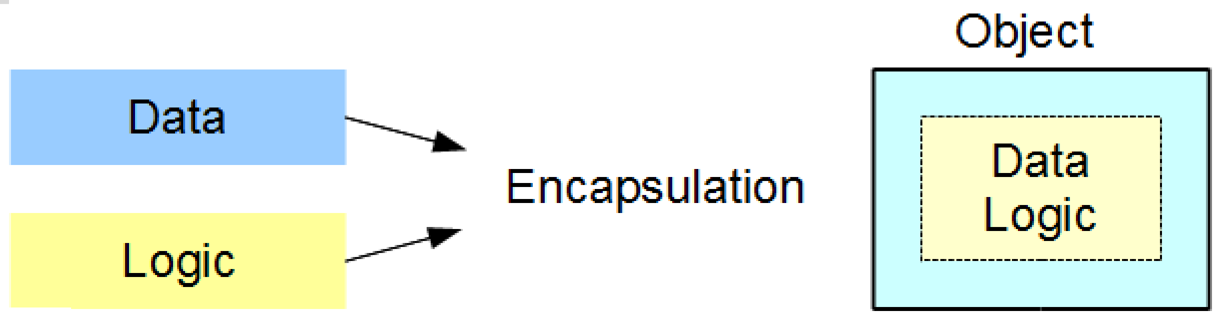
Principles of the Object Oriented Paradigm

Principle of Encapsulation



Public Interface

A *sample* Client Program:

for the Rectangle Class

```
public class RectangleClient {

    public static void main(String[] args) {

        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        System.out.println("r1's area = " + r1.area() );

        System.out.println("r2's area = " + r2.area() );

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1);
        System.out.println("r2: " + r2);
    }
}
```

Classes as Custom Data Type:

a summary

```
public class TestClass {
   public static void main(String[] args) {

      int x;                       // no-arg constructor
      int x = 5;                   // custom contructor
      x = 12;                      // mutator/setter method
      int j = x;                   // accessor method
      System.out.println(x);      // toString method
      if ( x == j ) {              // equals method

      }
      if ( j == x ) {              // equals method

      }
   } // end of main method
} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
public class TestClass {
   public static void main(String[] args) {

      int x;       // Rectangle r1 = new Rectangle();
      int x = 5;   // Rectangle r2 = new Rectangle(3,3);
      x = 12;                   // r1.grow(5,4);
      int j = x;                // int area = r2.area();
      System.out.println(x);    // r1.toString();
      if ( x == j ) {           // r1.equals(r2);

      }
      if ( j == x ) {           // r2.equals(r1);

      }
   } // end of main method
} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows objects on the stack and heap

int main( ) {

      int x;                  // Rectangle r1;
      int x = 5;              // Rectangle r2 = new Rectangle(5);
      x = 12;                      // r1 = 12;
      int j = x;                   // j = r1;
      cout << x;                   // cout << r1;
      if ( x == j ) {              // r1 == r2);

      }
      if ( j == x ) {              // r2 == r2;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows objects on the stack and heap

int main( ) {

      int x;                  // Rectangle r1;
      int x = 5;              // Rectangle r2 = new Rectangle(5);
      x = 12;                      // r1 = 12;
      int j = x;                   // j = r1;
      cout << x;                   // cout << r1;
      if ( x == j ) {              // r1 == r2);

      }
      if ( j == x ) {              // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows objects on the stack and heap!

int main( ) {

      int x;                    // Rectangle r1;
      int x = 5;                // Rectangle r2 = new Rectangle(5);
      x = 12;                        // r = 12;
      int j = x;                     // j = r1;
      cout << x;                     // cout << r1;
      if ( x == j ) {                // r1 == r2);

      }
      if ( j == x ) {                // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows operator overloading

int main( ) {

      int x;                          // Rectangle r1;
      int x = 5;                      // Rectangle r2 = new Rectangle(5);
      x = 12;                         // r1 = 12;
      int j = x;                      // j = r1;
      cout << x;                      // cout << r1;
      if ( x == j ) {                 // r1 == r2);

      }
      if ( j == x ) {                 // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                          // Rectangle r1;
      int x = 5;                      // Rectangle r2 = new Rectangle(5);
      x = 12;                         // r1 = 12;
      int j = x;                      // j = r1;
      cout << x;                      // cout << r1;
      if ( x == j ) {                 // r1 == r2);

      }
      if ( j == x ) {                 // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                      // Rectangle r1;
      int x = 5;                  // Rectangle r2 = new Rectangle(5);
      x = 12;                     // r1.operator=(12);
      int j = x;                  // j = r1;
      cout << x;                  // cout << r1;
      if ( x == j ) {             // r1 == r2);

      }
      if ( j == x ) {             // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                      // Rectangle r1;
      int x = 5;                  // Rectangle r2 = new Rectangle(5);
      x = 12;                     // r1 = 12;
      int j = x;                  // j = r1;
      cout << x;                  // cout << r1;
      if ( x == j ) {             // r1 == r2);

      }
      if ( j == x ) {             // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

C++

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                          // Rectangle r1;
      int x = 5;                      // Rectangle r2 = new Rectangle(5);
      x = 12;                         // r1 = 12;
      int j = x;                      // operator=(j, r1);
      cout << x;                      // cout << r1;
      if ( x == j ) {                 // r1 == r2);

      }
      if ( j == x ) {                 // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                        // Rectangle r1;
      int x = 5;                    // Rectangle r2 = new Rectangle(5);
      x = 12;                       // r1 = 12;
      int j = x;                    // j = r1;
      cout << x;                    // cout << r1;
      if ( x == j ) {               // r1 == r2);

      }
      if ( j == x ) {               // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows operator overloading, on most operators:

int main( ) {                                              C++

      int x;                       // Rectangle r1;
      int x = 5;                   // Rectangle r2 = new Rectangle(5);
      x = 12;                      // r1 = 12;
      int j = x;                   // j = r1;
      cout << x;                   // operator<<(cout, r1);
      if ( x == j ) {              // r1 == r2);

      }
      if ( j == x ) {              // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// C++ allows operator overloading, on most operators:
                                                                C++
int main( ) {

      int x;                         // Rectangle r1;
      int x = 5;                     // Rectangle r2 = new Rectangle(5);
      x = 12;                        // r1 = 12;
      int j = x;                     // j = r1;
      cout << x;                     // cout << r1;
      if ( x == j ) {                // r1 == r2;

      }
      if ( j == x ) {                // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                          // Rectangle r1;
      int x = 5;                      // Rectangle r2 = new Rectangle(5);
      x = 12;                         // r1 = 12;
      int j = x;                      // j = r1;
      cout << x;                      // cout << r1;
      if ( x == j ) {                 // r1.operator==(r2);

      }
      if ( j == x ) {                 // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                        // Rectangle r1;
      int x = 5;                    // Rectangle r2 = new Rectangle(5);
      x = 12;                       // r1 = 12;
      int j = x;                    // j = r1;
      cout << x;                    // cout << r1;
      if ( x == j ) {               // r1 == r2;

      }
      if ( j == x ) {               // r2.operator==(r1);

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

C++

```
// C++ allows operator overloading, on most operators:

int main( ) {

      int x;                        // Rectangle r1;
      int x = 5;                    // Rectangle r2 = new Rectangle(5);
      x = 12;                       // r1 = 12;
      int j = x;                    // j = r1;
      cout << x;                    // cout << r1;
      if ( x == j ) {               // r1 == r2;

      }
      if ( j == x ) {               // r2 == r1;

      }

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// Python allows operator overloading on
  arithmetic and relational operators:
```

Python

```
int main( ) {



      x = 5                  // Rectangle r2 = new Rectangle(5);
      x = 12                 // mutator/setter methods
      j = x                  // accessor methods
      print(x)               // print(r1, r2)
      if x == j:             // r1 == r2


      if j == x:             // r2 == r1




   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// Python allows operator overloading on
  arithmetic and relational operators:

int main( ) {                                              Python



      x = 5                    // Rectangle r2 = new Rectangle(5);
      x = 12                   // mutator/setter methods
      j = x                    // accessor methods
      print(x)                 // print(r1, r2)
      if x == j:               // r1 == r2


      if j == x:               // r2 == r1



   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// Python allows operator overloading on
   arithmetic and relational operators:
```

Python

```
int main( ) {

      x = 5                // Rectangle r2 = new Rectangle(5);
      x = 12               // mutator/setter methods
      j = x                // accessor methods
      print(x)             // __str__
      if x == j:           // r1 == r2

      if j == x:           // r2 == r1

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// Python allows operator overloading on
  arithmetic and relational operators:
```

Python

```
int main( ) {



      x = 5                   // Rectangle r2 = new Rectangle(5);
      x = 12                  // mutator/setter methods
      j = x                   // accessor methods
      x                       // __repr__
      if x == j:              // r1 == r2


      if j == x:              // r2 == r1



   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

Python

```
// Python allows operator overloading on
  arithmetic and relational operators:

int main( ) {

      x = 5                    // Rectangle r2 = new Rectangle(5);
      x = 12                   // mutator/setter methods
      j = x                    // accessor methods
      print(x)                 // print(r1, r2)
      if x == j:               // r1 == r2

      if j == x:               // r2 == r1

   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

```
// Python allows operator overloading on
   arithmetic and relational operators:

