Java Tutorial: Working with Objects and Classes

Mark A. Austin

University of Maryland

austin@umd.edu
ENCE 688R, Spring Semester 2023

March 5, 2023

Overview

Working with Objects

Object-based Software

Basic Assumptions:

- Everything is an object.
- New kinds of objects can be created by making a package containing other existing objects.
- Objects have relationships for other types of objects.
- Objects have type.
- Object communicate via message passing – all objects of the same type can receive and send the same kinds of messages.
- Objects can have executable behavior.
- Objects can be design to respond to occurrences and events.
- Systems will be created through a composition (assembly) of objects.

Working with Objects   Encapsulation and Data Hiding   Relationships Among Classes   Association Relationships   Inheritance Mechan

○○●○○○○○○○○○○○   ○○○○○○○○○○○   ○○○○○   ○○○○○○○   ○○○○○○○○○○○

Working with Objects and Classes

Working with Objects and Classes:

- Collections of objects share similar traits (e.g., data, structure, behavior).
- Collections of objects will form relationships with other collections of objects.
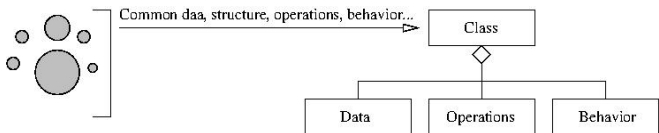

Definition of a Class

A class is a specification (or blueprint) of an object's structure and behavior.
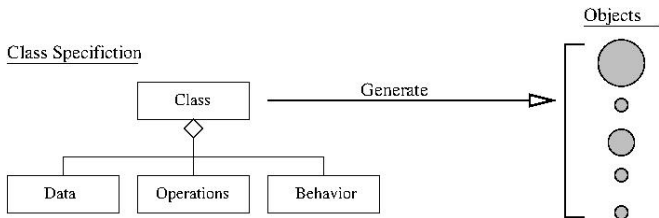

Definition of an Object

An object is an instance of a class.

Working with Objects and Classes

From Collections of Objects to Classes:

Generation of Objects from Class Specifications:

Working with Objects and Classes

Key Design Tasks

- Identify objects and their attributes and functions,
- Establish relationships among the objects,
- Establish the interfaces for each object,
- Implement and test the individual objects,
- Assemble and test the system.

Implicit Assumptions:

- Manual synthesis of the object model is realistic for systems that have a modest number of elements and relationships.
- As the dimensionality of the problem increases some form of automation will be needed to discover elements and relationships.

Working with Objects and Classes

Organizational and Efficiency Mechanisms:
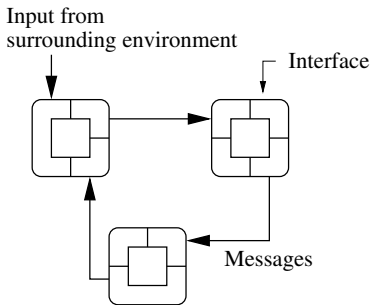
Network of Communicating Objects
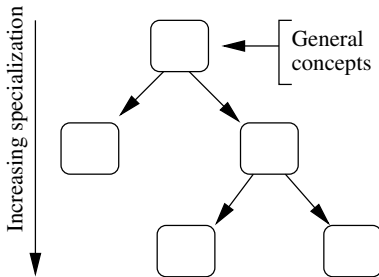
Problem Domain Concepts organized into a Class Hierarchy.

Example 1. Working with Points

A Very Simple Class in Java

```
1      public class Point {
2         int x, y;
3
4         public Point ( int x, int y ) {
5            this.x = x; this.y = y;
6         }
7      }
```

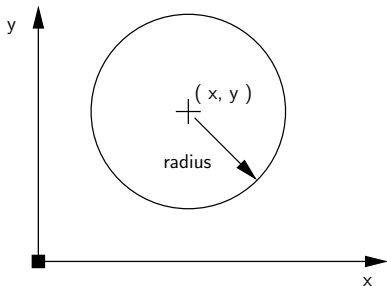### Creating an Object

```
8      Point  first = new Point ( 1, 2 );
9      Point second = new Point ( 2, 5 );
```

Accessing and Printing the attributes on an Object

```
10     System.out.printf(" first point (x,y) = (%2d, %2d)\n",  first.x,  first.y );
11     System.out.printf("second point (x,y) = (%2d, %2d)\n", second.x, second.y );
```

Example 2. Working with Circles

A circle can be described by the (x,y) position of its center and
by its radius.

There are numerous things we can do with circles:

- Compute their circumference, perimeter or area,
- Check if a point is inside a circle.

