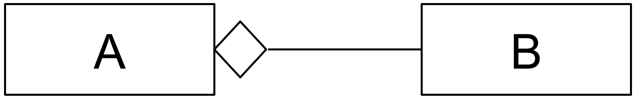
EEСS 1720
Building Interactive Systems

Lecture 9 :: Object Hierarchies 1

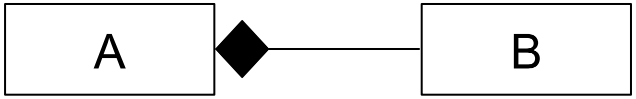
- Class Relationships (IS-A)
- Interfaces/Inheritance (intro)

Recall

- 2 types of relationships between classes (HAS-A, IS-A)
  - HAS-A
    - When one class (reference type), “has” one or more fields that are of another class (reference type)
    - Defines an “association” between the two classes (2 types):
      - “loosely associated” (AGGREGATION)
      - A maintains *aliases* to existing B object(s)



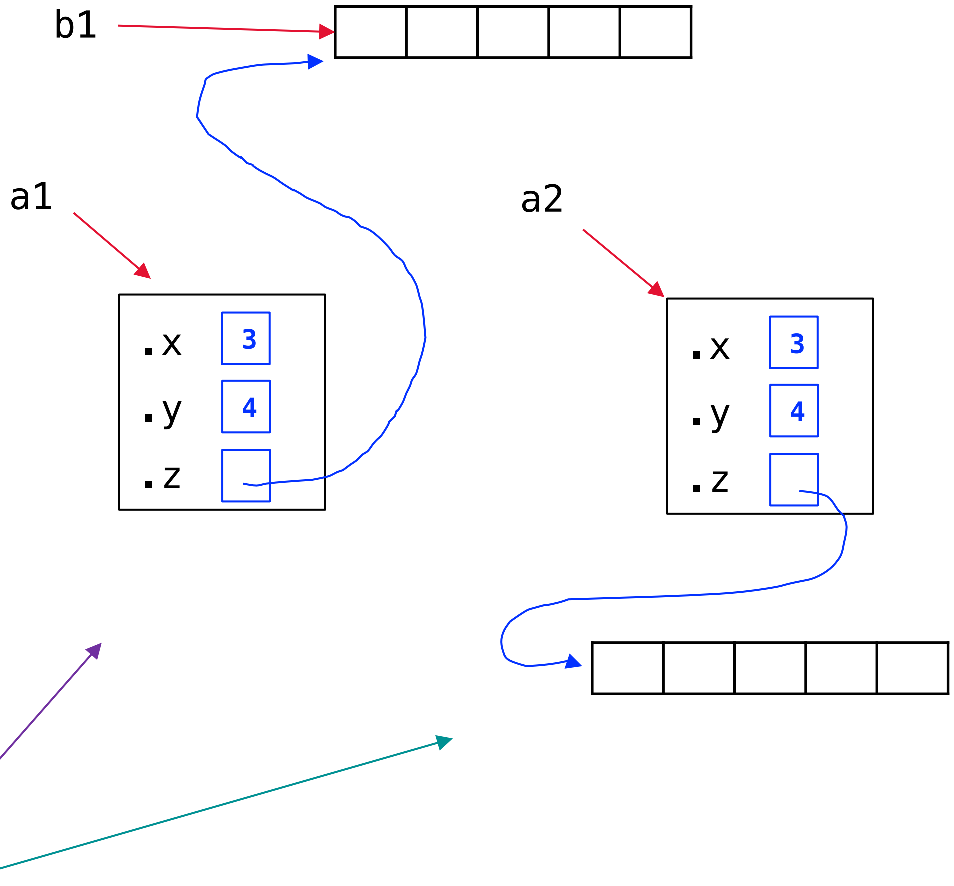
      - “strongly associated” (COMPOSITION)
      - A maintains its own *copies* of existing B objects(s)



Recall

- 2 types of HAS-A

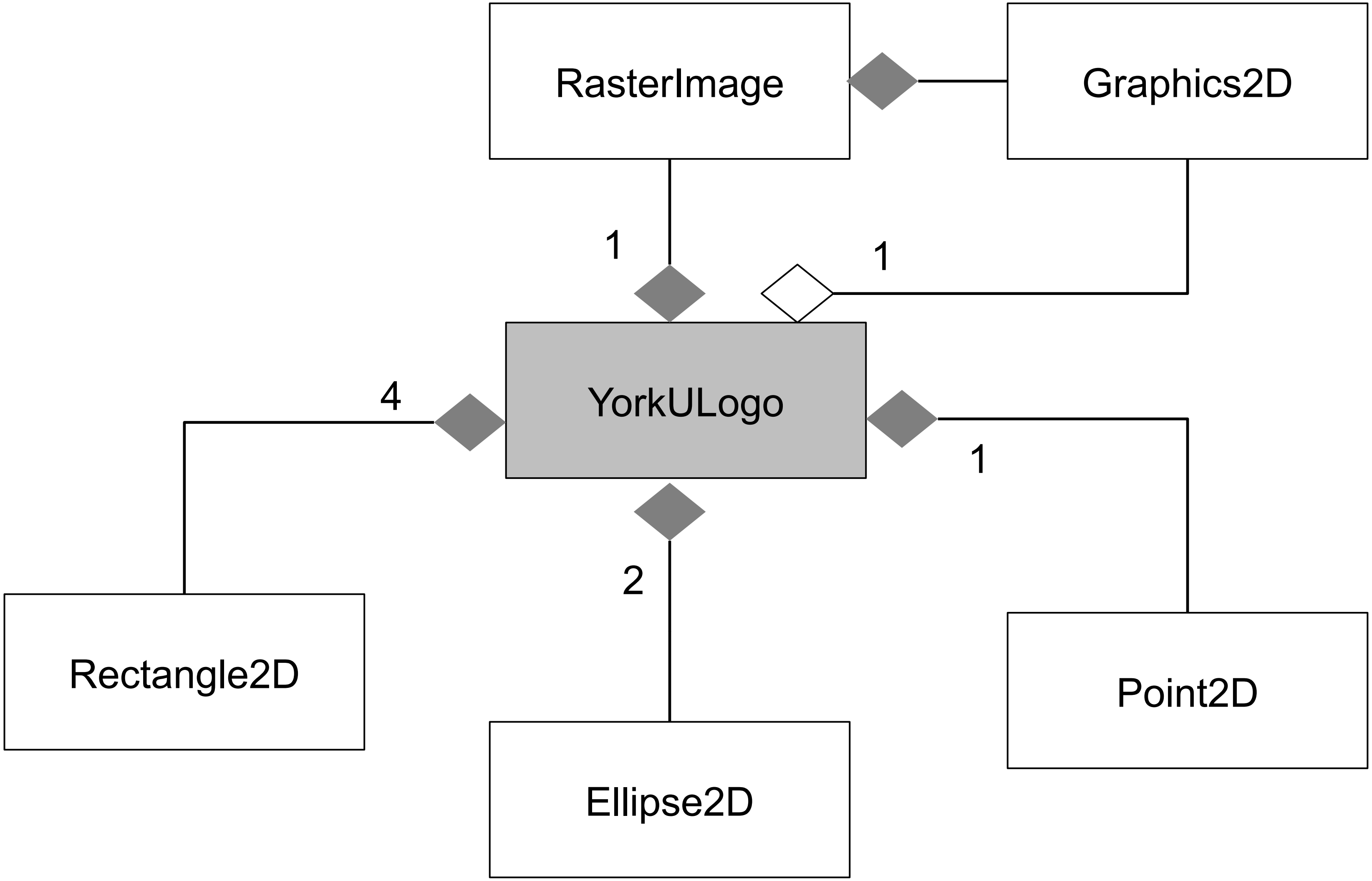
```
e.g. (class A)
- x : int
- y : int
- z : B[]
+ A()
+ A(int,int,B[])

+ getZ() : B[]
+ setZ(B)
```

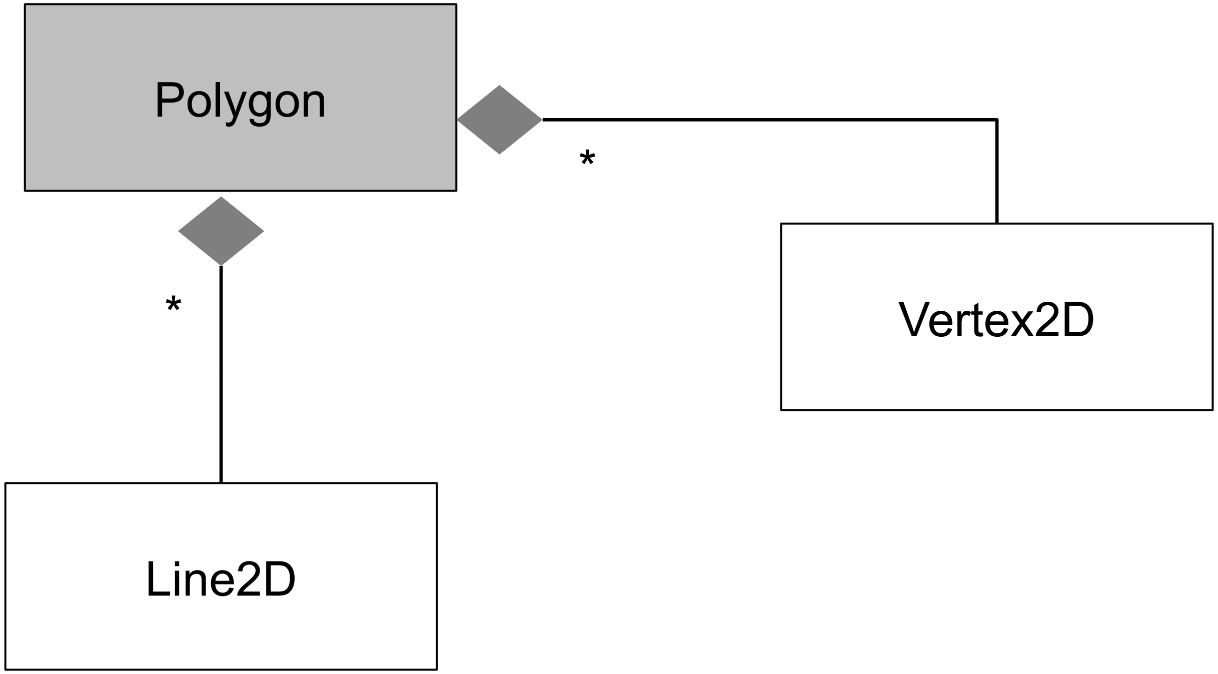


```
B[] b1 = new B[5];
// init elements of b1

A a1 = new A(3,4,b1); // aggregation

A a2 = new A(3,4,b1); // composition
```