                                                    Python
int main( ) {


      x = 5                  // Rectangle r2 = new Rectangle(5);
      x = 12                 // mutator/setter methods
      j = x                  // accessor methods
      print(x)               // print(r1, r2)
      if x == j:             // r1.__eq__(r2)


      if j == x:             // r2.__eq__(r2)



   } // end of main method

} // end of class TestClass
```

Classes as Custom Data Type:

a summary

Python

```
// Python allows operator overloading on
   arithmetic and relational operators:

int main( ) {

      x = 5                 // Rectangle r2 = new Rectangle(5);
      x = 12                // mutator/setter methods
      j = x                 // accessor methods
      print(x)              // print(r1, r2)
      if x == j:            // r1 == r2

      if j == x:            // r2 == r1

   } // end of main method

} // end of class TestClass
```

Principles of the Object Oriented Paradigm

- Hierarchical nature
- Identifiable Components
- Common Patterns

Handle and Model **Complex** Systems

Decomposition

Decomposition is used to break software into *components* that can be combined to solve the original problem.

Algorithmic decomposition breaks down the problem into major steps in the overall process of forming the solution. A top down structured approach to software development.

Decom

Decomposition is used

components that can be co ... nal

problem.

Highlights the
ordering of events.

Algorithmic decomposition breaks down the problem into major steps in the overall process of forming the solution. A top down structured approach to software development.

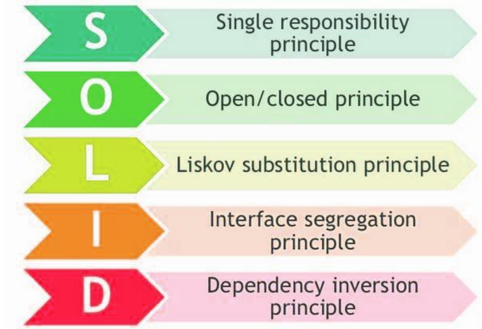
Decomposition

Decomposition is used to break software into *components* that can be combi nal problem.

Emphasizes the entities that
either cause action or
are themselves the subjects
on which these actions are
performed upon.

Algorithmic decomp into major steps in th e solution. A top down structured ap o software development.

Object decomposition breaks down the problem into identifiable objects. The objects themselves are derived directly from the problem domain.



Decomposition

Single Responsibility Principle

limiting the impact of change

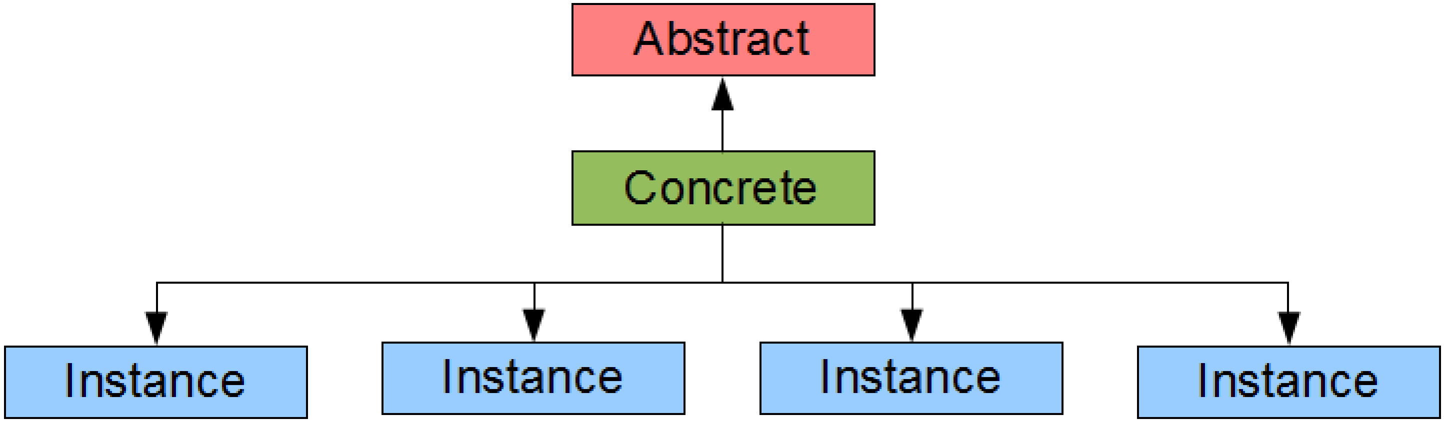
‘*gather together those things that change for the same reasons*’

a class should only **have one responsibility**, further defined by Martin as ‘***one reason to change***’

Robert Martin

Principles of the Object Oriented Paradigm

Principle of Abstraction



Abstraction is the Key

In order to go fast, we have to accept that we will write *bad* code…

Code that we will at “some point” go back and refactor or change to make better!

Specific code is easier to write then *general* code!