Example 2. Working with Circles

```
1   /*
2    *  ====================================================================
3    *  Circle(): Basic implementation of a circle program.
4    *
5    *  Written by: Mark Austin                              February, 2019
6    *  ====================================================================
7    */
8
9   import java.lang.Math.*;
10
11  public class Circle {
12      public double dX, dY, dRadius;
13
14      // Constructor
15
16      public Circle () {}
17
18      public Circle ( double dX, double dY, double dRadius ) {
19          this.dX = dX;
20          this.dY = dY;
21          this.dRadius = dRadius;
22      }
23
24      // Compute the circle area ....
25
26      public double Area() {
27          return Math.PI*dRadius*dRadius;
28      }
```

Example 2. Working with Circles

```
29
30        // Copy circle parameters to a string format ...
31
32        public String toString() {
33            return "(x,y) = (" + dX + "," + dY + "): Radius = " + dRadius;
34        }
35
36        // --------------------------------------------------------
37        // Exercise methods in class Circle ...
38        // --------------------------------------------------------
39
40        public static void main( String [] args ) {
41
42            System.out.println("Exercise methods in class Circle");
43            System.out.println("================================");
44
45            Circle cA = new Circle();
46            cA.dX = 1.0; cA.dY = 2.0; cA.dRadius = 3.0;
47
48            Circle cB = new Circle( 1.0, 2.0, 2.0 );
49
50            System.out.printf("Circle cA : %s\n", cA.toString() );
51            System.out.printf("Circle cA : Area = %5.2f\n", cA.Area() );
52            System.out.printf("Circle cB : %s\n", cB );
53            System.out.printf("Circle cB : Area = %5.2f\n", cB.Area() );
54        }
55    }
```

Example 2. Working with Circles

Script of Program Input and Output

```
Exercise methods in class Circle
================================
Circle cA : (x,y) = (1.0,2.0): Radius = 3.0
Circle cA : Area = 28.27
Circle cB : (x,y) = (1.0,2.0): Radius = 2.0
Circle cB : Area = 12.57
```

Points to note:

- Objects are created with constructor methods. The line:
    ```
    public Circle () {}
    ```

    is the default constructor. It creates circle objects with all of
    the circle attribute values initialized to zero.

Example 2. Working with Circles

More points to note:

- The next three statements use the dot notation (.) to manually initialize the (x,y) coordinates of the circle center and its radius.

- A second constructor method:

```
public Circle( double dX, double dY, double dRadius ) {
}
```

creates a circle object and initializes the circle attribute values in one line.

- Statements of the form this.dX = dX take the value of dX passed to the contructor method and assign it to the attribute dX associated with this object.

Accessing Object Data and Object Methods

Now that we have created an object, we can use its data fields. The dot operator (.) is used to access the different public variables of an object.

Example 1

```
Circle cA = new Circle();
cA.dX = 1.0;
cA.dY = 2.0;
cA.dRadius = 3.0;
```

To access the methods of an object, we use the same syntax as accessing the data of the object, i.e., the dot operator (.).

Accessing Object Methods

Example 2

```
Circle cA  = new Circle();
cA.dRadius = 2.5;
double dArea = cA.getArea();
```

Notice that we did not write dArea = getArea(cA);

Example 3

Let a, b, c, and d be complex numbers. To compute a*b + c*d we write

```
a = new Complex(1,1); .. etc ..

Complex sum = a.Mult(b).Add( c.Mult(d) );
```

Encapsulation and Data Hiding

Encapsulation and Data Hiding

Definition of Aggregation

- Aggregation is the grouping of components into a package.
- Aggregation does not imply that the components are hidden or inaccessible. It merely implies that the components are part of a whole.

Definition of Encapsulation

- Encapsulation is a much stronger form of organization.
- Encapsulation forces users of a system to deal with it as an abstraction (e.g., a black box) with well-defined interfaces that define what the entity is, what it does, and how it should be used.
- The only way to access an object's state is to send it a message that causes one of the object's internal methods to execute.
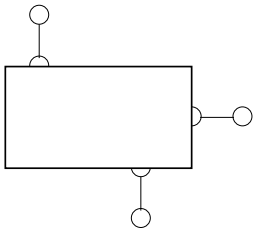
Encapsulation and Data Hiding
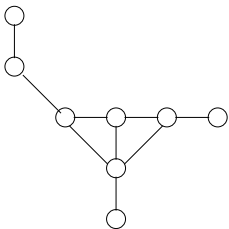
Unstructured Components
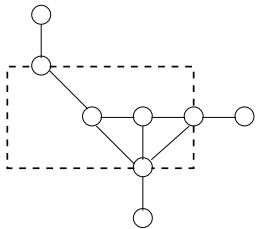
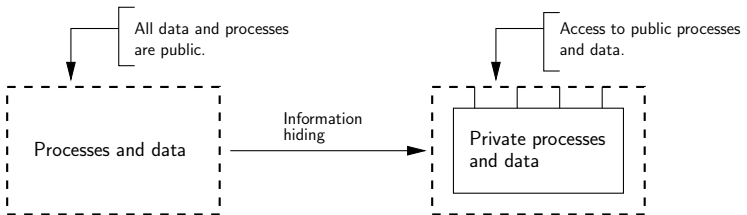Aggregation
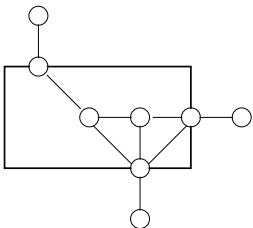
Designer's view of Aggregation