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;
import imagePackage.RasterImage;

public class YorkULogo {

    // fields
    private RasterImage img;
    private Graphics2D gfx;
    private Point2D position;
    private Rectangle2D bounds;
    private Rectangle2D backgnd;
    private Rectangle2D whiteRect;
    private Ellipse2D whiteCircle;
    private Rectangle2D redRect;
    private Ellipse2D redCircle;

    // ctors
    public YorkULogo(int x, int y, int w, int h) {

        img = new RasterImage(640,480);
        gfx = img.getGraphics2D();
        this.position = new Point2D.Double(x,y);
        this.bounds = new Rectangle2D.Double(x, y, w, h);
        this.backgnd = new Rectangle2D.Double(x, y, w, h);
        this.whiteRect = new Rectangle2D.Double(x + 0.4*w , y, 0.6*w, 0.5*h);
        this.whiteCircle = new Ellipse2D.Double(x + 0.4*w , y + 0.5*h - 0.3*w,
                                                                      0.6*w,0.6*w);

        this.redRect = new Rectangle2D.Double(x + 0.6*w, y, 0.2*w, 0.5*h);
        this.redCircle = new Ellipse2D.Double(x + 0.6*w, y + 0.5*h - 0.1*w ,
                                                                   0.2*w,0.2*w);
    }
    // …
```

YU logo example:
YorkULogo object maintains
“has-a” relationship(s) with
Rectangle2D, Ellipse2D
Point2D, RasterImage and
Graphics2D object(s)


RasterImage
Graphics2D
1
1
YorkULogo
4
1
2
Rectangle2D
Ellipse2D
Point2D



** used if many (e.g. array/ArrayList of Point2D)*

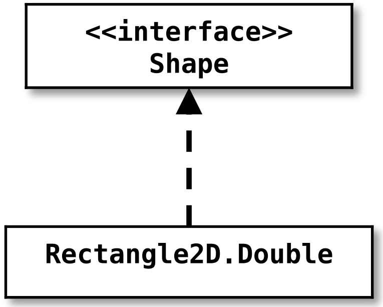
IS-A Relationships

Hierarchical relationships between classes/objects

2 types of “IS-A” relationship

Interface

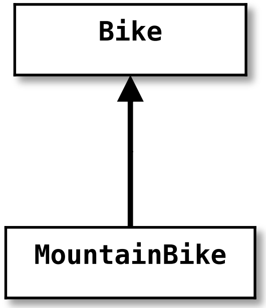
- hierarchical relationship between a special type (interface) and classes



Rectangle2D.Double “is-a” Shape

Inheritance

- hierarchical relationship between classes



MountainBike “is-a” Bike

Differences (high level)

Interface

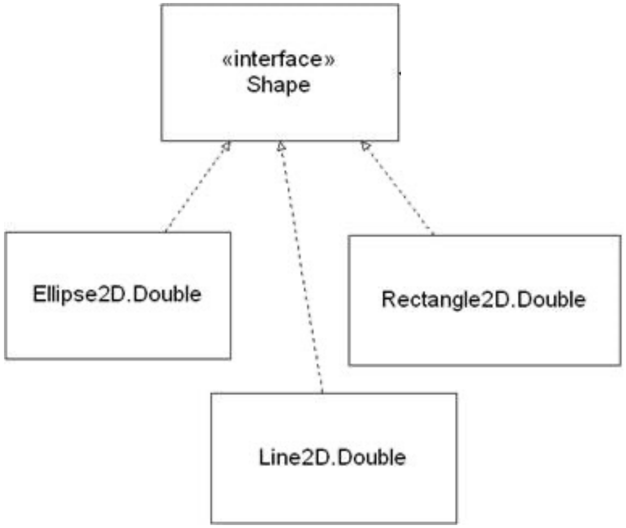
- abstracts behaviour of a category of types
- specifies behaviours (by declaring method headers)
- does not define the method internals
- CANNOT INSTANTIATE EVER!
- sub-types that are classes MUST ”implement” all declared methods from the interface

Inheritance

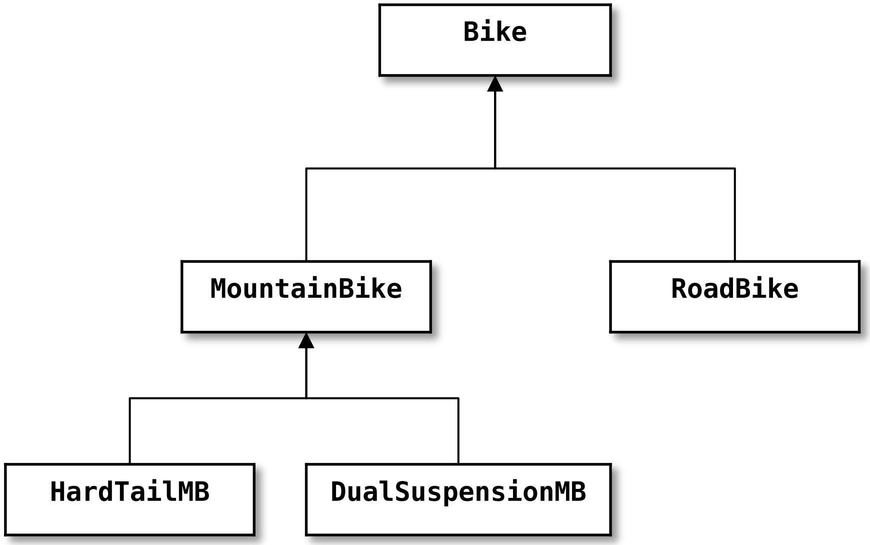
- a class that abstracts both properties (fields) and behaviour (methods) of a category of types
- can choose to define or partially define these (i.e. have a full implementation)
- If partially defined, cannot instantiate
- If fully defined, can instantiate
- sub-types (other classes in the category) can “extend”

What is meant by “IS-A” ?

Interfaces



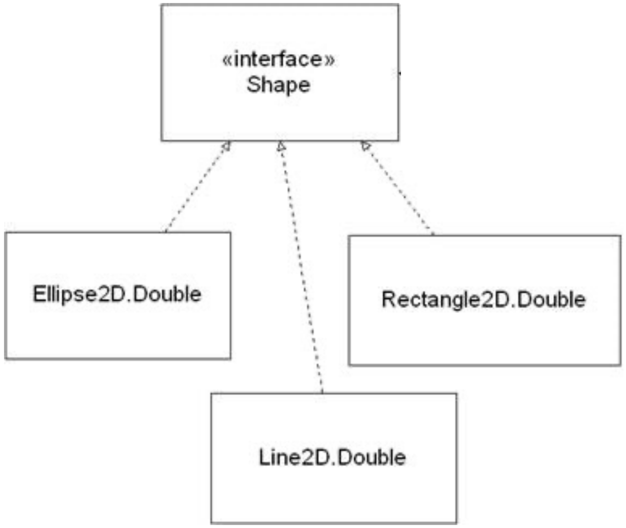
Inheritance



Is-a = “is substitutable for”

What is meant by “IS-A” ?

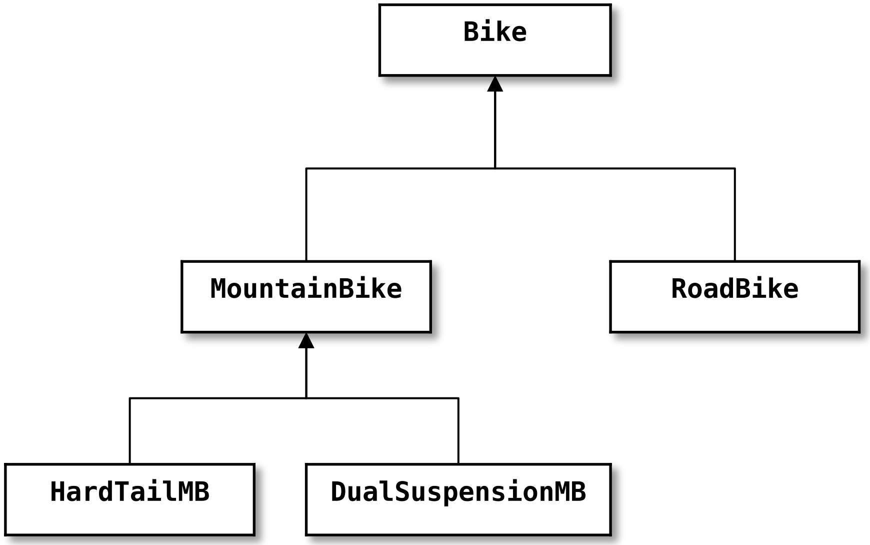
Interfaces



Shape references can be assigned :

Rectangle2D.Double objects
Ellipse2D.Double objects
Line2D.Double objects

Inheritance

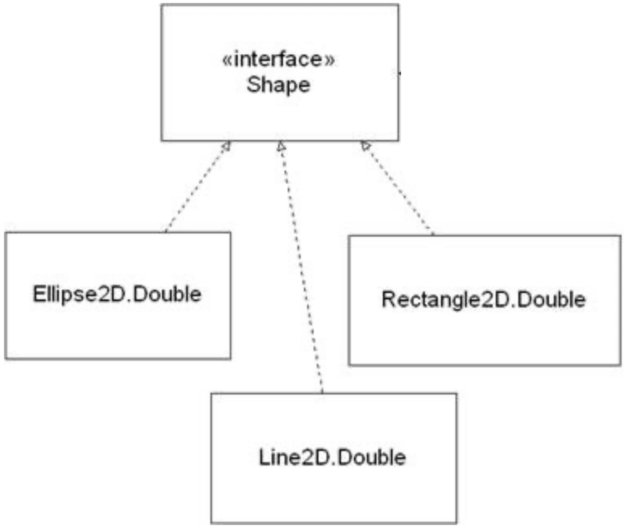


Bike references can be assigned:

Bike, MountainBike, RoadBike,
HardTailMB or DualSuspensionMB objects

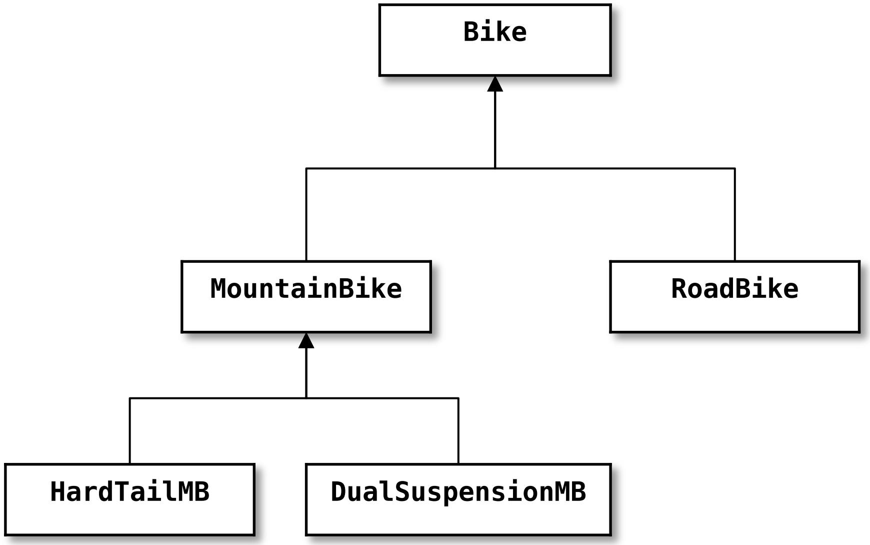
What is meant by “IS-A” ?

Interfaces



Rectangle2D.Double “is-a” Shape
Ellipse2D.Double “is-a” Shape
Line2D.Double “is-a” Shape

Inheritance



MountainBike ”is-a” Bike
RoadBike “is-a” Bike
HardTailMB “is-a” MountainBike; & “is-a” Bike
DualSuspensionMB “is-a” MountainBike; & ”is-a” Bike

Differences (high level)

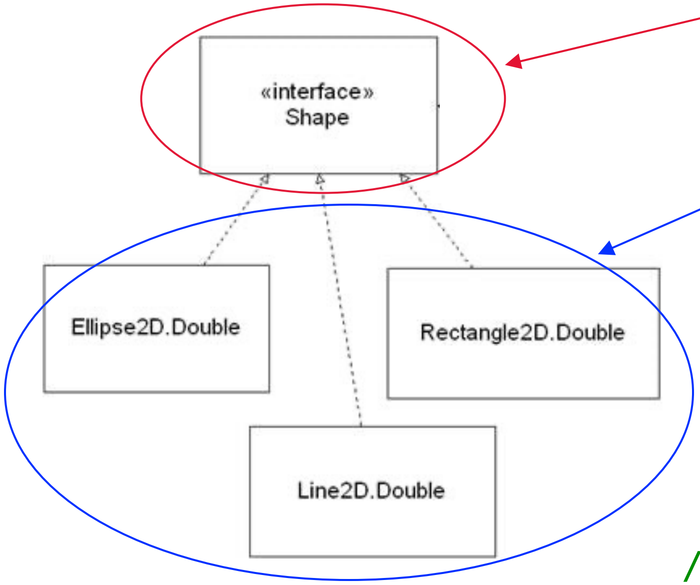
Interface

- Parent (super-type):
  - abstracts behaviour of a category of types
  - specifies behaviours (by defining method headers)
  - does not define the method internals
- Children (sub-types):
  - classes that are declared to “implement” the super-type
  - “implement” means they have a full definition of all methods indicated by the parent interface

Inheritance

- Parent (super-type):
  - a class that abstracts both properties (fields) and behaviour (methods) of a category of types
  - can choose to define or partially define these (i.e. have a full implementation)
- Children (sub-types):
  - Classes that are declared to “extend” the super-type class
  - “extend” means they assume all the properties from the parent
  - may choose to re-define or add new fields/methods

Shapes

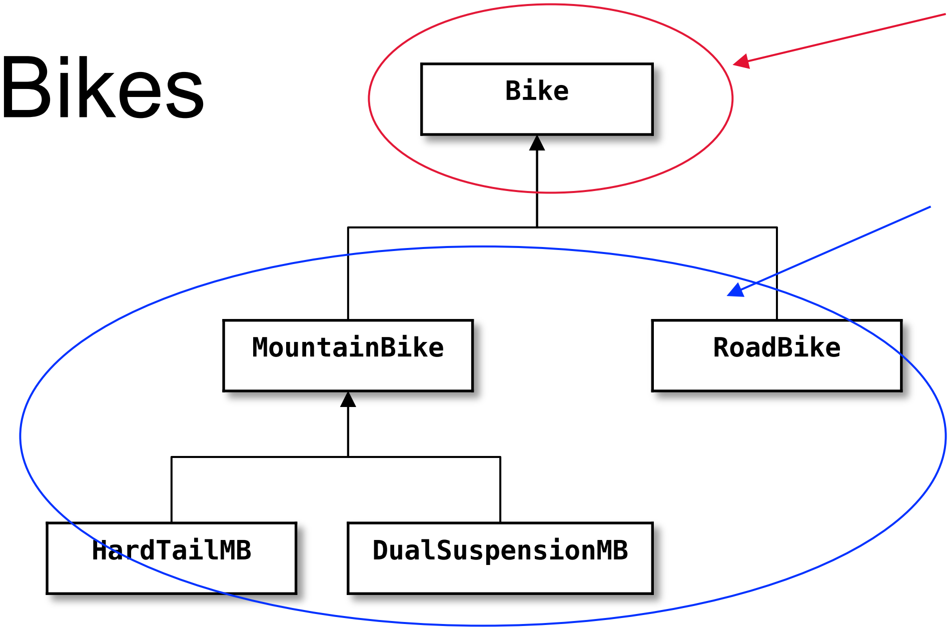


parent (super-type) → interface

children (sub-types) → classes