Abstraction is the Key

~~In order to go fast, we have to accept that we will write *bad* code…~~

Abstraction does not mean you have to solve every specific problem!

Abstraction allows you to build an architecture that will not force you to start from scratch every time you need to *pivot* or discover a new requirement.

More pragmatic and economical to build flexible architecture.

Principle of Abstraction

Purpose of abstraction is to handle the complexity of a software system by hiding unnecessary details from the user or client.

Abstraction assists us the process of ***decomposition***!

Principle of Abstraction

- High Level Programming Language: The same code or program can run on multiple computer architectures. The compiler handles the details of the translation from high level language to machine level instruction code.

High level language → machine architectures

Multiple machine architectures

Multiple machine architectures

Multiple machine architectures

Principle of Abstraction

- Abstraction by Parameterization

- Abstraction by Specification
  - Modifiability
  - Locality

Principle of Abstraction

- Abstraction by Parameterization

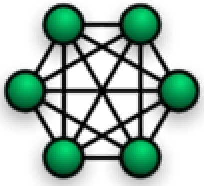
- Abstraction by Specification
  - Modifiability
  - Locality

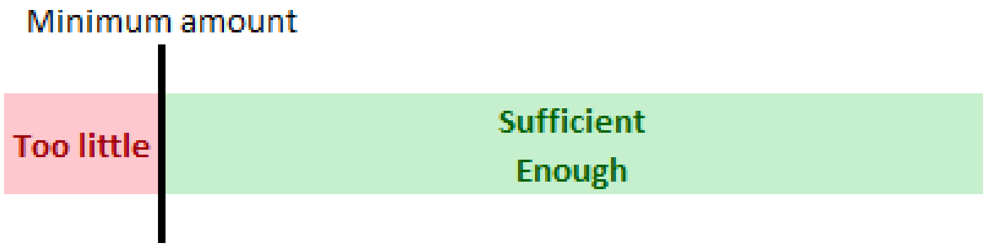
- Abstraction by *parameterization* and abstraction by *specification* are powerful methods for program construction. The enable us to define three different kinds of abstraction:
  - procedural
  - data
  - iteration

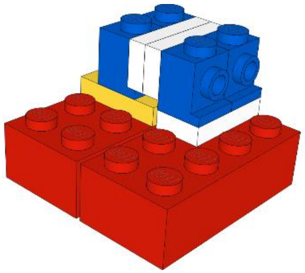
Quality of Abstraction

- How can determine if our class and object structure is well designed? Consider the following five factors.

1. Coupling
2. Cohesion
3. Sufficiency
4. Completeness
5. Primitiveness







Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse classes should have a weak coupling so that they can be used independent of other classes.

Weak

Implies minimal if any dependencies between classes. Classes which are independent can be used as building blocks to form new programs.

Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse classes ould have coupling so that they can be other classes.

However in OO Inheritance is one of the most powerful tools and this implies the **strongest** type of coupling!

Weak

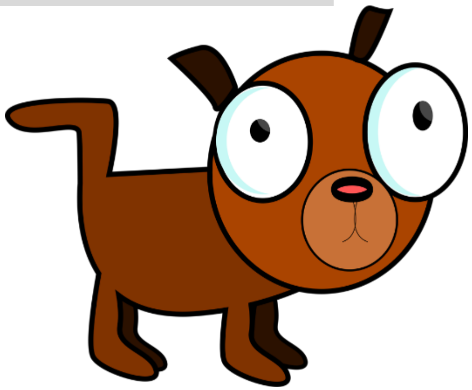
Implies minimal if any dependen n classes. Classes which are independent can be used as building blocks to form new programs.

Cohesion

Measures the degree of relatedness among the elements (entities) of a single class.

All the members and methods of a class should work together to provide a clearly identified behavior of a specific entity.

Example, a class Dog is cohesive if its characteristics embrace the behavior of a dog and only a dog and not a cat who thinks she is a dog!