Encapsulation – User's view of Abstraction

Encapsulation and Data Hiding

Principle of Information Hiding

The principle of information hiding states that information which is likely to change (e.g., over the lifetime of a software/systems package) should be hidden inside a module.
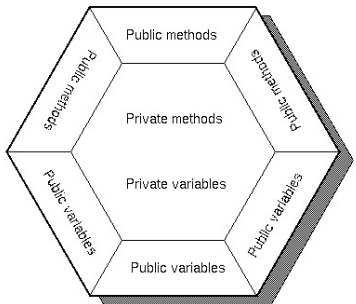
Application. Process for Implementation of Information Hiding.

All data and processes
are public.

Access to public processes
and data.

Processes and data

Information
hiding

Private processes
and data

Encapsulation and Data Hiding

Graphical Representation of a Class

Graphical representation of a Class

The object wrapping protects the object code from unintended access by other code.

Encapsulation and Data Hiding

In object-oriented terminology, and particularly in Java,

- The wrapper object is usually called a class, the functions inside the class are called private methods,
- The data inside the class are private variables.
- Public methods are the interface functions for the outside world to access your private methods.

Implementation. The keyword private in:

```
public class Point {
    private int x, y;
    .... .....
}
```

restricts to scope of x and y to lie inside the boundary of Point objects.

Encapsulation and Data Hiding

Access to a point's coordinates is controlled through the public methods:

```
public int  getX() {
    return x;
}
public void setX(int x) {
    this.x = x;
}
```

Example 2. Revised Circle Program

Revised circle program where data and circle properties can only be accessed through an interface.

```
 1   /*
 2    *  ===================================================================
 3    *  Circle(): Implementation of the Circle class where data and circle
 4    *            properties can only be accessed through an interface.
 5    *
 6    *  Written by: Mark Austin                              February, 2019
 7    *  ===================================================================
 8    */
 9
10   import java.lang.Math.*;
11
12   public class Circle {
13       protected double dX, dY, dRadius;
14
15       // Constructor
16
17       public Circle () {}
18
19       public Circle( double dX, double dY, double dRadius ) {
20           this.dX = dX;
21           this.dY = dY;
22           this.dRadius = dRadius;
23       }
24
25       // Compute the circle area ....
```

Example 2. Revised Circle Program

```
26
27      private double Area() {
28          return Math.PI*dRadius*dRadius;
29      }
30
31      // Create public interface for variables and area computation....
32
33      public void setX (double dX) {
34          this.dX = dX;
35      }
36
37      public double getX () {
38          return dX;
39      }
40
41      ... details for setY() and getY() removed ...
42
43      public void setRadius (double dRadius ) {
44          this.dRadius = dRadius;
45      }
46
47      public double getRadius () {
48          return dRadius;
49      }
50
51      public double getArea() {
52          return Area();
53      }
54
55      // Copy circle parameters to a string format ...
```

Example 2. Revised Circle Program

```
56
57      public String toString() {
58          return "(x,y) = (" + dX + "," + dY + "): Radius = " + dRadius;
59      }
60
61      // --------------------------------------------------------
62      // Exercise methods in class Circle ...
63      // --------------------------------------------------------
64
65      public static void main( String [] args ) {
66
67          System.out.println("Exercise methods in class Circle");
68          System.out.println("================================");
69
70          Circle cA = new Circle();
71          cA.setX(1.0);
72          cA.setY(2.0);
73          cA.setRadius(3.0);
74
75          Circle cB = new Circle( 1.0, 2.0, 2.0 );
76
77          System.out.printf("Circle cA : %s\n", cA.toString() );
78          System.out.printf("Circle cA : Area = %5.2f\n", cA.getArea() );
79
80          System.out.printf("Circle cB : %s\n", cB );
81          System.out.printf("Circle cB : Area = %5.2f\n", cB.getArea() );
82      }
83  }
```

Example 2. Revised Circle Program

Points to note:

- Use of the keyword protected in:

      protected double dX, dY, dRadius;

  restricts access of dX, dY and dRadius to methods within Circle and any subclass of Circle.

