   // public abstract methods' signature
   ...
}
```

All methods in an interface shall be public and abstract (default). You cannot use other access modifier such as private, protected and default, or modifiers such as static, final.
All fields shall be public, static and final (default).
An interface may "extends" from a super-interface.

When to use Interface

- Interface is best choice for Type declaration or defining contract between multiple parties In large projects, where many developers are involved, different people write different module (functionalities). In order to make sure that the code developed by one developer is easily integrated with the code developed by another developer, both of them must agree on a set of common notations (function names, argument lists, return values etc.).
- In case of java, this is achieved by specifying an Interface class. An Interface is very similar to a class except that it only contains methods without any implementation. An Interface can also contain constants (Final variables)
- abstraction, you can also use abstract class but choosing between Interface in Java and abstract class is a skill.

Example

```
public class Main
    {
     public static void main(String[] args) {
       shapeA circleshape=new circle();
       circleshape.Draw();
       circleshape.Draw();
           }
    }
```

```
interface shapeA
{
   public   String baseclass="shape";
   public void Draw();
}
interface shapeB extends shapeA
{
   public   String baseclass="shape2";
   public void Draw2();
}
 class circle implements shapeB
{
  public   String baseclass="shape3";
  public void Draw() {
      System.out.println("Drawing Circle here:"+baseclass);
  }
  @Override
  public void Draw2() {
       System.out.println("Drawing Circle here:"+baseclass);
  }
}
```

The output is :

```
java code
Drawing Circle here:shape3
Drawing Circle here:shape3
```

Conclusion

- Object Orientation supports the programmers by:
  - using abstraction and encapsulation to enables us to focus on and program different parts of a complex system without worrying about ‘the whole’.
  - using inheritance to ‘factor out’ common code
  - using polymorphism to make programs easier to change
  - using tools help document and manage large software projects.

Assignment 1

- Use the same example of simple location in this presentation, calculate the distance from your location to the IT college Tripoli university location. And there is overloading or overriding in the code show it and explain the differences between overloading and overriding.

- Draw the UML diagram of the shape class example from slide 24 to the end of movable classes 31.