Sufficient, Complete and Primitive

Sufficient mean that the class captures *enough* characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures *all* the characteristics of the abstraction.

Sufficient, Complete and Primitive

Sufficient mean that the class captures *enough* characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures *all* the characteristics of the abstraction.

Lets say you are designing a class Set, we need to include operations that both add and remove items in the set. Neglecting one operation, does not allow us to meaningfully use it. Therefore that class is not a sufficient implementation of a `Set`. However if the class does not implement the difference operation, though it may not be complete, it can still be used.

Sufficient, Complete and Primitive

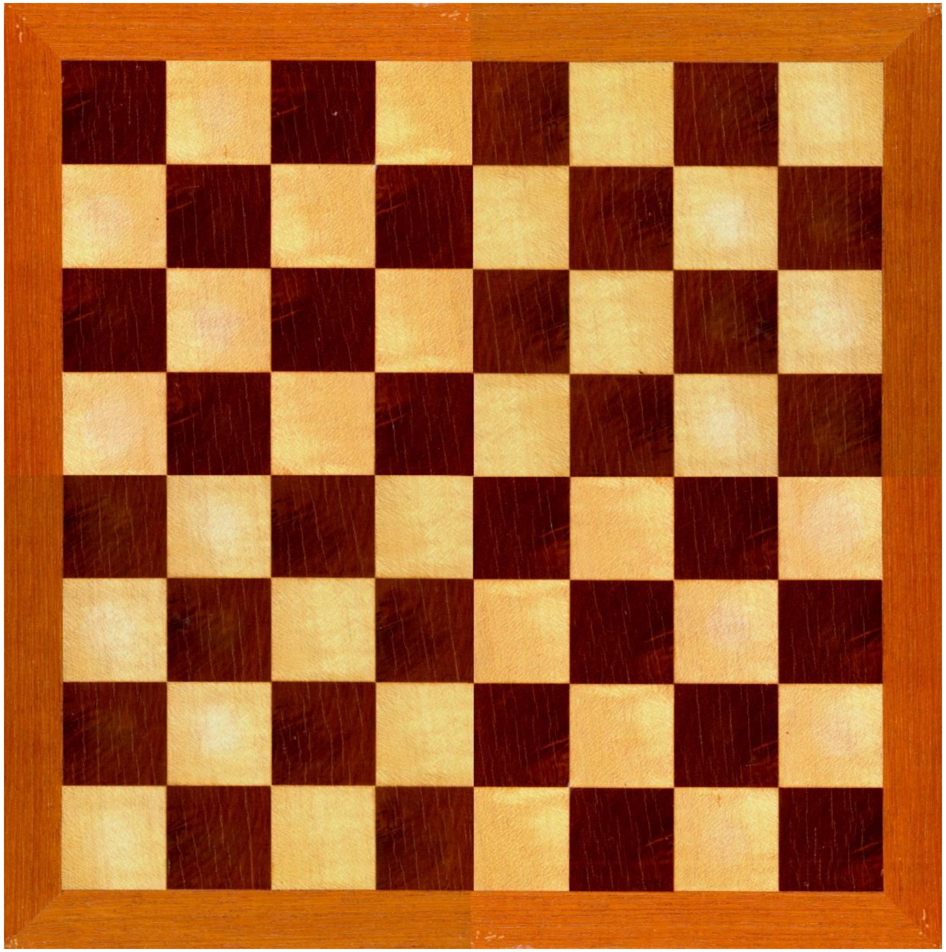
Sufficient mean that the class captures *enough* characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures *all* the characteristics of the abstraction.

Lets say you are designing a class Set, we need to include operations that both add and remove items in the set. Neglecting one operation, does not allow us to meaningfully use it. Therefore that class is not a sufficient implementation of a `Set`. However if the class does not implement the difference operation, though it may not be complete, it can still be used.

Primitive means that the classes and objects should be designed as small independent building blocks which can be used to build higher level and more complex operations.

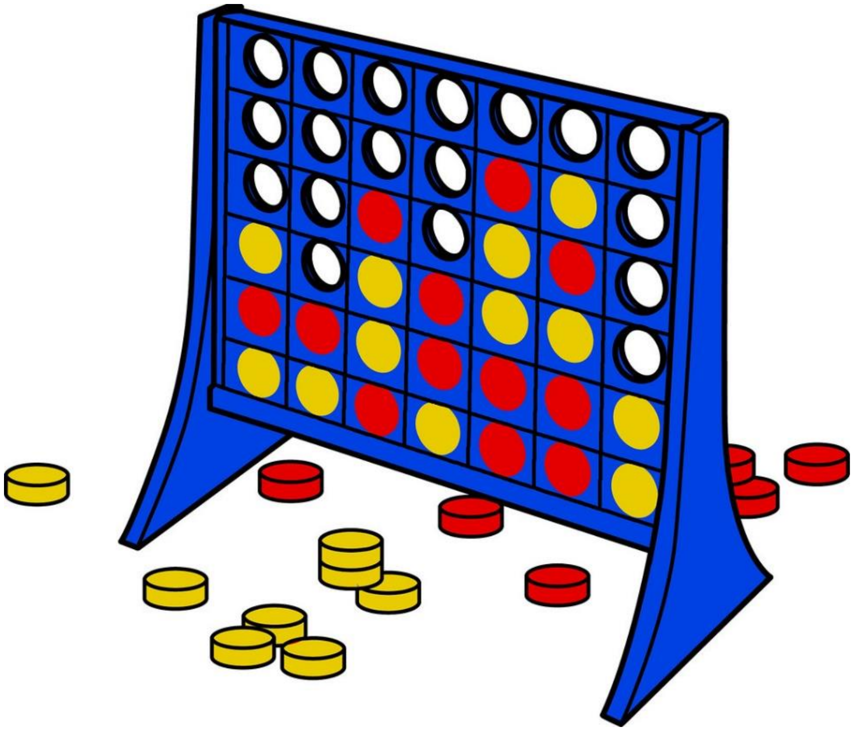
Object Decomposition:
Principle of Abstraction

Object Decomposition:
Principle of Abstraction



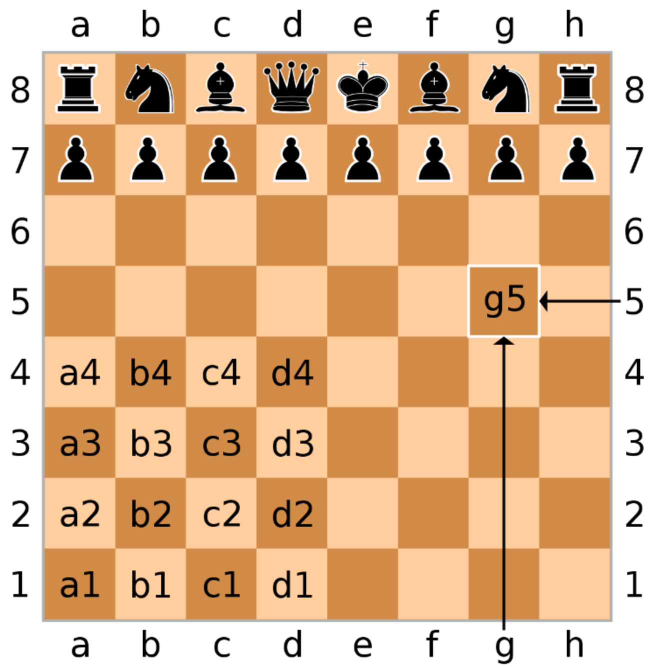
Object Decomposition:

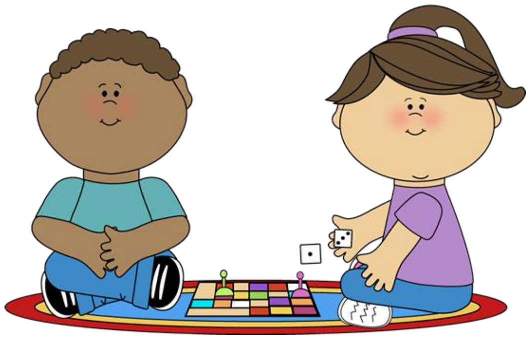
Principle of Abstraction



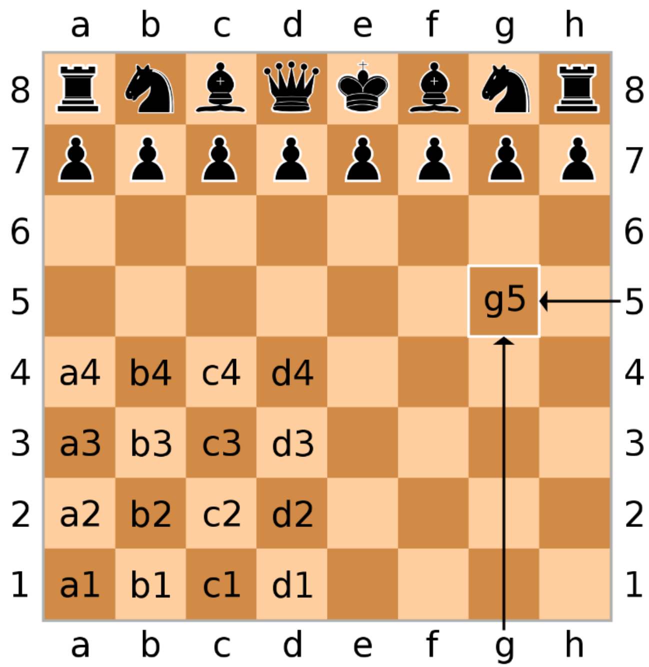
Object Decomposition:

Principle of Abstraction



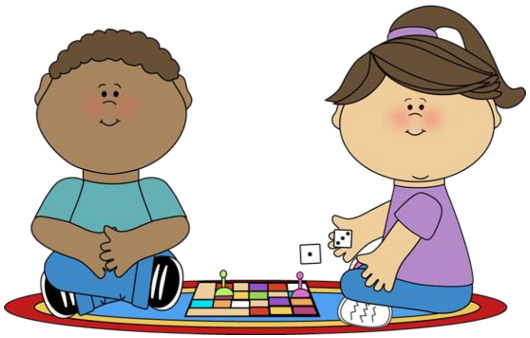


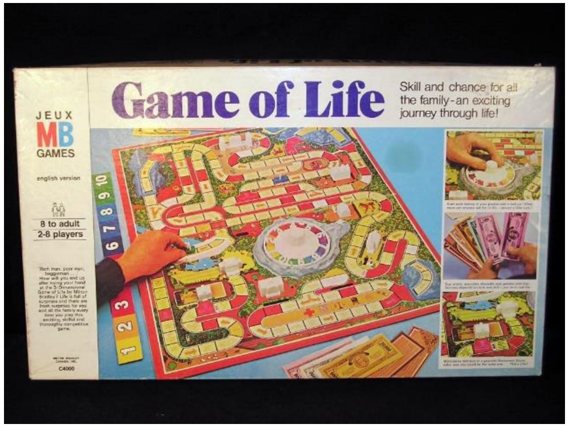
Object Decomposition:
Principle of Abstraction



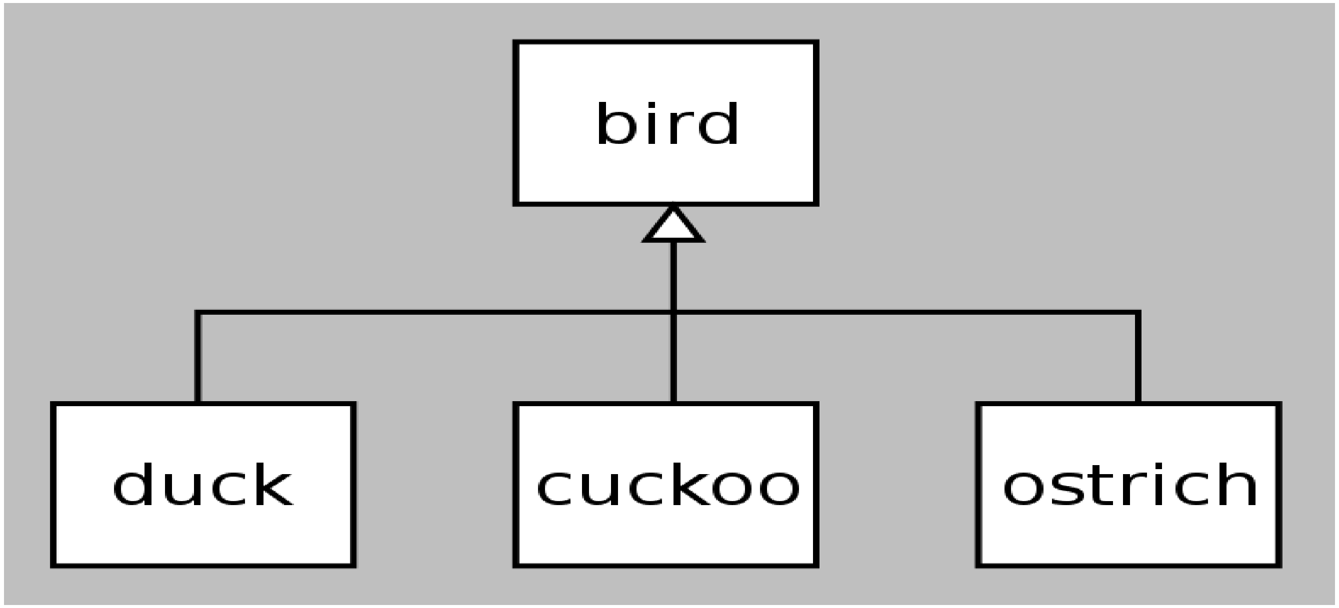
Object Decomposition:

Principle of Abstraction





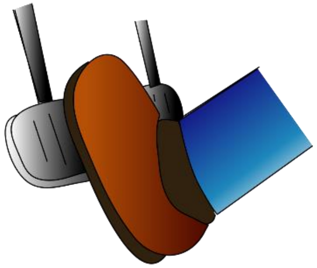
Inheritance and Polymorphism



Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Inheritance:
a vehicle hierarchy



sedan

van

jeep

Inheritance:

a vehicle hierarchy

switch on

turn

signal

accelerate

break

reverse

sedan

van

jeep

Inheritance:

a vehicle hierarchy

Inheritance allows us to *derive* new classes from existing classes.

sedan

van

jeep

Inheritance:

a vehicle hierarchy

contains all the attributes and behaviors common to all vehicles

vehicle

sedan

van

jeep

Inheritance:

a vehicle hierarchy

Derive new classes each to represent a specific type of vehicles!

vehicle