```
// with an interface, this is possible:

Rectangle2D.Double r = new Rectangle2D.Double();
Ellipse2D.Double e = new Ellipse2D.Double();

Shape s;
s = r;
s = e;

// etc
Shape anotherShape = new Line2D.Double();
```

Bikes

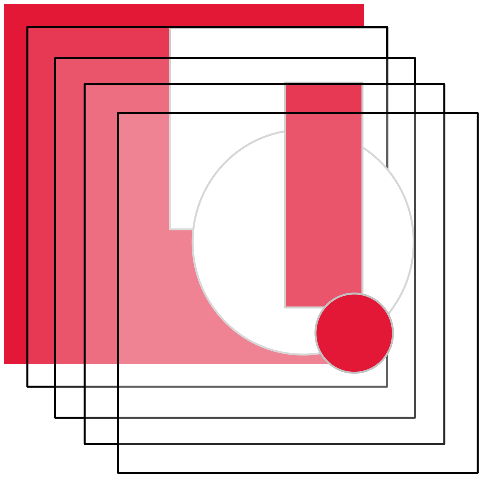


parent (super-type) → class

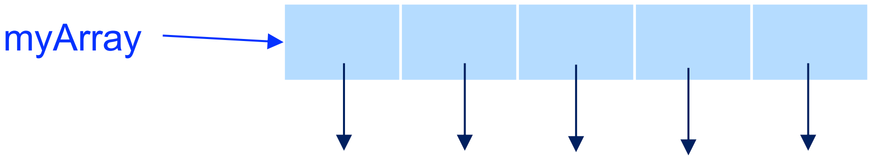
children (sub-types) → classes

```
// with inheritance, this is possible:

MountainBike mb  = new MountainBike();
RoadBike rb      = new RoadBike();

Bike b;
b = mb;
b = rb;

// etc

Bike anotherBike = new DualSuspensionMB();
```

How would we put the parts of YorkULogo into a single array?



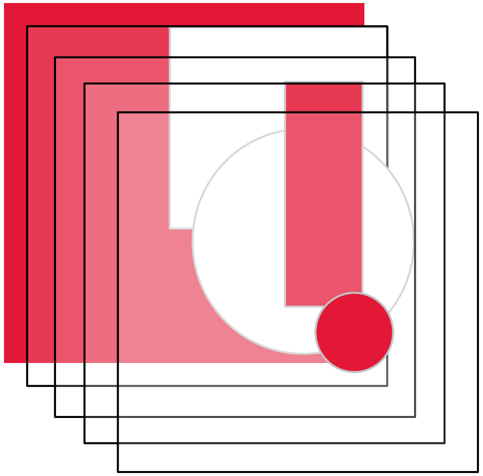
```
                    YorkULogo
// fields
–  backgnd     : Rectangle2D.Double
–  whiteRect   : Rectangle2D.Double
–  whiteCircle : Ellipse2D.Double
–  redRect     : Rectangle2D.Double
–  redCircle   : Ellipse2D.Double

// constructors & methods (not shown)
```



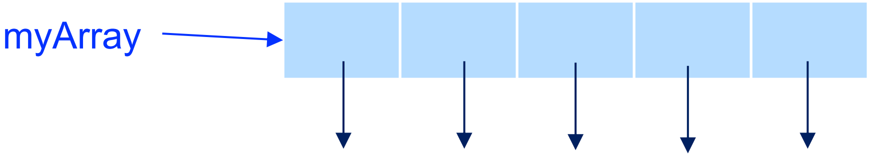
XX = ?