- The methods getX() and setX(), etc, create a public interface for Circle.

- By convention, the toString() method creates and returns a string description of the objects contents. And it can be called in two ways as demonstrated at the bottom of main(). The fragment of code cA.toString() will return a string which will be matched against the %s format specification. However, cB also calls toString() and is shorthand for cB.toString().

Relationships

Among Classes

Relationships Among Classes

Motivation

- Classes and objects by themselves are not enough to describe the structure of a system.
- We also need to express relationships among classes.
- Object-oriented software packages are assembled from collections of classes and class-hierarchies that are related in three fundamental ways.

Relationships Among Classes

1. Use: Class A uses Class B (method call).
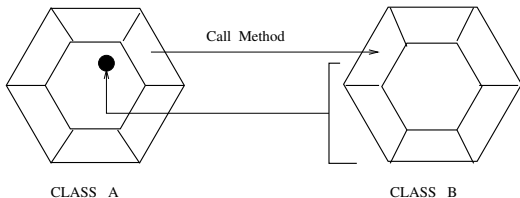
CLASS A   CLASS B

Class A uses Class B if a method in A calls a method in an object of type B.

Example

```
double dAngle = Math.sin ( Math.PI / 3.0 );
```

Relationships Among Classes

2. Containment (Has a): Class A contains a reference to Class B.

CLASS A   CLASS B
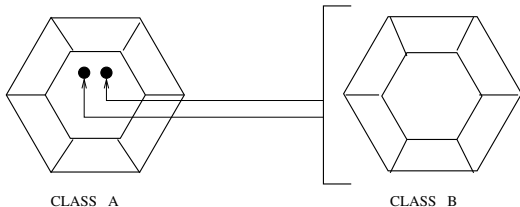
Clearly, containment is a special case of use (i.e., see Item 1.).

Example

```
public class LineSegment {
   private Point start, end;
   .......
}
```

Relationships Among Classes

3. Inheritance (Is a): In everyday life, we think of inheritance as something that is received from a predecessor or past generation. Here, Class B inherits the data and methods (extends) from Class A.

CLASS A   CLASS B
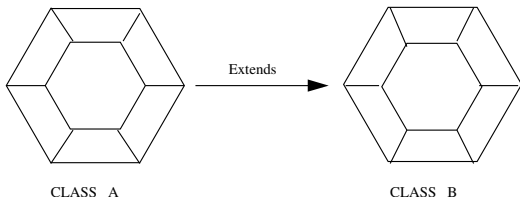
Examples of Java Code

```
public class ColoredCircle extends Circle { .... }
public class GraphicalView extends JFrame { .... }
```

Association Relationships

Definition

As association is a discrete and/or logical relationship between classes. Associations are the glue that tie the elements of a system together.

Binary Association Relationships

Binary associations express static bidirectional relationships between two classes.

Meta-Model

multiplicity indicators

Class A — association name — Class B

role of class A   role of class B
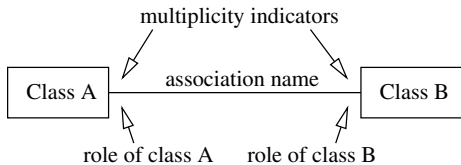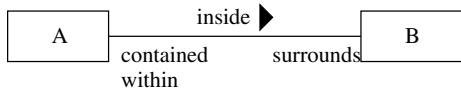
Engineering Viewpoint

B

A

Example

A — inside ▶ — B

contained within   surrounds
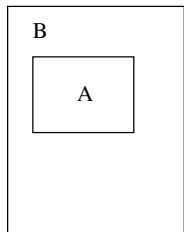
Binary Association Relationships

Meta-Model for Links and Association Relationships. Links and associations establish relationships among entities within the problem world or the solution world.
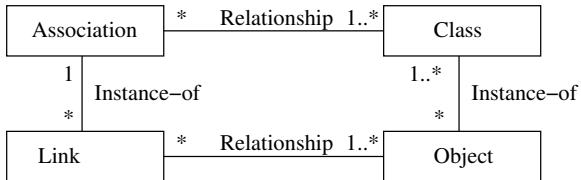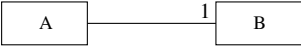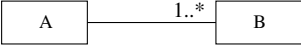
Points to note:

- Associations are descriptions of links with a common implementation.
- Links are instances of an association relationship.