sedan

van

jeep

Inheritance:

…hicle hierarchy

The derived classes are specialized classes which inherit the properties and methods of the class they are derived from.

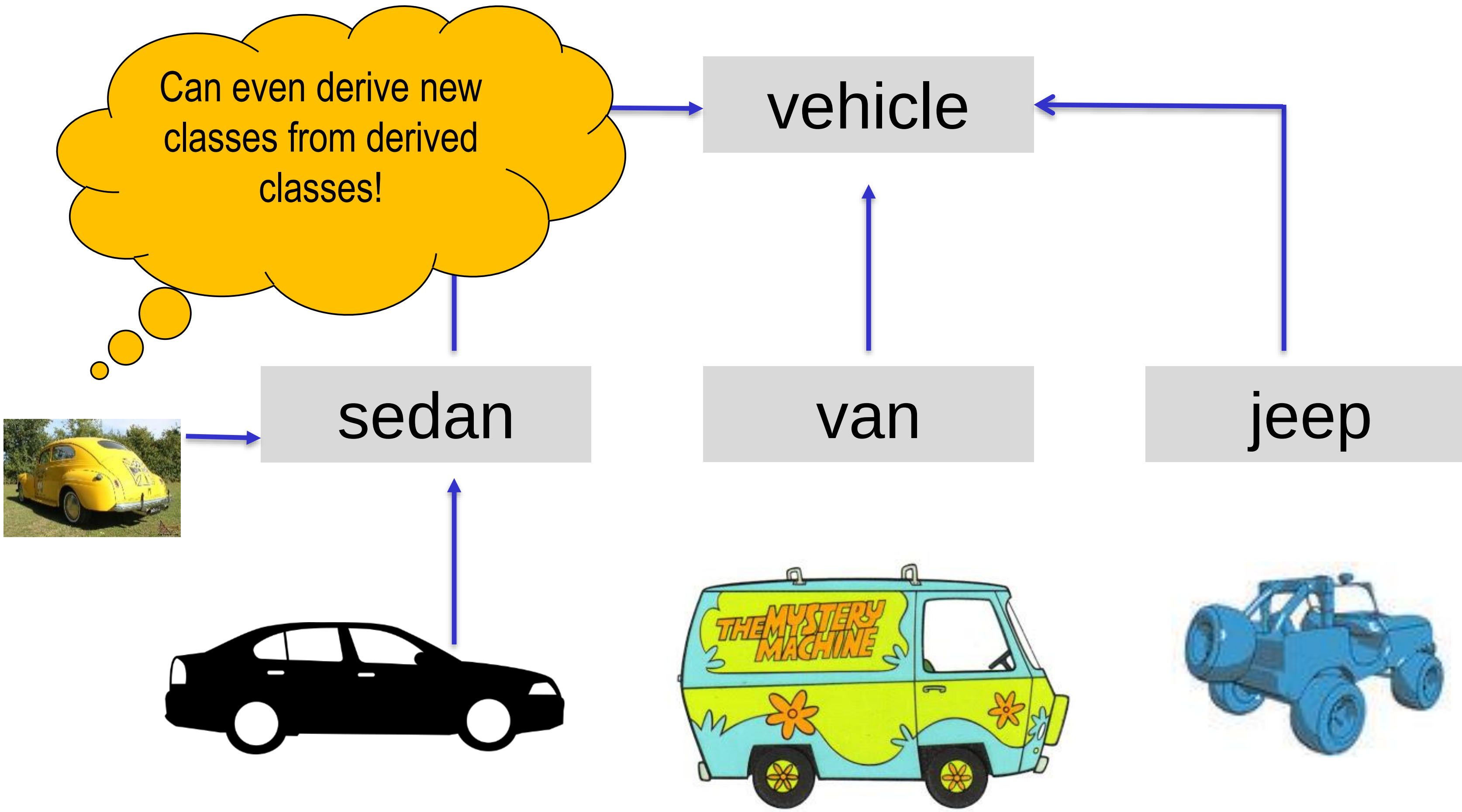
vehicle

sedan

van

jeep

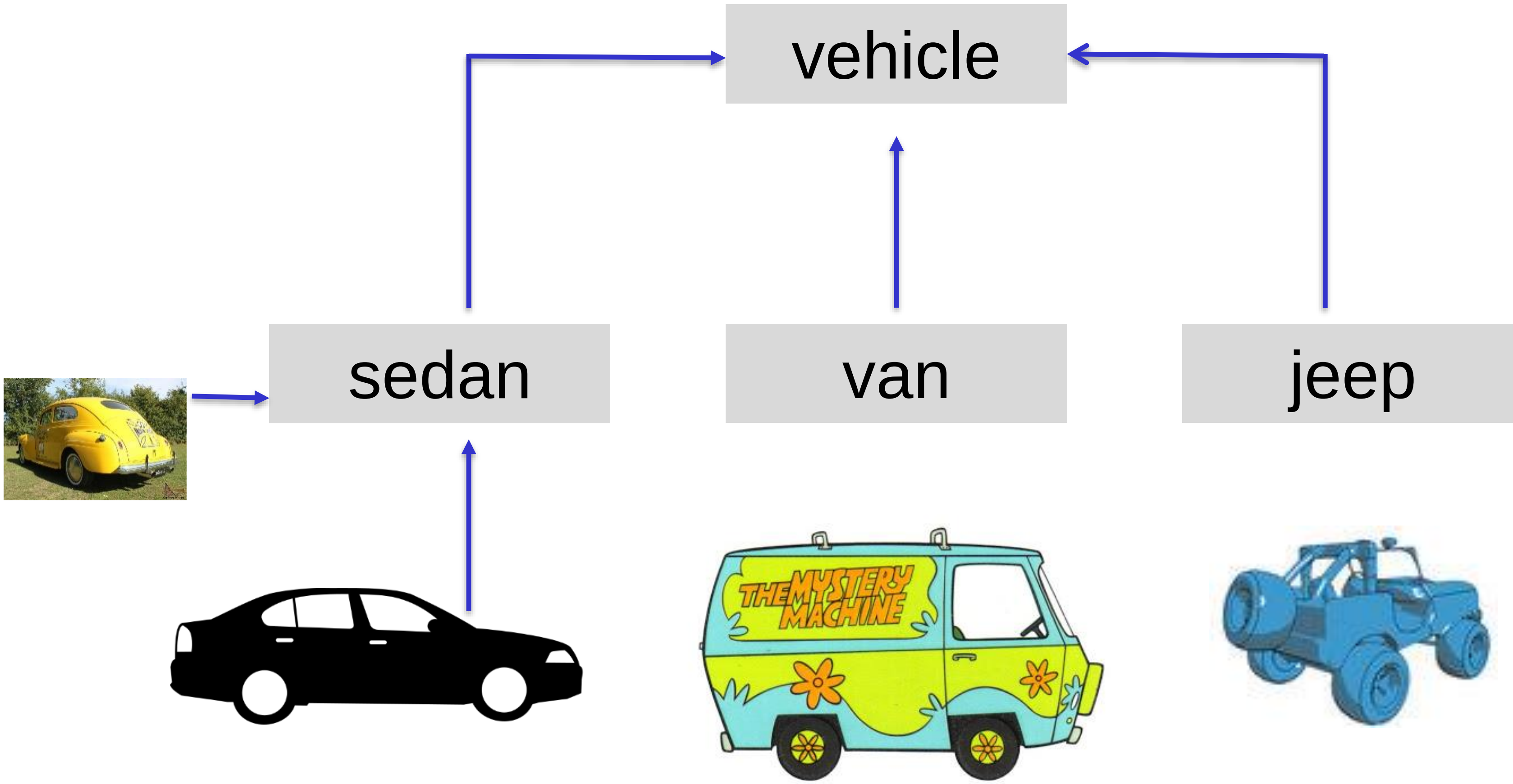
Inheritance:
a vehicle hierarchy



Inheritance:

a vehicle hierarchy

Inheritance represents a
parent .. child
heirarchy



Inheritance:

a vehicle hierarchy

Single Inheritance

A new class is derived from only one parent class.

Java only supports single Inheritance!

vehicle

sedan

van

jeep

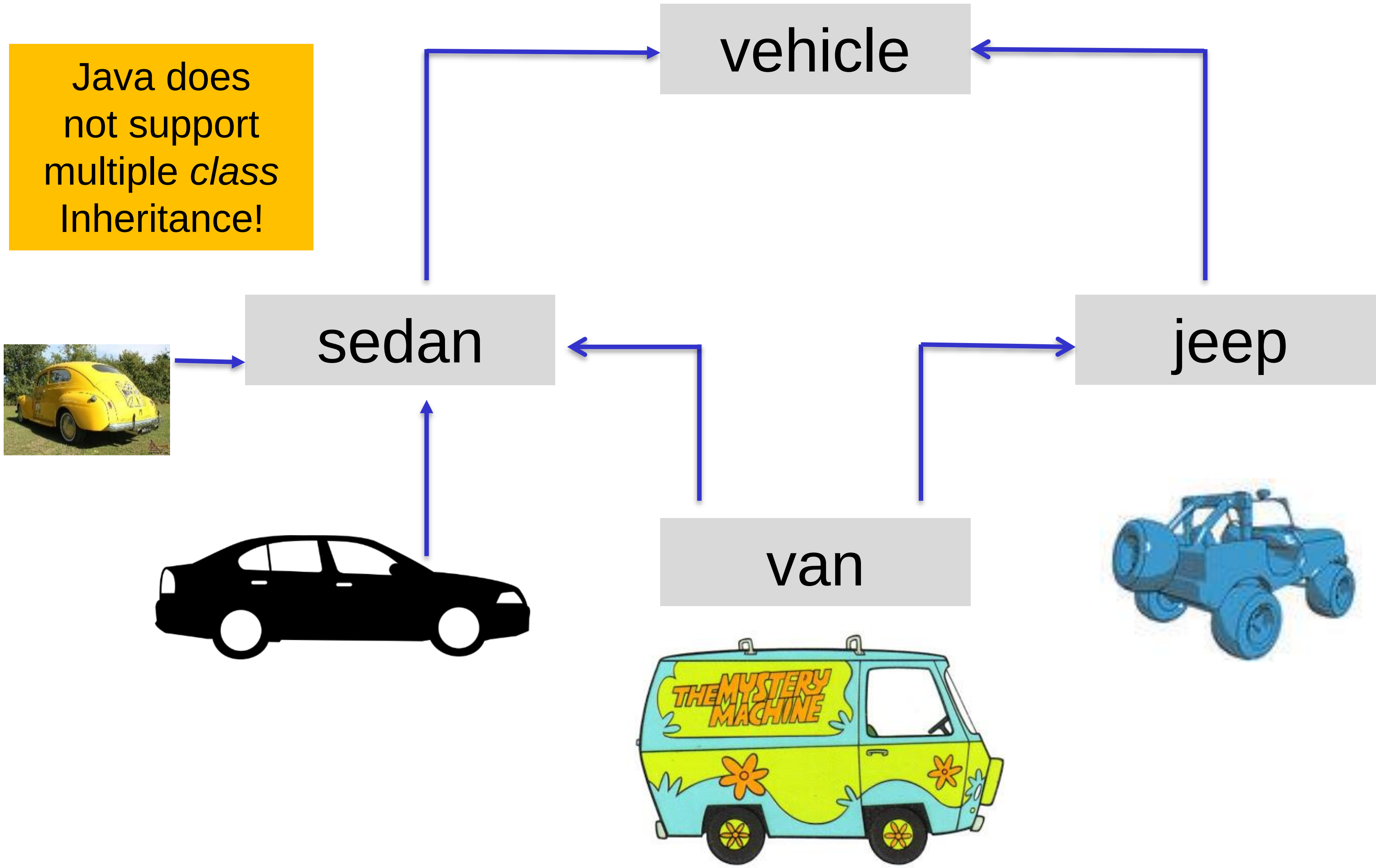
Inheritance:

a vehicle hierarchy

Multiple Inheritance

A new class is derived from more than one parent class.

Java does not support multiple *class* Inheritance!



Inheritance vs. Object Composition

- Object composition refers to the physical entities that make-up or *compose* the object. Example:

  - a vehicle has tires
  - a vehicle has a rear view mirror
  - a vehicle has a break pedal, etc.

Inheritance vs. Object Composition

- Object composition refers to the physi
make-up or *compose* the object. Exam

  - a vehicle has tires
  - a vehicle has a rear view mirror
  - a vehicle has a break pedal, etc.

Note the
“has a”
Relationship!

Inheritance vs. Object Composition

- Object composition refers to the physical entities that make-up or *compose* the object. Example:

    - a vehicle has tires
    - a vehicle has a rear view mirror
    - a vehicle has a break pedal, etc.

- Inheritance represents a hierarchical relationship. Example:

    - a sedan is a vehicle
    - a van is a vehicle
    - a jeep is a vehicle

Inheritance vs. Object Composition

- Object composition refers to the physical entities that make-up or *compose* the object. Example:

  - a vehicle has tires
  - a vehicle has a rear view mirror
  - a vehicle has a break pedal, etc.

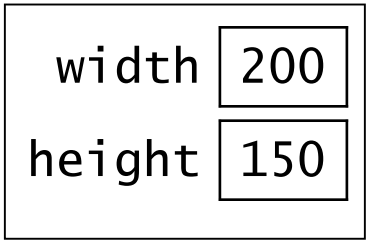
- Inheritance represents a hierarchical relationship. Example:

  - a sedan is a vehicle
  - a van is a vehicle
  - a jeep is a vehicle

Note the
"is a"
Relationship!

Recall: A Class for `Rectangle` Objects

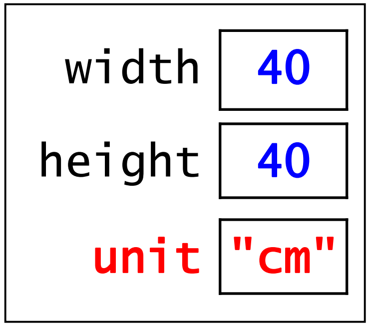
- Every `Rectangle` object has two fields:
  - `width`
  - `height`



- It also has methods inside it:
  - `grow()`
  - `area()`
  - `toString()`
  - etc.

Squares *are* Special Rectangles!

- A square also has a `width` and a `height`.
  - but the two values must be the same
- Assume that we also want `Square` objects to have a field for the unit of measurement.



Squares *are* Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same

width	40
height	40
unit	"cm"

- Assume that we also want `Square` objects to have a field for the unit of measurement.

- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();
```