```
XX    myArray   = new XX[5];
ArrayList<XX> myList = new ArrayList<XX>();
```

How would we put the parts of YorkULogo into a single array?



```
YorkULogo

// fields
–  backgnd     : Rectangle2D.Double
–  whiteRect   : Rectangle2D.Double
–  whiteCircle : Ellipse2D.Double
–  redRect     : Rectangle2D.Double
–  redCircle   : Ellipse2D.Double

// constructors & methods (not shown)
```



XX = Shape

```
Shape myArray  = new Shape[5];
ArrayList<Shape> myList = new ArrayList<Shape>();
```

```
// inside constructor for YorkULogo, can create
// an array of Shapes  (as arrays must be a single type only)

this.logoParts = new Shape[5];

this.logoParts[0] = this.backgnd;
this.logoParts[1] = this.whiteRect;
this.logoParts[2] = this.whiteCircle;
this.logoParts[3] = this.redRect;
this.logoParts[4] = this.redCircle;
```

logoParts is an array of Shape references

each Shape reference in logoParts array can be assigned any object that has an “is-a” relationship with Shape

”is-a” == “is substitutable for”

- A reference type can be **<u>substituted</u>** with any instance of any class that is considered a ”subtype” in that same hierarchy
  - A Shape reference can be substituted with an object of any class that “implements” the Shape interface
  - A Bike reference can be substituted with an object of any class it is considered an ancestor to (i.e. any child, grandchild, etc)

- Similarly, a *method* can have its argument **<u>substituted</u>** with any instance of a class that is considered to have an *“is-a”* relationship with that argument type
  - The draw(..) method accepts a Shape argument, thus we can pass any object of a class that has an “is-a” relationship with Shape

```
// inside constructor for YorkULogo, can create
// an array of Shapes  (array must be of a single type only)

this.logoParts = new Shape[5];

this.logoParts[0] = this.backgnd;
this.logoParts[1] = this.whiteRect;
this.logoParts[2] = this.whiteCircle;
this.logoParts[3] = this.redRect;
this.logoParts[4] = this.redCircle;


// inside drawLogo, can now refer the draw method
// to a Shape type (logoParts[i])

this.gfx.setColor(Color.black);

for (int i=0; i<this.logoParts.length; i++) {

       this.gfx.draw(logoParts[i]);

}
```

Any method that has a Shape argument, can be assigned <u>any</u> object that has an “is-a” relationship with Shape

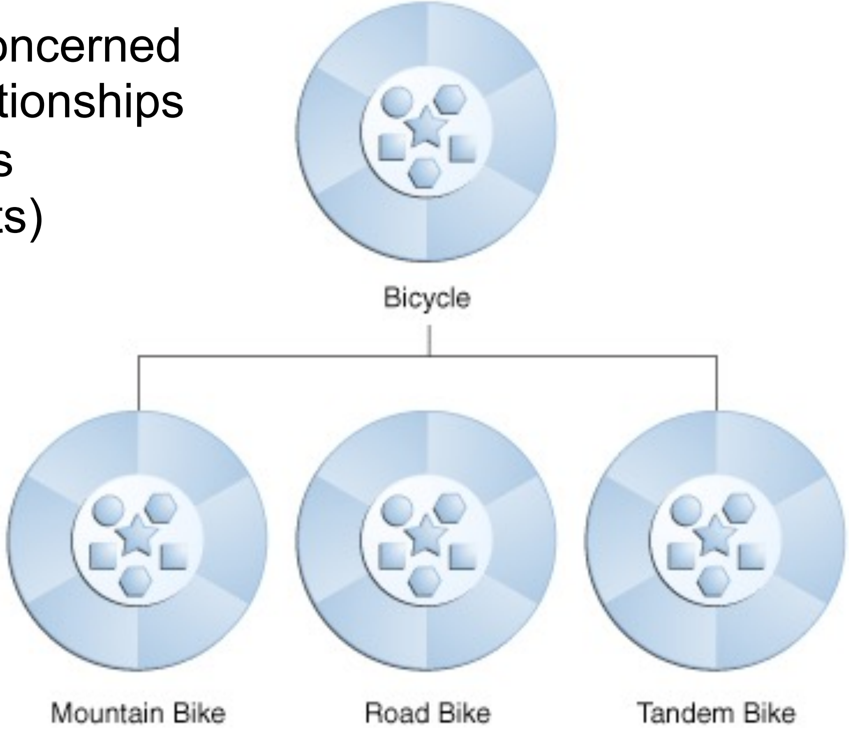
Substitution Principle:

When a parent is expected, a child is accepted

INHERITANCE

Inheritance

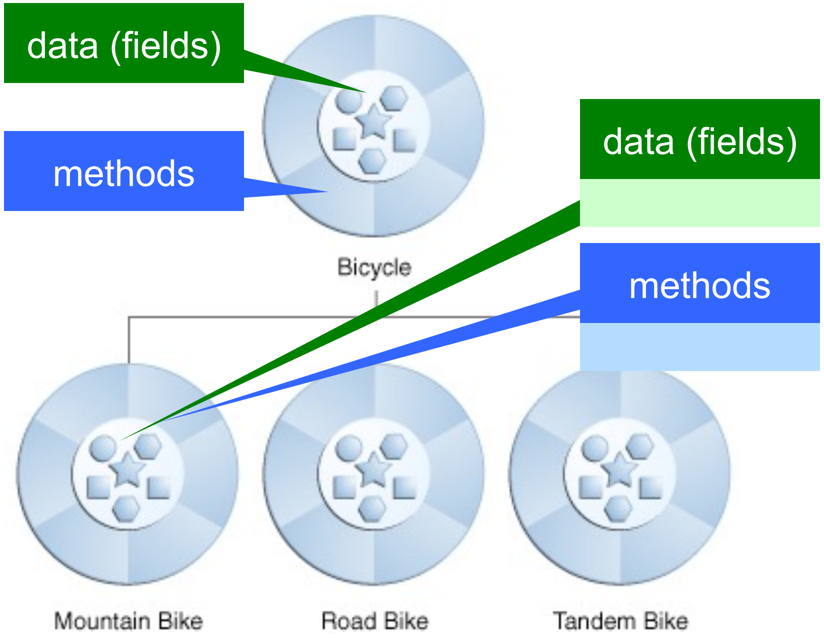
Inheritance is concerned with **“IS-A”** relationships between classes (and their objects)



A child class is specified using “extends” keyword

```
class ChildClass extends ParentClass {

    // ParentClass fields do not need to be specified
    // they are automatically “inherited”

    // ChildClass may ADD additional fields that
    // distinguish Child from Parent



    // ParentClass methods automatically inherited

    // ChildClass may ADD additional methods

}
```

Classes (Child vs. Parent)



data (fields)

methods

data (fields)

methods

parent (superclass)

```
Class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }


}
```

```
Class MountainBike extends Bicycle{

    int shocks = 2;   // new field added

    void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear +
             “shocks: “ + shocks);
    }
}
```

child (subclass)

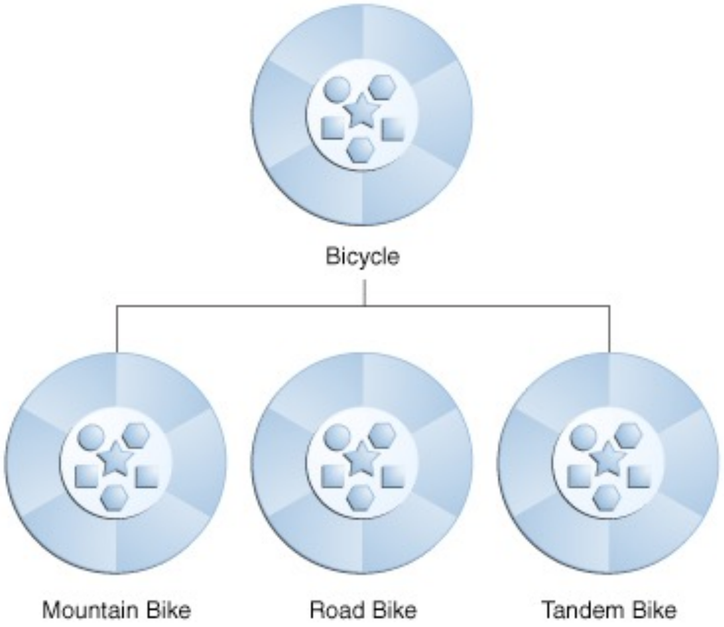
A Parent class

- “shares” its “DNA” with the child class
- i.e. fields/methods in the Parent class, are automatically shared or in common with the Child class

- We can consider the Child class as a “version” of the Parent class (with some additional features).
- Thus, the Child **IS_A** more ***specialized*** version of the Parent

- **Note:**
  - fields or methods may be ”shadowed” or “overridden” in the Child class

Classes (Child vs. Parent)



parent (superclass)