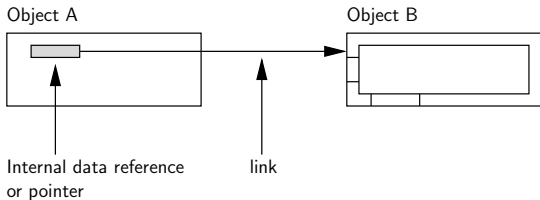## Association Relationships

Multiplicity Constraints. Indicate the number of objects participating in an instance of an association.
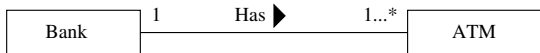
Relationship	Multiplicity
A —1— B	Exactly one to one
A —0..1— B	Optional (zero or one)
A —*— B	Many (zero or more)
A —1..*— B	Many (one or more)
A —m..n— B	Numerically specified

Association Relationships

Example 1. Object A links to Object B
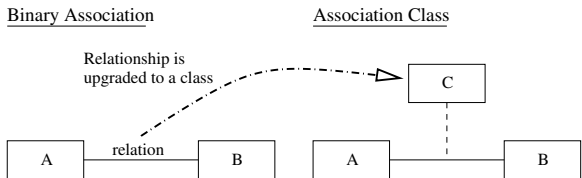

Example 2. A bank and a suite of ATMs

- A bank has one or more ATMs.
- Each ATM is associated with one (and only one) bank.

Association Class Relationships

From Binary Relations to Association Classes

Binary Association

Association Class

Relationship is
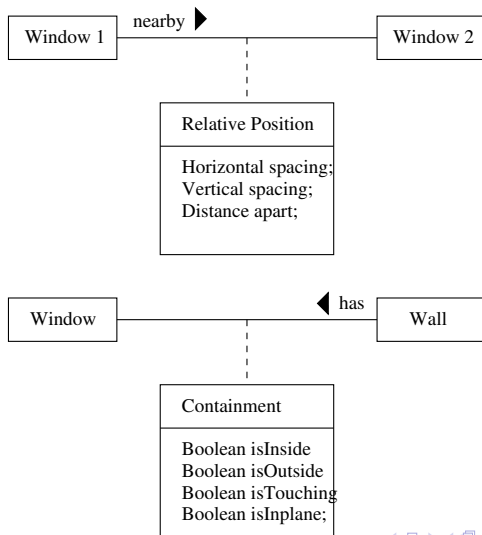upgraded to a class

C

A   relation   B

A   B

Association classes are used when:

- The association itself has attributes or operations that need to be represented in the class model.
- It makes sense for the "one association occurrence, one association class instance" constraint to exist.

Association Class Relationships

Two examples:

Inheritance Mechanisms

Inheritance Mechanisms

Inheritance Structures

Inheritance structures allow you to capture common characteristics in one model artifact and permit other artifacts to inherit and possibly specialize them. Class hierarchies are explicitly designed for customization through extension.

In this approach to development:

- Forces us to identify and separate the common elements of a system from those aspects that are different/distinct.
- Commonalities are captured in a super-class and inherited and specialized by the sub-classes.
- Inherited features may be overridden with extra features designed to deal with exceptions.

Base and Derived Classes

Goal: Avoid duplication and redundancy of data in a problem specification.
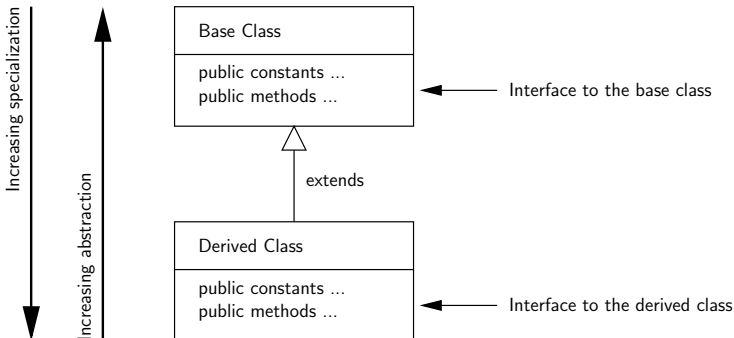
Base and Derived Classes

Points to note:

- A class in the upper hierarchy is called a superclass (or base, parent class).
- A class in the lower hierarchy is called a subclass (or derived, child, extended class).
- The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher-level classes.

Inheritance Mechanisms

Example 2. Hierarchy of Temperature Sensors

Temperature Thermometer

- Consider a class hierarchy for attributes and functions in a family of temperature sensors.
- The super-class represents a generic temperature sensor.
- Super-class attributes: measured temperature, sensor weight, mean-time-to-failure (MTTF).
- Methods are provided to test the sensor.

Water Temperature Thermometer

- A water temperature thermomenter is a generic temperature sensor + a field to store the depth at which the temperature was recorded.