- But there may be differences as well:

```
System.out.println(r);
```

output:

```
20 x 30
```

```
System.out.println(sq);
```

output:

```
square with 40-cm sides
```

Squares *are* Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same

width	40
height	40
unit	"cm"

- Assume that we also want `Square` objects to have a field for the unit of measurement.

- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();     // same computation
```

- But there may be differences as well:

```
System.out.println(r);
```

output:
`20 x 30`

```
System.out.println(sq);
```

output:
`square with 40-cm sides`

Squares *are* Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same

width	40
height	40
unit	"cm"

- Assume that we also want `Square` objects to have a field for the unit of measurement.

- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();
```

- But there may be differences as well:

```
System.out.println(r);
```

output:
`20 x 30`

```
System.out.println(sq);
```

output:
`square with 40-cm sides`

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square {
    int width, height;
    String unit;

    public Square(int side, String unit) {
        width = height = side;
        this.unit = unit;
    }
    public int area() {
        return width * height;
    }
    ...
```

Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square {
   int width, height;
   String unit;

   public Square(int side, String unit) {
       width = height = side;
       this.unit = unit;
   }
   public int area() {
       return width * height;
   }
   ...
```

Using Inheritance

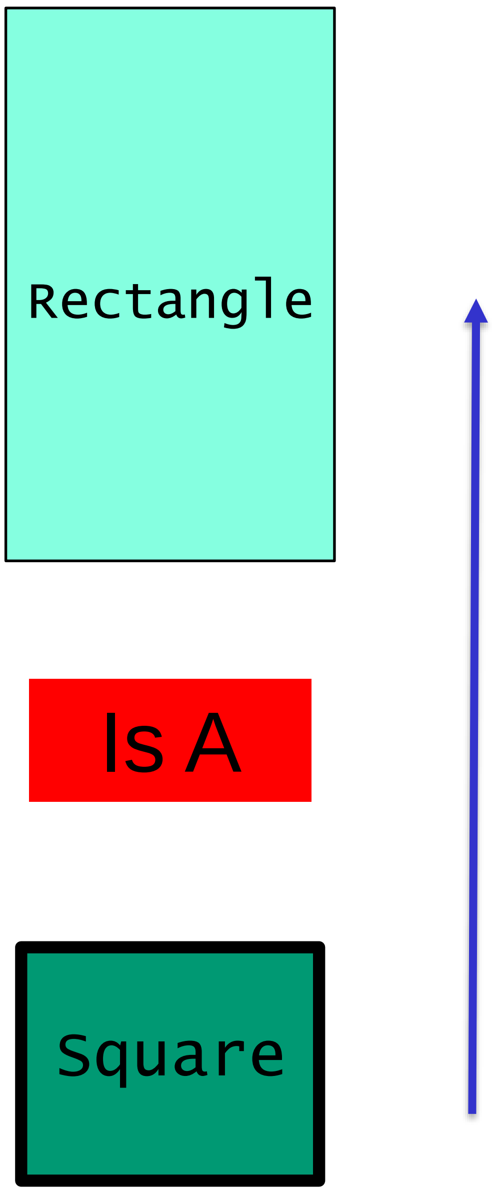
```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square {
   int width, height;
   String unit;

   public Square(int side, String unit) {
       width = height = side;
       this.unit = unit;
   }
   public int area() {
       return width * height;
   }
   ...
```

Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

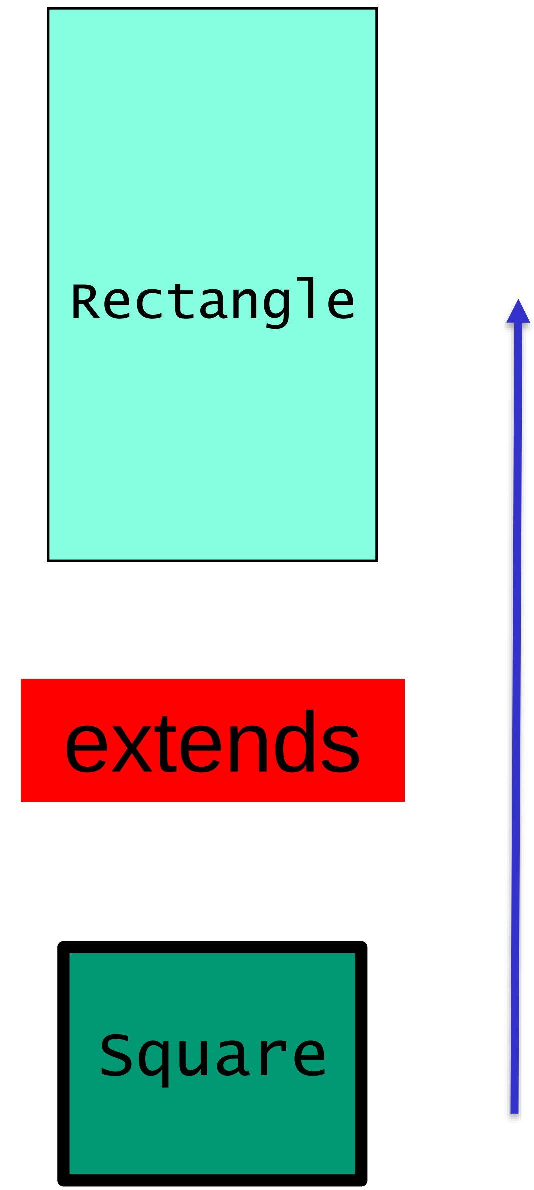
```
public class Square {
   int width, height;
   String unit;

   public Square(int side, String unit) {
       width = height = side;
       this.unit = unit;
   }
   public int area() {
       return width * height;
   }
   ...
```



Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

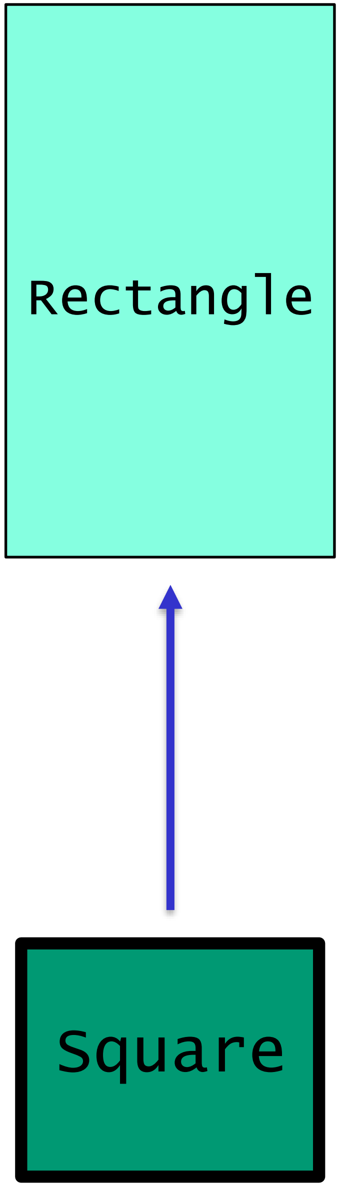
```
public class Square {
   int width, height;
   String unit;

   public Square(int side, String unit) {
       width = height = side;
       this.unit = unit;
   }
   public int area() {
       return width * height;
   }
   ...
```



Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

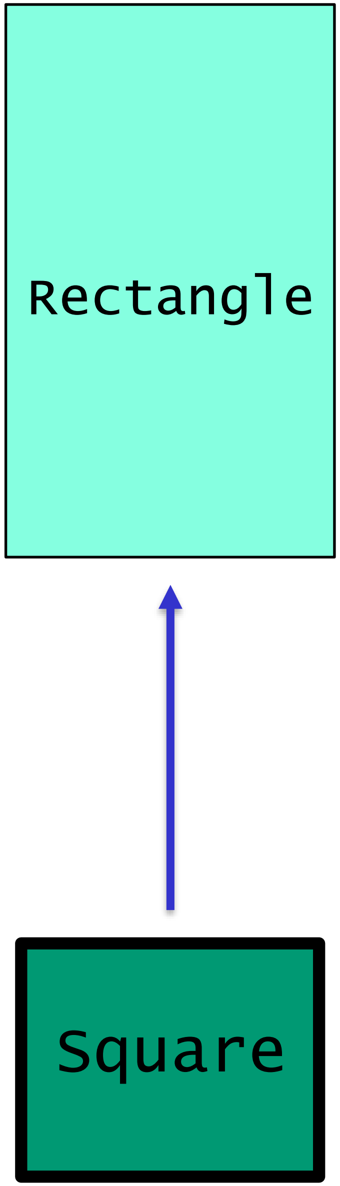
```
public class Square {
   int width, height;
   String unit;

   public Square(int side, String unit) {
       width = height = side;
       this.unit = unit;
   }
   public int area() {
       return width * height;
   }
   ...
```



Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       // initialize data members


   }

   // inherits other methods
}
```

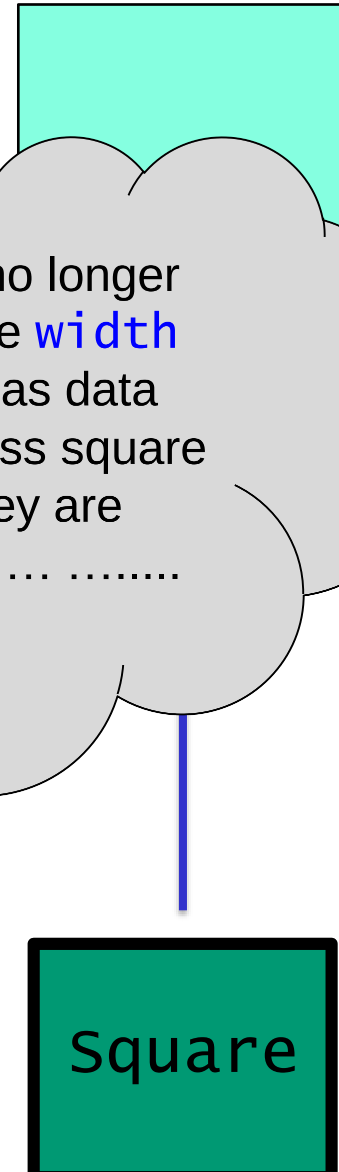


Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       // initialize data members


   }

   // inherits other methods
}
```

Note that we no longer have to include `width` and `height` as data members of class square because they are inherited from … ……..