```
Class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }


}
```

```
Class MountainBike extends Bicycle{

    int gear = 11;    // shadows existing gear
    int shocks = 2;

    void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear +
             “shocks: “ + shocks);
    }
}
```

child (subclass)

Shadowing/Overriding

- `public` fields in the child class with the same name as `public` fields in the parent class are said to “shadow” (i.e. substitute)

- `public` methods in the child class with the same signature as `public` methods in the parent class are said to “override” (i.e. substitute)

- gear in MountainBike “**shadows**”(replaces) Bicycle’s version of gear field.

- toString() in the MountainBike class “**overrides**” (replaces) Bicycle’s toString() method.

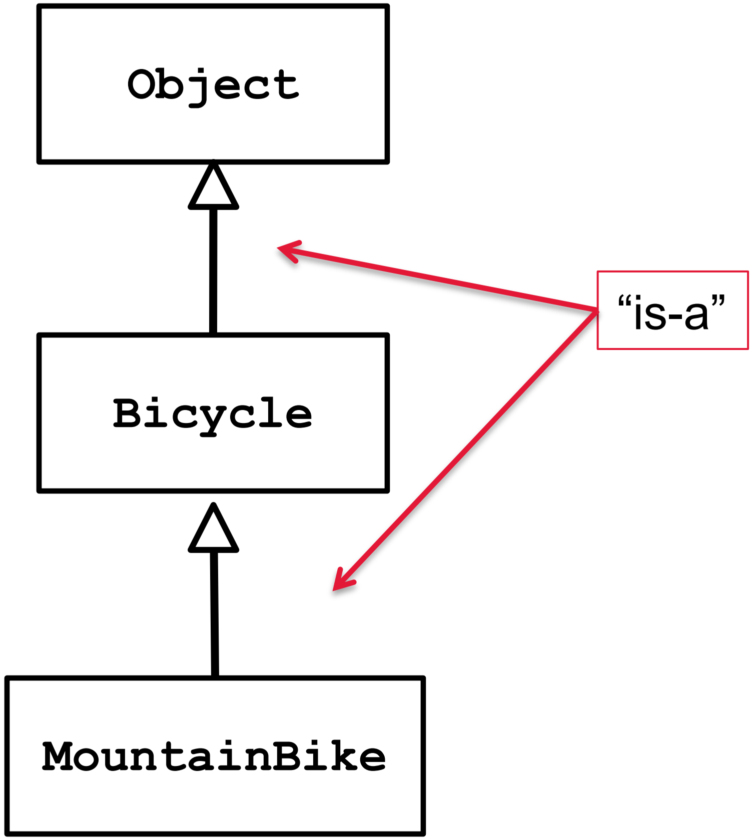
Some Definitions

- we say that a subclass (child) is “derived” from its superclass (parent)

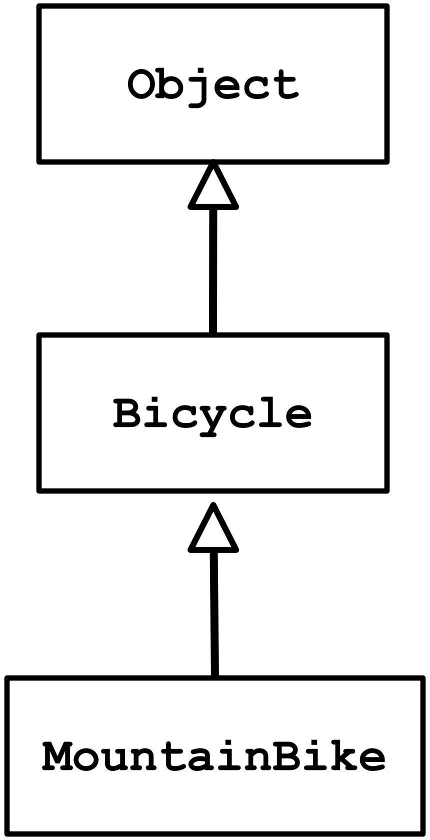
- with the exception of **`Object`**, every class in Java has **one and only one** superclass
  - Java only supports *single inheritance*

- a class **`X`** can be derived from a class that is derived from a class, and so on, all the way back to **`Object`**
    - **`X`** is said to be *descended* from all of the classes in the inheritance chain going back to **`Object`**
    - all of the classes **`X`** is derived from are called *ancestors* of **`X`**

Inheritance (UML)



Inheritance



```
Object obj;
obj = new Object();
obj = new Bicycle();
obj = new MountainBike();

// does not work the other way
// parent object cannot be assigned
// directly to a child reference

Bicycle b =  new Object();      // illegal
MountainBike m = new Bicycle(); // illegal
```

Why Inheritance?

- a subclass inherits all non-private members (fields and methods ***but not constructors***) from its superclass
  - the new class can introduce new fields and methods
  - the new class can re-define (override) its superclass methods

- PURPOSE?
  1. CODE-REUSE

     if there is an existing class that provides some of the functionality you need you can derive a new class from the existing class (code reuse)
  2. POLYMORPHISM

     Because of the substitution principle (is-a), you can use parent references to “hold” different types of objects (from a common family)

Polymorphism

- “of many forms”

- “is-a” == “is substitutable for”
  - provides a mechanism for a uniform reference type to take on (hold/ be assigned) different types of objects
  - objects bear some level of resemblance (in terms of their state and/or their behaviour)

- E.g.
  - Shape objects exhibit similar behaviour
    - e.g. can be drawn

  - Bike objects have some similar properties (shared fields):
    - wheels, seat, handlebars, gears, cadence, brakes, etc.
  - Bike objects may also have similar behaviour (shared methods):
    - gearUp(), gearDown(), etc.

Polymorphic Behaviour

- “of many forms”

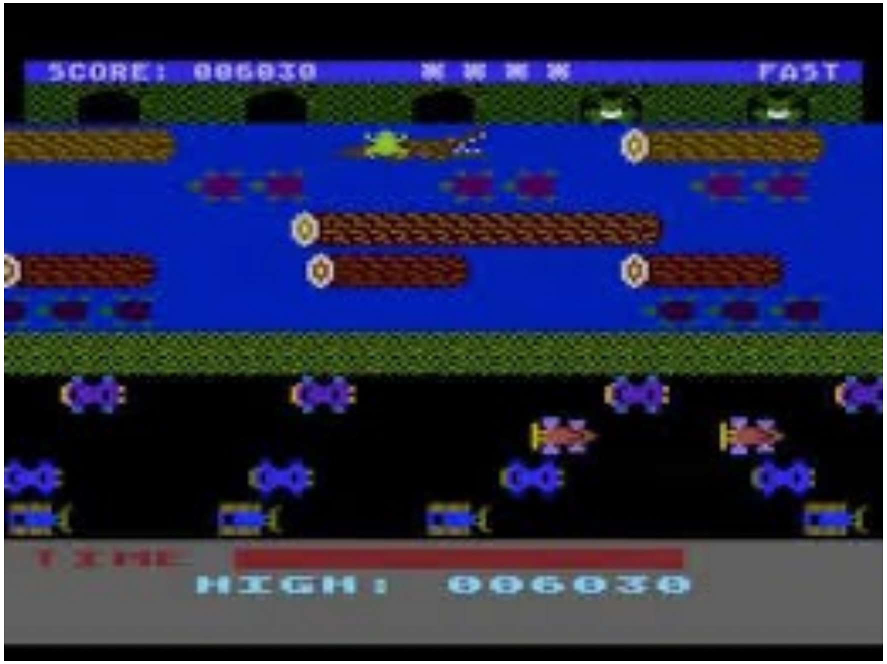
- If the same method is passed different “is-a” objects, the method can appear to *behave* differently

  == polymorphic “behaviour”

```
RasterImage img = new RasterImage();
Graphics2D gfx = img.getGraphics2D();