Inheritance Mechanisms
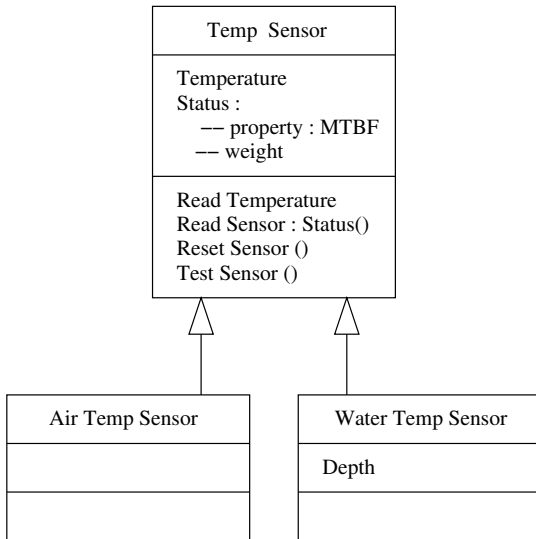
Inheritance Mechanisms

Multiple Inheritance Structures

- In a multiple inheritance structure, a class can inherit properties from multiple parents.
- The downside is that properties and/or operations may be partially or fully contradictory.

Example

- People is a generalization of Children and Customers.
- Young customers inherits properties from Customers and Children.

Note. Unlike C++ and Python, Java explicitly prevents multiple inheritance. Java classes can, however, have multiple interfaces.

Inheritance Mechanisms
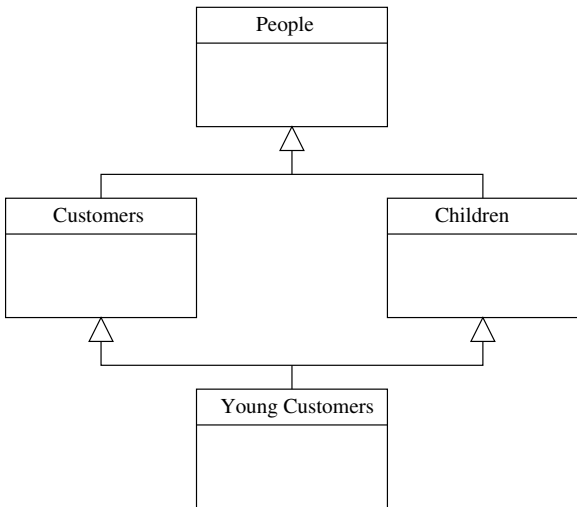
Example 3. Extending Circle to Colored Circle

```
 1    /*
 2     *   ===================================================================
 3     *   ColoredCircle(): Implementation of the ColoredCircle class where
 4     *                    data and circle properties can only be accessed
 5     *                    through an interface.
 6     *
 7     *   Written By: Mark Austin                              April 2019
 8     *   ===================================================================
 9     */
10
11    package objects;
12
13    import java.awt.Color;
14
15    public class ColoredCircle extends Circle {
16        private Color color;
17
18        // Constructor methods
19
20        public ColoredCircle() {
21            super();
22            this.color = Color.blue;
23        }
24
25        public ColoredCircle( double dX, double dY, double dRadius, Color color ) {
26            super();
27
28            this.dX = dX;
```

Example 3. Extending Circle to Colored Circle

```
28              this.dX = dX;
29              this.dY = dY;
30              this.dRadius = dRadius;
31              this.color   = color;
32          }
33
34          // Set and retrieve colors ....
35
36          public void setColor( Color color ) {
37              this.color = color;
38          }
39
40          public String getColors() {
41              return "Color (r,g,b) = (" + color.getRed() + "," + color.getGreen() + "," + colo
42          }
43
44          // ================================================
45          // Exercise methods in class ColoredCircle()......
46          // ================================================
47
48          public static void main( String [] args ) {
49
50              System.out.println("Exercise methods in class ColoredCircle");
51              System.out.println("========================================");
52
53              // Create, initialize, and print circle "cA" ...
54
55              ColoredCircle cA = new ColoredCircle( 1.0, 2.0, 3.0, Color.blue );
```
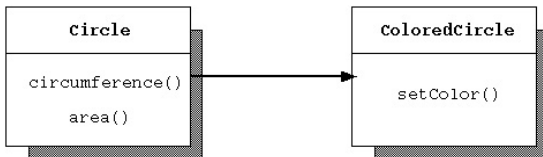
Example 3. Extending Circle to Colored Circle

Example 3. Extending Circle to create Colored Circle

Two public methods are defined for this class:

- setColor. This method takes a color as its argument and assigns this value to the color of the circle.
- ColoredCircle. This method has the same name as the class itself; it is a constructor method.

The method call super() invokes the constructor method of the superclass [i.e., the method Circle()].

Composition of

Object Models

Composition of Object Models

Definition

Composition is known as is a part of or is a relationship.