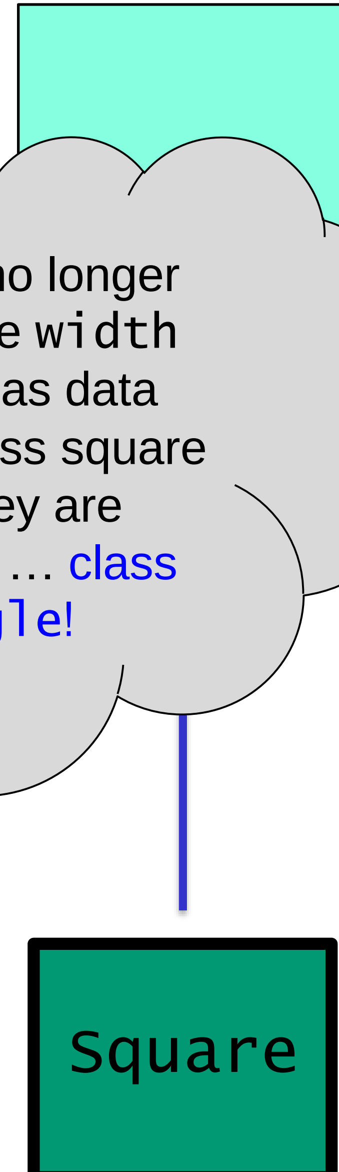


Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int      {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectang
   String unit;

   public Square(int side, String unit) {
       // initialize data members


   }

   // inherits other methods
}
```

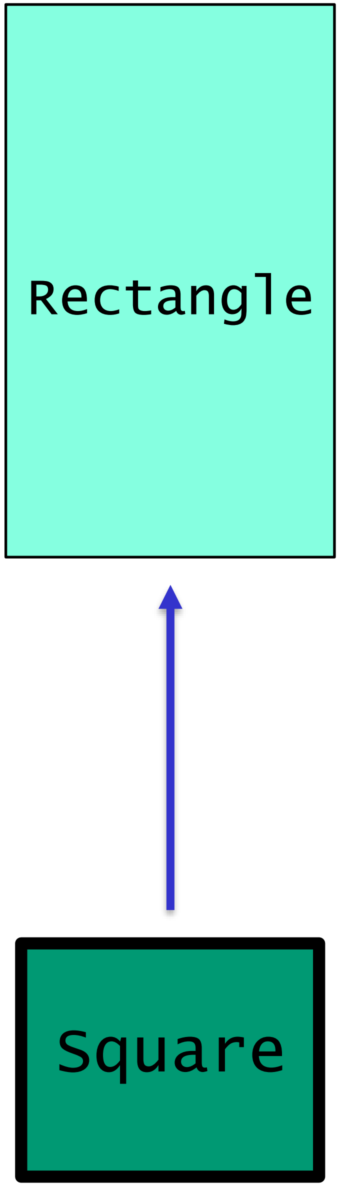
Note that we no longer have to include `width` and `height` as data members of class square because they are inherited from … class `Rectangle`!



Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       // initialize data members


   }

   // inherits other methods
}
```

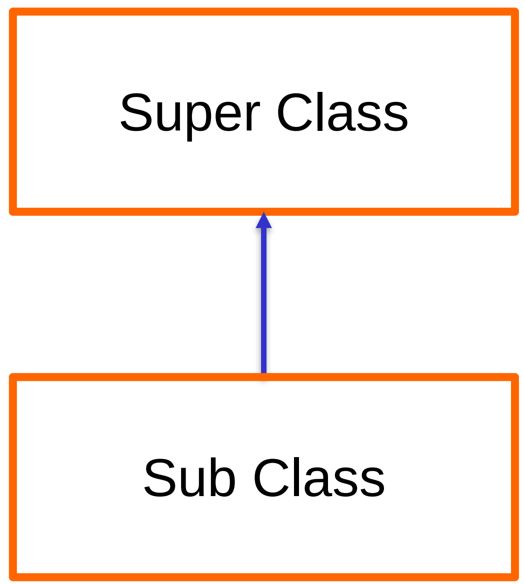


Using Inheritance

- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!

- `Square` is a *subclass* of `Rectangle`.

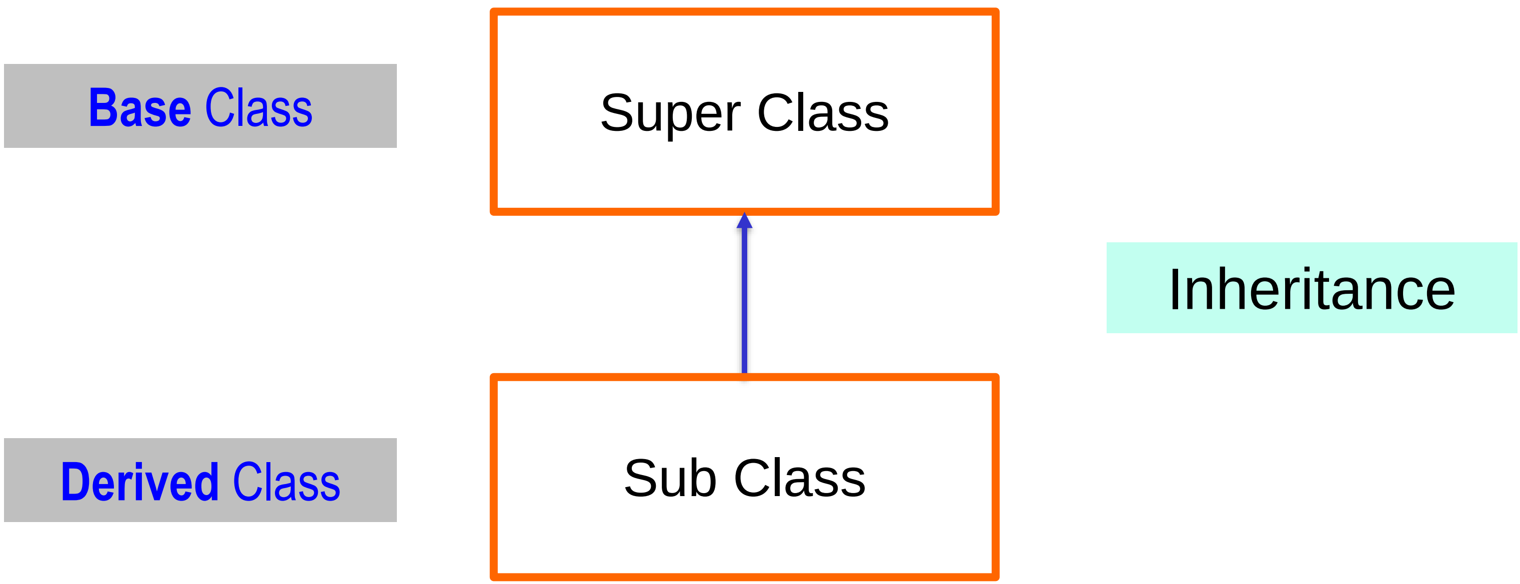
- `Rectangle` is a *superclass* of `Square`.



Inheritance

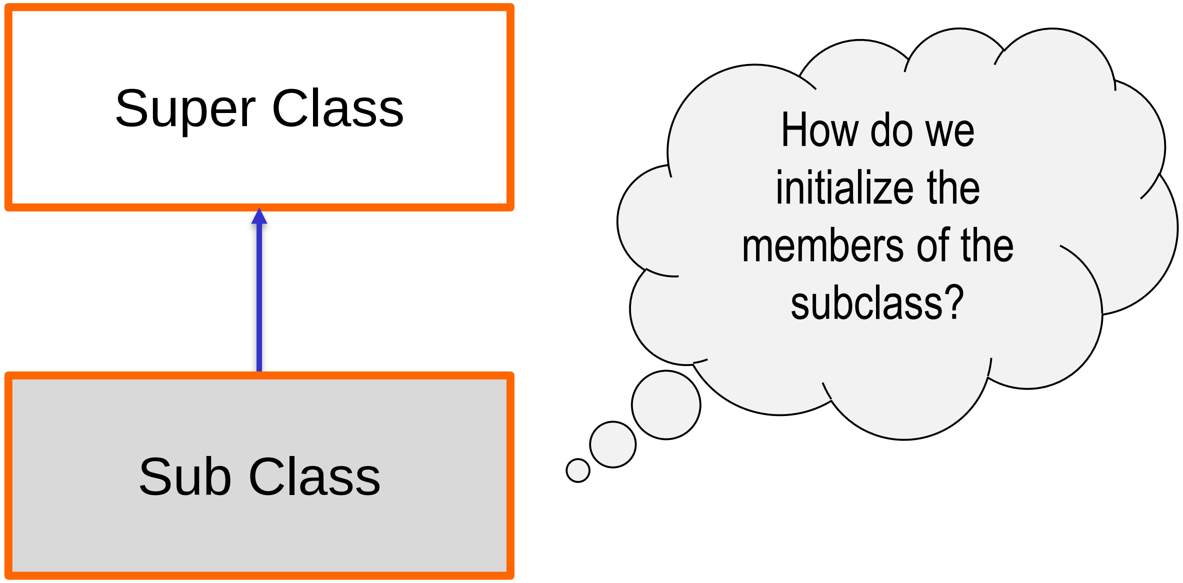
Using Inheritance

- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.