gfx.draw( new Rectangle2D.Double() );   // draws Rectangle
gfx.draw( new Ellipse2D.Double() );     // draws Ellipse
```

Example (Frogland elements)

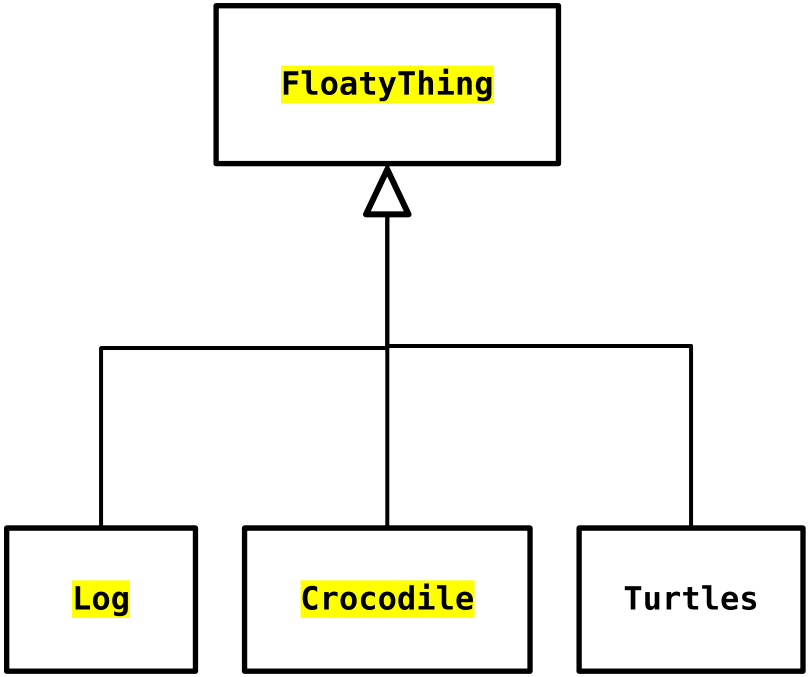
- Think about categories of “things” that look/act in a particular way

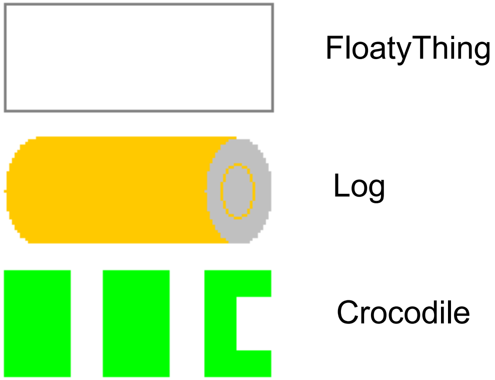


e.g. floating things?

- logs
- crocodiles
- turtles

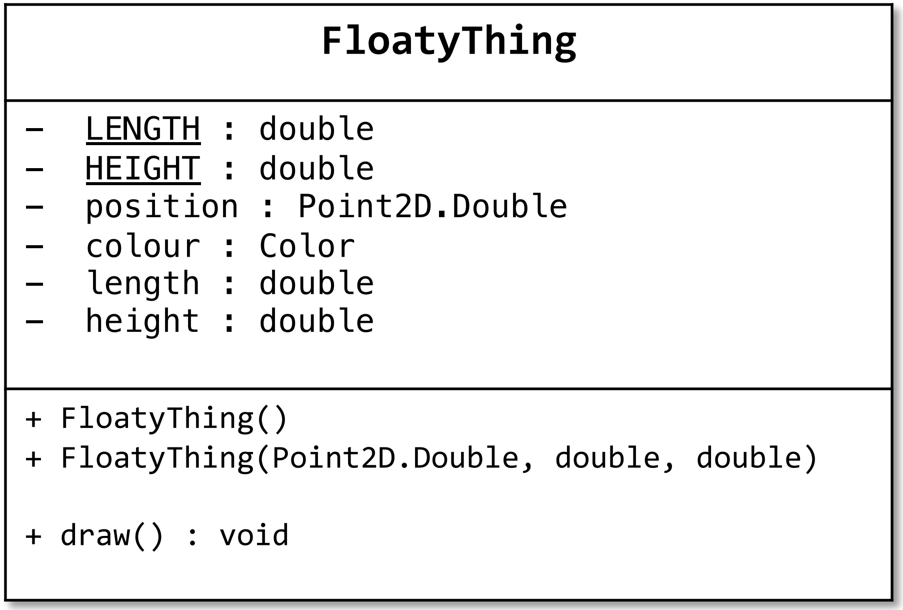
Let’s define a category of things that float





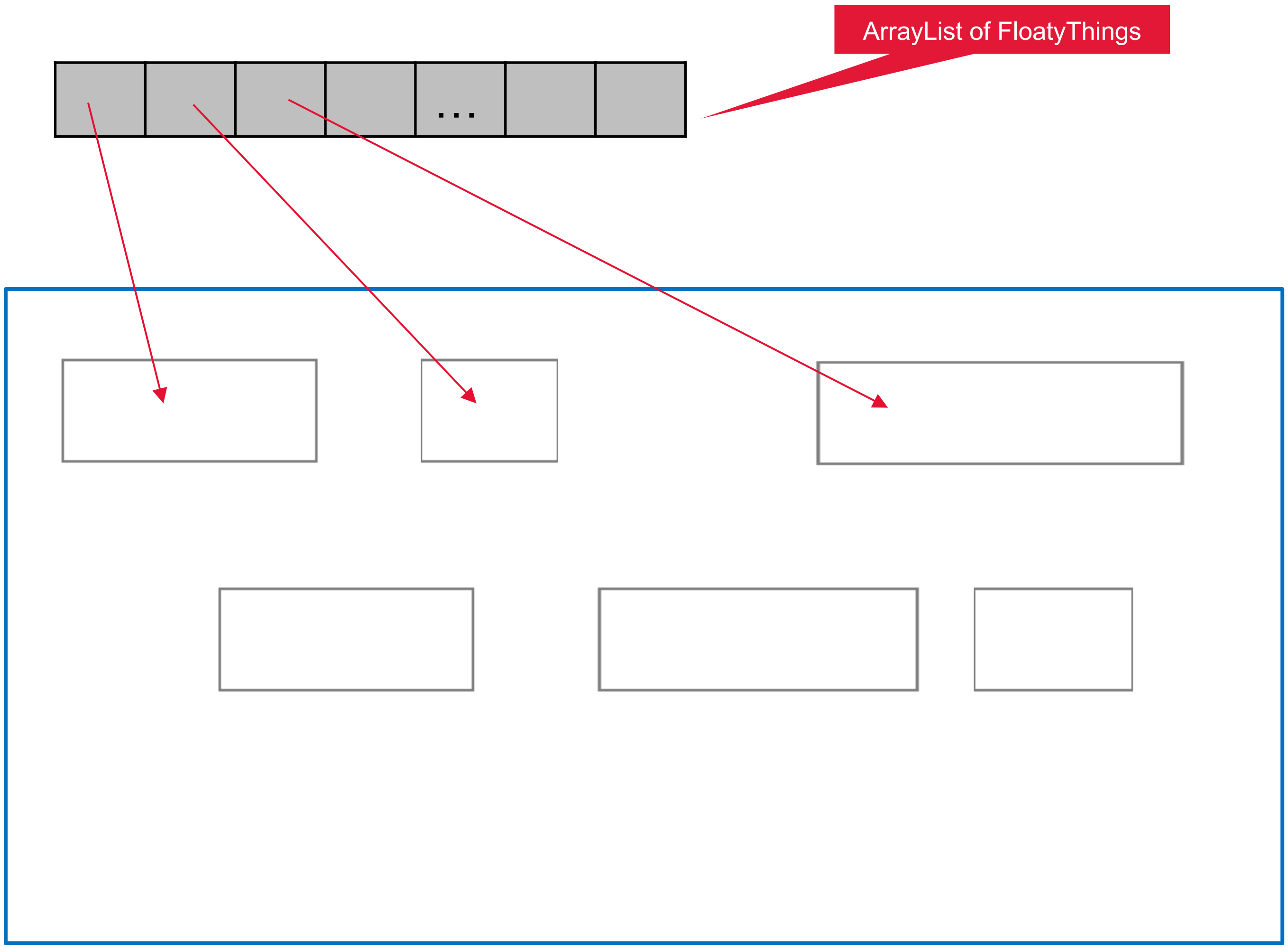
Graphical representations

superclass (parent)



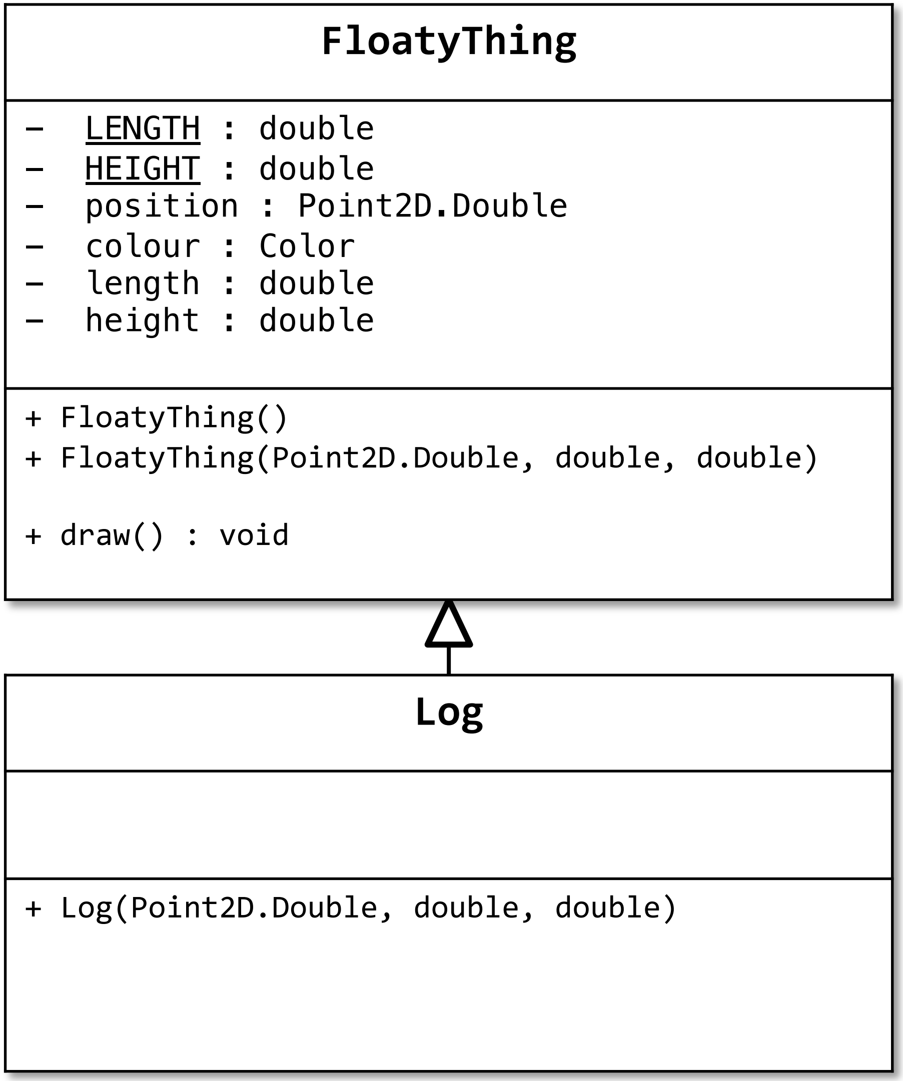
```
FloatyThing

–  LENGTH : double
–  HEIGHT : double
–  position : Point2D.Double
–  colour : Color
–  length : double
–  height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw() : void
```