The member object is a part of the containing class and the member object cannot survive or exist outside the enclosing or containing class or doesn't have a meaning after the lifetime of the enclosing object.
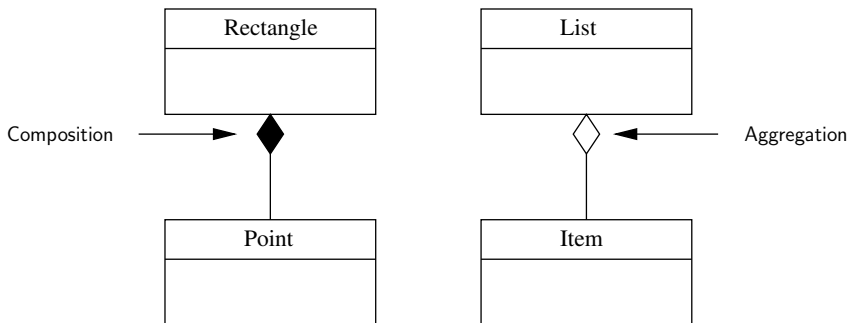
Is it Aggregation or Composition?

- Ask the question: if the part moves, can one deduce that the whole moves with it in normal circumstances?

Example: A car is composition of wheels and an engine. If you drive the car to work, hopefully the wheels go too!

Composition of Object Models

Notation for Aggregation and Composition

Recall: Aggregation is all about grouping of things ...

Example 4. Modeling Line Segments

Example 1. Line segment is composed from two points:
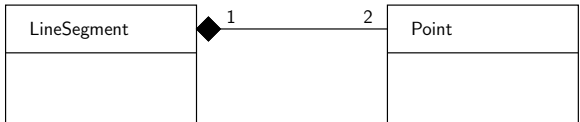
Source Code: Abbreviated Point.java

```
1   public class Point {
2       private int x, y;
3
4       public Point(int x, int y) {
5           this.x = x;
6           this.y = y;
7       }
8
9       public int  getX() {
10          return x;
11      }
12
13      public void setX(int x) {
14          this.x = x;
15      }
```

Example 4. Modeling Line Segments

Source Code: Point.java continued:

```
16
17      public int  getY()     { return y; }
18      public void setY(int y) { this.y = y; }
19
20      public String toString() {
21          return "(" + x + "," + y + ")";
22      }
23  }
```

**Source Code:** Abbreviated LineSegment.java

```
1   public class LineSegment {
2       Point begin, end;
3
4       public LineSegment (int x1, int y1, int x2, int y2) {
5           begin = new Point(x1, y1);
6           end   = new Point(x2, y2);
7       }
8
9       public String toString() {
10          return "Line segment: from " + begin + " to " + end;
11      }
12  }
```

Example 4. Modeling Line Segments

Creating a Line Segment Object:

```
LineSegment segmentA = new LineSegment( 1, 2, 3, 4 );
```

The layout of memory is as follows:
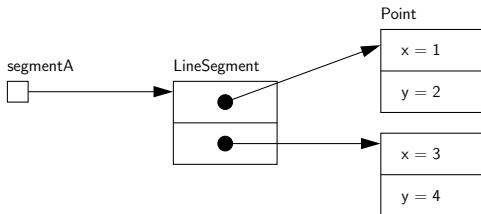
Here, segmentA refers to the memory location for the linesegment object. The linesegment object contains references to Point objects containing the (x,y) coordinates.

Spatial Applications

Spatial Models (Points, Lines, Polygons)
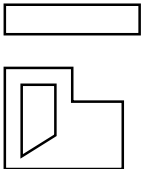
Points, lines and regions are fundamental spatial data types.

Point   Line   Region

- Points are 0-dimensional entities. Lines are 1-dimensional entities. Regions are 2-dimensional entities.
- Spatial operations: union, intersection, difference.
- We need software that can compute operations on these entities in a consistent manner (e.g., Google: Java Topology Suite).
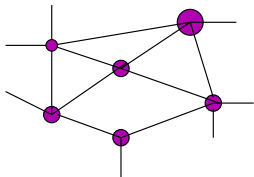
Class Diagram for GIS Domain

Partitions and Networks
Partitions and networks are two abstractions for modeling collections of spatial objects.
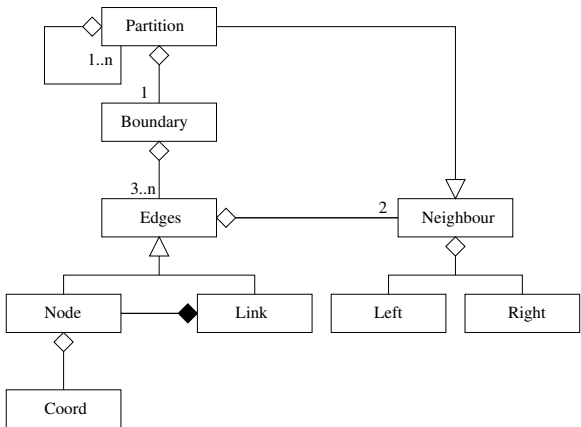
Partitions
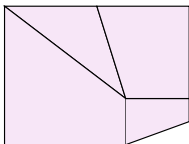
Spatially Distributed Network

- Examples of partitions: rooms in a building, districts in a state, countries in a continent.
- Examples of networks: plumbing and HVAC networks, highways and railway networks, communication and power networks.