Using Inheritance

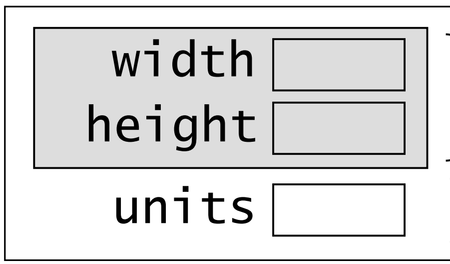
- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.



Encapsulation and Inheritance

- A subclass has direct access to the *public* fields and methods of a superclass.
  - it ***cannot*** access its *private* fields and methods

- Example: we can think of a `Square` object as follows:

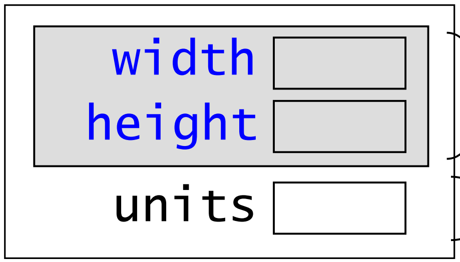


private fields inherited from `Rectangle`;
cannot be accessed directly by `Square` methods

field defined in `Square`;
it *can* be accessed directly by `Square` methods

Encapsulation and Inheritance

- A subclass has direct access to the *public* fields and methods of a superclass.
  - it ***cannot*** access its *private* fields and methods
- Example: we can think of a `Square` object as follows:



private fields inherited from `Rectangle`;
cannot be accessed directly by `Square` methods

field defined in `Square`;
it *can* be accessed directly by `Square` methods

Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       width = height = side;


   }

   // inherits other methods
}
```

As `width` and `height` are *private* data member of the superclass Rectangle, we cannot directly access them here!

Using Inheritance

```
public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       this.width = this.height = side;


   }

   // inherits other methods
}
```

As `width` and `height` are *private* data member of the superclass Rectangle, we cannot directly access them here!

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...   // other methods
    public int area() {
        return width * height;
    }
}
```

As width and height are *private* data member of the superclass Rectangle, we cannot directly access them here!

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        this.width = this.height = side;


    }

    // inherits other methods
}
```

Encapsulation and Inheritance

- Change the modifier in the super class from private to protected.
- The protected modifier allows the fields to remain private within the class they are defined in but allows them to be accessible to all subclasses.
- But for the most part it is more prudent to use the public accessor and mutator methods of the super class – even within the subclass.

Using Inheritance:
option #1

```
public class Rectangle {
    protected width;
    protected height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        width = height = side;
        this.unit = unit;

    }

    // inherits other methods
}
```

But this would bypass the validation that should be performed to ensure that we are not making a bad assignment!

Using Inheritance:

option #1

```
public class Rectangle {
   protected width;
   protected height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       setWidth(side);
       setHeight(side);
       this.unit = unit;

   }
   // inherits other methods
}
```

We could invoke public *mutator* methods of the `Rectangle` class, but ...

Using Inheritance:

option #1

```
public class Rectangle {
   protected width;
   protected height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       setWidth(side);
       setHeight(side);
       this.unit = unit;

   }
   // inherits other methods
}
```

We could invoke public *mutator* methods of the `Rectangle` class, but ... We are already doing this in the Rectangle constructor!

Using Inheritance:

option #2

```
public class Rectangle {
   private width;
   private height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       setWidth(side);
       setHeight(side);
       this.unit = unit;
   }

   // inherits other methods
}
```

As we are initializing members of the superclass, it is most appropriate to have the constructor of the superclass initialize them.

Using Inheritance:

option #3

```
public class Rectangle {
   private width;
   private height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

As we are initializing members of the superclass, it is most appropriate to have the constructor of the superclass initialize them.

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       super(side, side);

       this.unit = unit;
   }

   // inherits other methods
}
```

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        super(side);

        this.unit = unit;
    }

    // inherits other methods
}
```

As we are initializing members of the superclass, it is most appropriate to have the constructor of the superclass initialize them.

Using Inheritance:

option #3

```
public class Rectangle {
   private width;
   private height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {
       super(side);

       this.unit = unit;
   }

   // inherits other methods
}
```

Note that the call to the superclass constructor must be the very first statement in the body of the subclass constructor.

Using Inheritance:

option #3

```
public class Rectangle {
   private width;
   private height;

   public Rectangle(int w, int h) {
       setWidth(w);
       setHeight(h);
   }
   ...  // other methods
   public int area() {
       return width * height;
   }
}
```

```
public class Square extends Rectangle {
   String unit;

   public Square(int side, String unit) {



       this.unit = unit;
   }

   // inherits other methods
}
```

What would happen if we remove the **explicit** call to the constructor of the superclass?

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // no-arg constructor
        super();
        this.unit = unit;
    }

    // inherits other methods
}
```

The Java compiler would add a call to the *no-argument* constructor of the superclass!

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // no-arg constructor
        super();
        this.unit = unit;
    }

    // inherits other methods
}
```

If a *no-arg constructor* has not been defined in the superclass Java would issue a compiler error.

A note about Constructors

If a class does not define *any* constructors the Java compiler will create a no-argument constructor for our class. This constructor will be used if we create an object without passing any arguments.

```
Rectangle r = new Rectangle();
```

However once we define any constructor, then it is up to the class to define a no-argument constructor should we want to allow objects to be created with just default values.

A note about Constructors

If a class does not define *any* constructors the Java compiler will create a no-argument constructor for our class. This constructor will be used if we create an object without passing any arguments.

```
Rectangle r = new Rectangle();
```

However once we define any constructor, then it is up to the class to define a no-argument constructor should we want to allow objects to be created with just default values.

A note about Constructors

If a class does not define *any* constructors the Java compiler will create a no-argument constructor for our class. This constructor will be used if we create an object without passing any arguments.

Rectangle r = new Rectangle();

However once we define any constructor, then it is up to the class to define a no-argument constructor should we want to allow objects to be created with just default values.

Constructor Chaining

Unlike the methods of a class, constructors of a superclass are not inherited by the subclass.

They can only be invoked from the constructors of the subclass using the keyword super.

Constructor Chaining

Unlike the methods of a class, constructors of a superclass are not inherited by the subclass.

They can only be invoked from the constructors of the subclass using the keyword super.

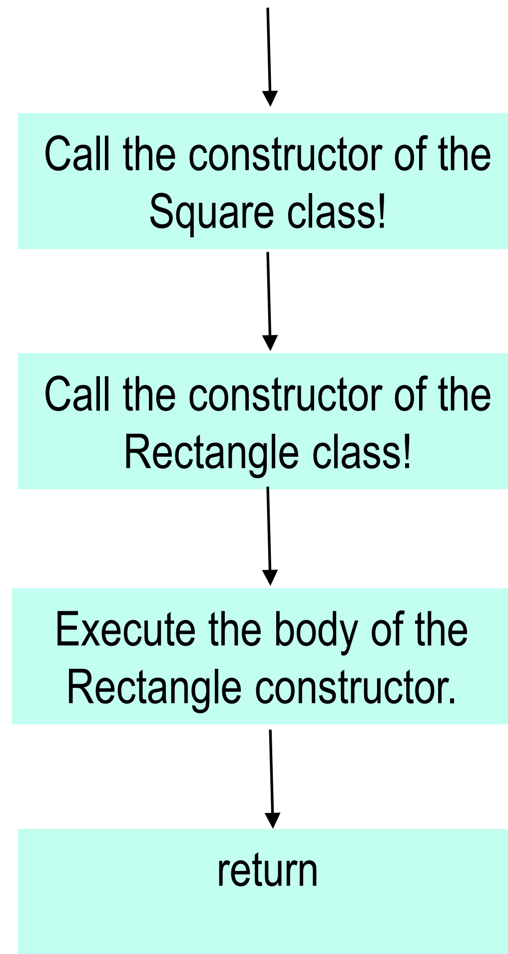
Constructing an instance of a class invokes the constructors of all the super classes along the inheritance chain.

Constructor Chaining

Square r = **new** Square(10, “cm”);

Constructor Chaining

Square r = new **Square(10, “cm”);**



Constructor Chaining

Square r = new **Square(10, “cm”);**

Call the constructor of the Square class!

Execute the remaining body of the Square constructor.

Overriding *Inherited* Methods