```java
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Point2D;

public class FloatyThing {

     private static final double LENGTH = 100.0;
     private static final double HEIGHT = 10.0;

     protected Point2D.Double position;
     protected Color colour;
     protected double length;
     protected double height;

     public FloatyThing() {
          this.colour = Color.gray;
          this.position = new Point2D.Double(0,0);
          this.length = FloatyThing.LENGTH;
          this.height = FloatyThing.HEIGHT;
     }
     public FloatyThing(Point2D.Double position, double length, double height) {
          this.colour = Color.GRAY;
          this.position = new Point2D.Double(position.x,position.y);
          this.length = length;
          this.height = height;
     }
     public void draw(Graphics2D gfx) {
          Color origCol = gfx.getColor();
          gfx.setColor(this.colour);
          gfx.drawRect((int)this.position.getX(), (int)this.position.getY(),
                                             (int)this.length, (int)this.height);
          gfx.setColor(this.colour);
     }
}
```



```
ArrayList<FloatyThings> topRow = new ArrayList<FloatyThings>();
topRow.add(new FloatyThing());
topRow.add(new FloatyThing(new Point2D.Double(0,40),10,30));

// …
```

subclass (child)



```
FloatyThing

−  LENGTH : double
−  HEIGHT : double
−  position : Point2D.Double
−  colour : Color
−  length : double
−  height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw() : void
```

```
Log

+ Log(Point2D.Double, double, double)
```

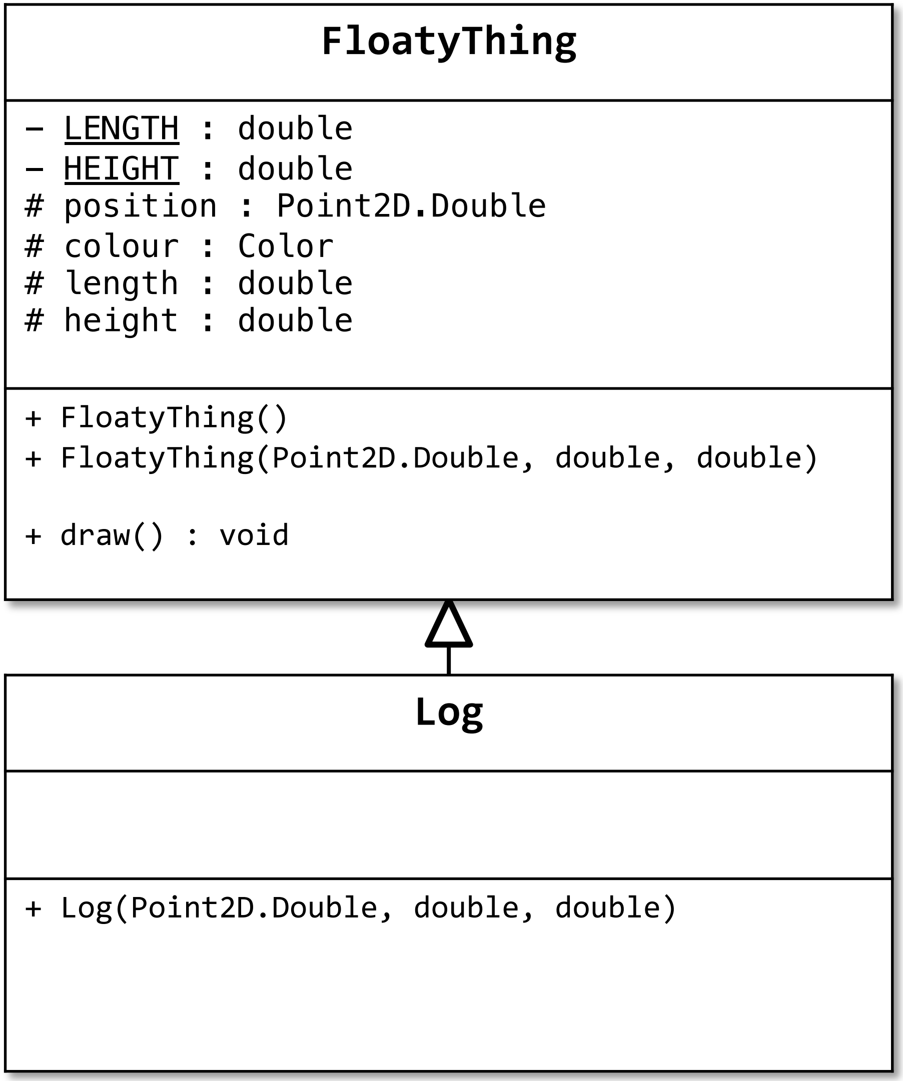
What parts of an API are inherited by the subclass? (Log extends FloatyThing)

Only public / protected features of super class

What is a Subclass?

- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
    - All **public** and **protected** components are inherited
    - **private** components are only accessible if the parent provides access through its API
  - the superclass subobject needs to be constructed (just like a regular object)
    - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

subclass (child)



FloatyThing

− LENGTH : double
− HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

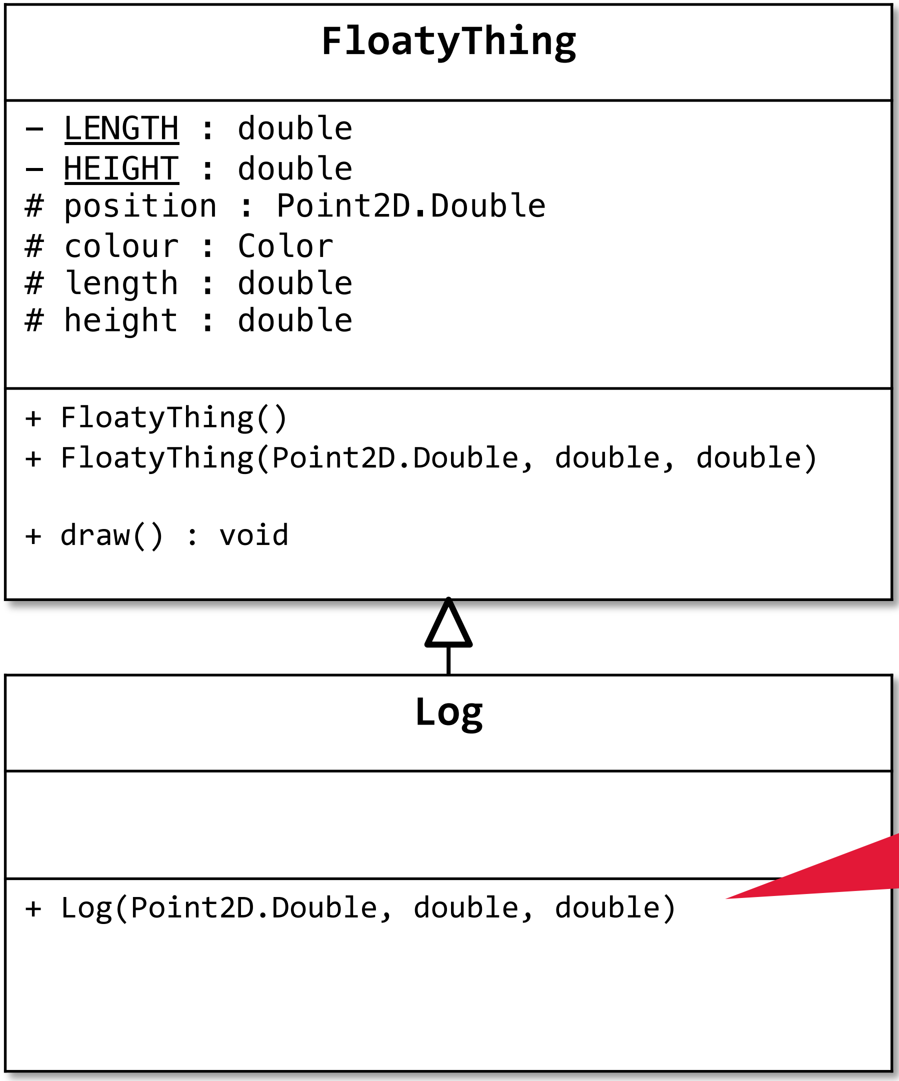
+ draw() : void

Log

+ Log(Point2D.Double, double, double)

# == protected access
(appears public to any subclass)
(appears private to any client)

subclass (child)



```
FloatyThing

- LENGTH : double
- HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw() : void
```

```
Log

+ Log(Point2D.Double, double, double)
```

subclass constructor *MUST* invoke one of its parent's constructors

subclass objects are always comprised of a superclass subobject (which needs to be created)

ASIDE:

Recall: No method may call a constructor …

However:

- A constructor can call another constructor within the same class **<u>if and only if</u>**:
- It does the call before it does anything else (i.e. as its first statement)

- To call a constructor within same class → use “this()”
- To call a constructor from the parent class → use “super()”

```java
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Point2D;

public class FloatyThing {

     private static final double LENGTH = 100.0;
     private static final double HEIGHT = 10.0;

     protected Point2D.Double position;
     protected Color colour;
     protected double length;
     protected double height;

     public FloatyThing() {

          this(Color.gray, new Point2D.Double(0,0),FloatyThing.LENGTH,
                                                            FloatyThing.HEIGHT);