Class Diagram for GIS Domain

Conceptual model for partition hierarchies (adapted from Chunithipaisanl S. et al., 2004)
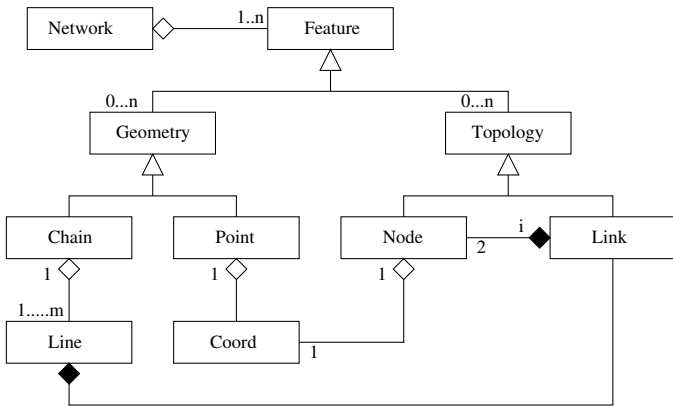
Class Diagram for GIS Domain

The conceptual model for partitions states:

- A Partition can be decomposed into 1 or more Partitions (sub-Partitions).
- Each Partition has one boundary (here we ignore the possibility of partitions containing holes).
- Boundaries are composed of edges (..at least 3 edges).
- Each Edge segment has a Node and Link.
- Nodes and Link are paired in a one-to-one correspondence.
- A Node has a coordinate.
- Edges also have Neighboring Partitions.
- Neighboring Partitions can be classified as to whether they are on the Left and Right of the Edge.

Class Diagram for GIS Domain

Conceptual model for networks (Adapted from: Chunithipaisanl S. et al., 2004).

Class Diagram for GIS Domain

The conceptual model for networks states:

- A Network is composed of Features.
- **2.** Each Feature has Geometry and Topology.
- Geometry is a generalization for Chains and Points...
- A Chain corresponds to one or more Line segments.
- A Point has a coordinate.
- Topology is a generalization for Nodes and Links.
- Nodes also have coordinates.

Layered Organization of Attributes in Urban Data
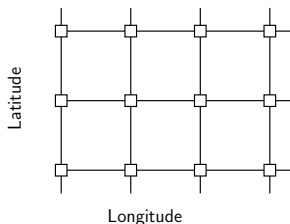
Geographic Information System
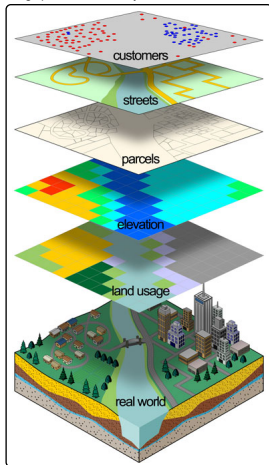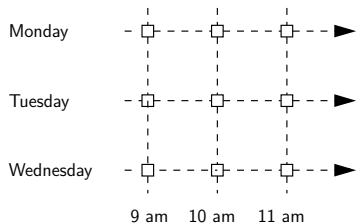

Layers of Data / Information in Military Decision Making
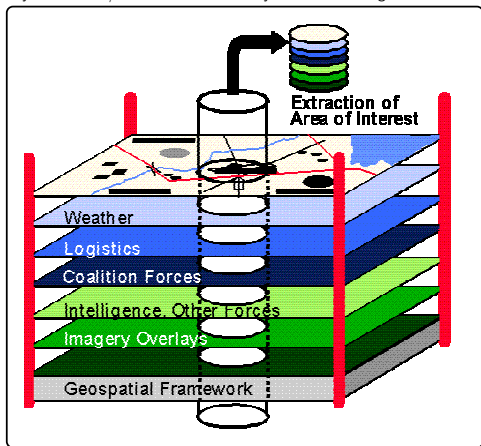
Spatial and Temporal Domains

Goal. We want to know that systems do the right thing (event) in the right place (spatial) at the right time (temporal).

Spatial Domain

Latitude

Longitude

Temporal Domain

Monday

Tuesday

Wednesday

9 am 10 am 11 am

2D Spatial Domain: OpenStreetMap, Java Topology Suite.
Temporal Domain: Calendars, Scheduling Algorithms, Ontologies of Time, UPPAAL.