     }
     public FloatyThing(Point2D.Double position, double length, double height) {
          this.colour = Color.GRAY;
          this.position = new Point2D.Double(position.x,position.y);
          this.length = length;
          this.height = height;
     }
     public void draw(Graphics2D gfx) {
          Color origCol = gfx.getColor();
          gfx.setColor(this.colour);
          gfx.drawRect((int)this.position.getX(), (int)this.position.getY(),
                                             (int)this.length, (int)this.height);
          gfx.setColor(this.colour);
     }
}
```

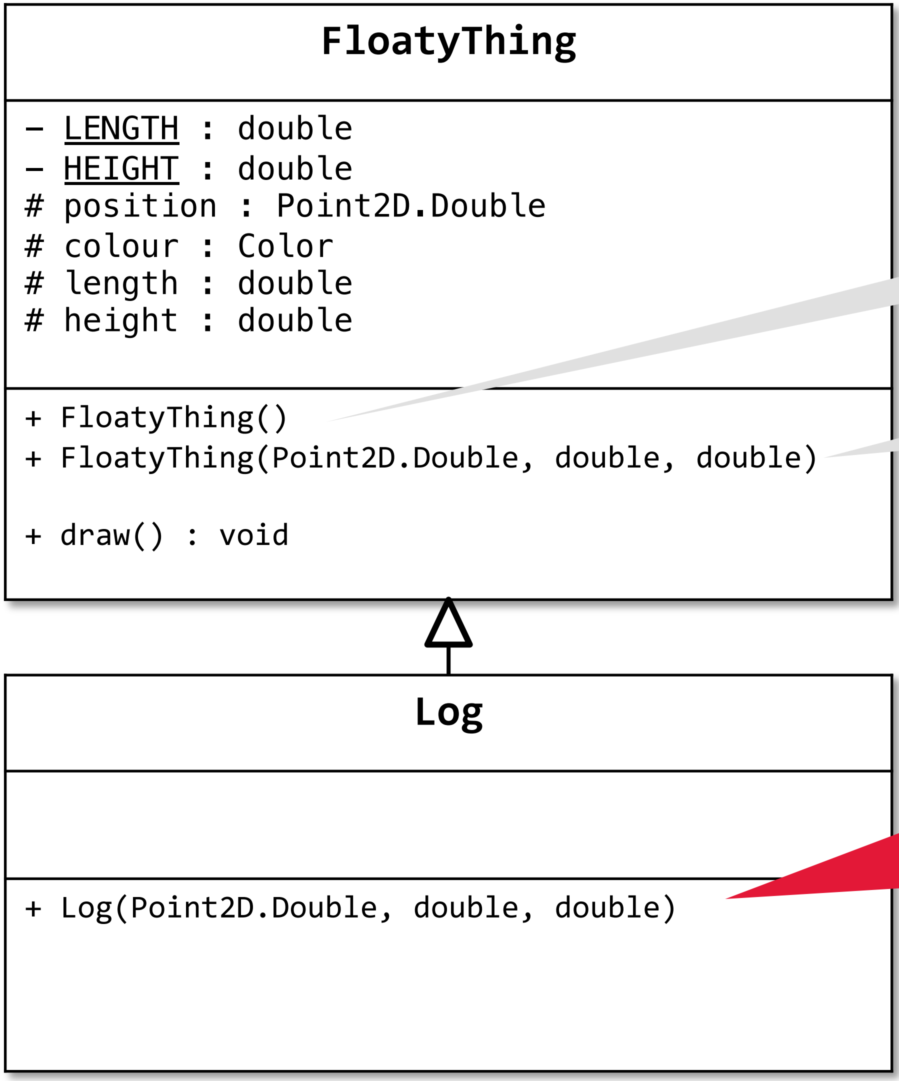
using this() as a method call in a constructor usually used to run the more specialized constructor (passing defaults)

Constructors of Subclasses

- the purpose of a constructor is to set the values of the fields of **`this`** object

- how can a constructor set the value of a field that belongs to the superclass and is **`private`**?
  - by calling the superclass constructor and using **`super()`** keyword (as a method call)

subclass (child)



FloatyThing

- LENGTH : double
- HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw() : void

Log

+ Log(Point2D.Double, double, double)

```
// invoked from Log(…) as:
super();
```

```
// invoked from Log(…) as:
super(p,x,y);
```

subclass constructor *MUST*
invoke one of its parent’s constructors

subclass objects are always comprised
of a superclass subobject
(which needs to be created)

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Point2D;


public class Log extends FloatyThing {

     // Log does not define any additional fields
     // (however Log inherits public/protected fields from FloatyThing)


     public Log(Point2D.Double position, double length, double height) {

          super(position, length, height);        // calls FloatyThing ctor

          this.colour = Color.ORANGE;
     }

}
```

If no call to super(..)

super()
is implicitly called by JVM
to try and invoke default ctor of parent

```java
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Point2D;


public class Log extends FloatyThing {

    // Log does not define any additional fields
    // (however Log inherits public/protected fields from FloatyThing)


    public Log(Point2D.Double position, double length, double height) {

        // implicit call to FloatyThing default ctor (i.e. super(); )

        this.colour = Color.ORANGE;

    }

}
```

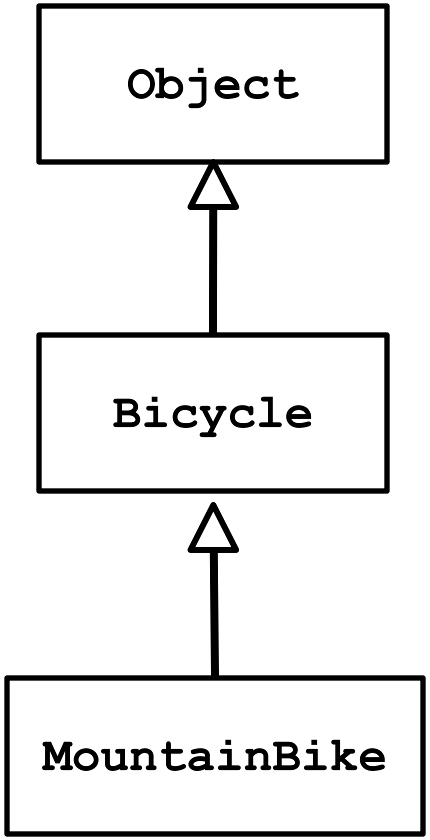
Implicit call to super() creates the FloatyThing part of a Log object, using default settings for the fields inherited from FloatyThing class

The colour field is then modified by Log’s ctor

Constructors of Subclasses

1. the first line in the body of every constructor ***must*** be a call to another constructor
   - if it is not then Java will insert a call to the superclass default constructor
     - if the superclass default constructor does not exist or is private then a compilation error occurs
2. a call to another constructor can only occur on the first line in the body of a constructor
3. the superclass constructor must be called during construction of the derived class

Inheritance



Bicycle ctor invokes Object ctor

MountainBike ctor invokes Bicycle ctor

But how?

```
public class Bicycle {

    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;

    public void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }

}
```

```
public class MountainBike extends Bicycle{

    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;
    public int shocks = 2;

    public void toString() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear +
             “shocks: “ + shocks);
    }
}
```

A class that does not extend, implicitly extends Object

A class without a ctor (is implicitly given a default ctor) i.e.. Bicycle() { }

Inside Bicycle() { }

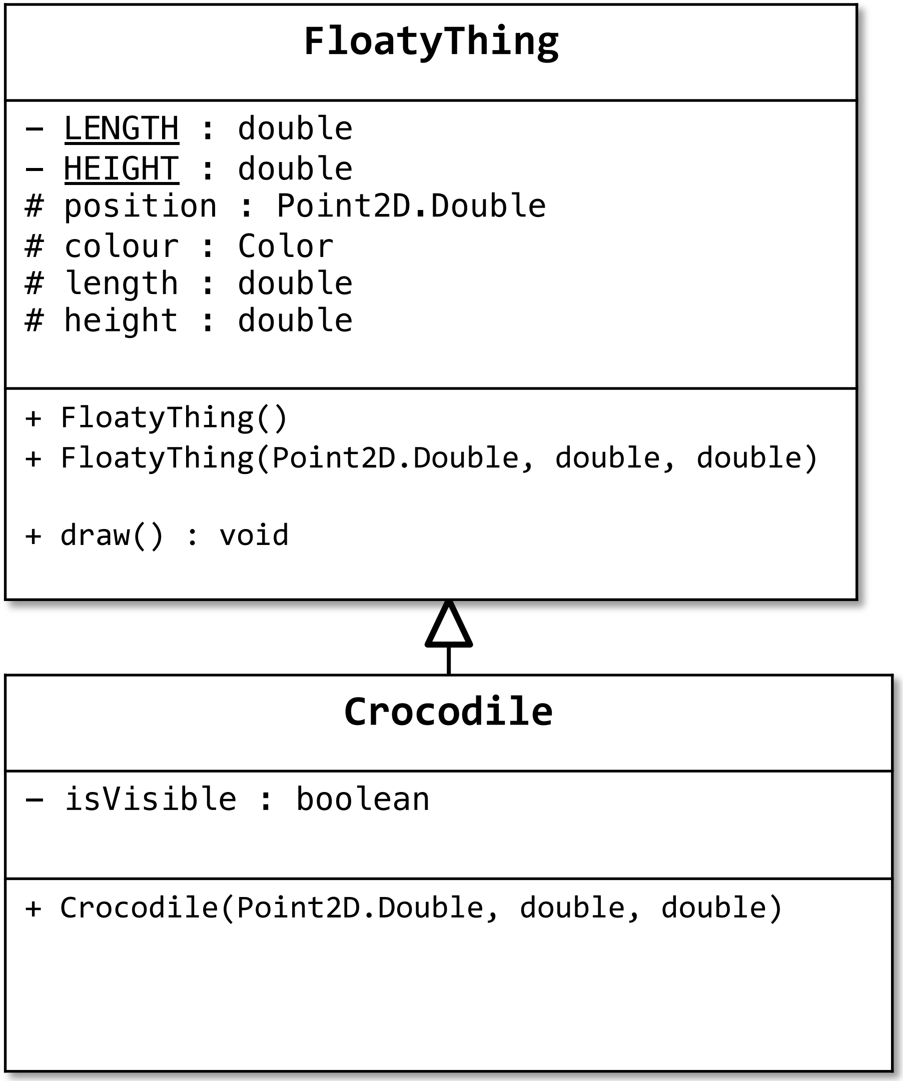
Exists an implicit call to super();

Implication:
All classes are instantiable (unless they are explicitly given a ctor that is private)

e.g. Math class would have something like this:
`private Math() { }`

** any static/utility class would have the same to prevent instantiation

subclass (child)



```
FloatyThing

- LENGTH : double
- HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw() : void
```

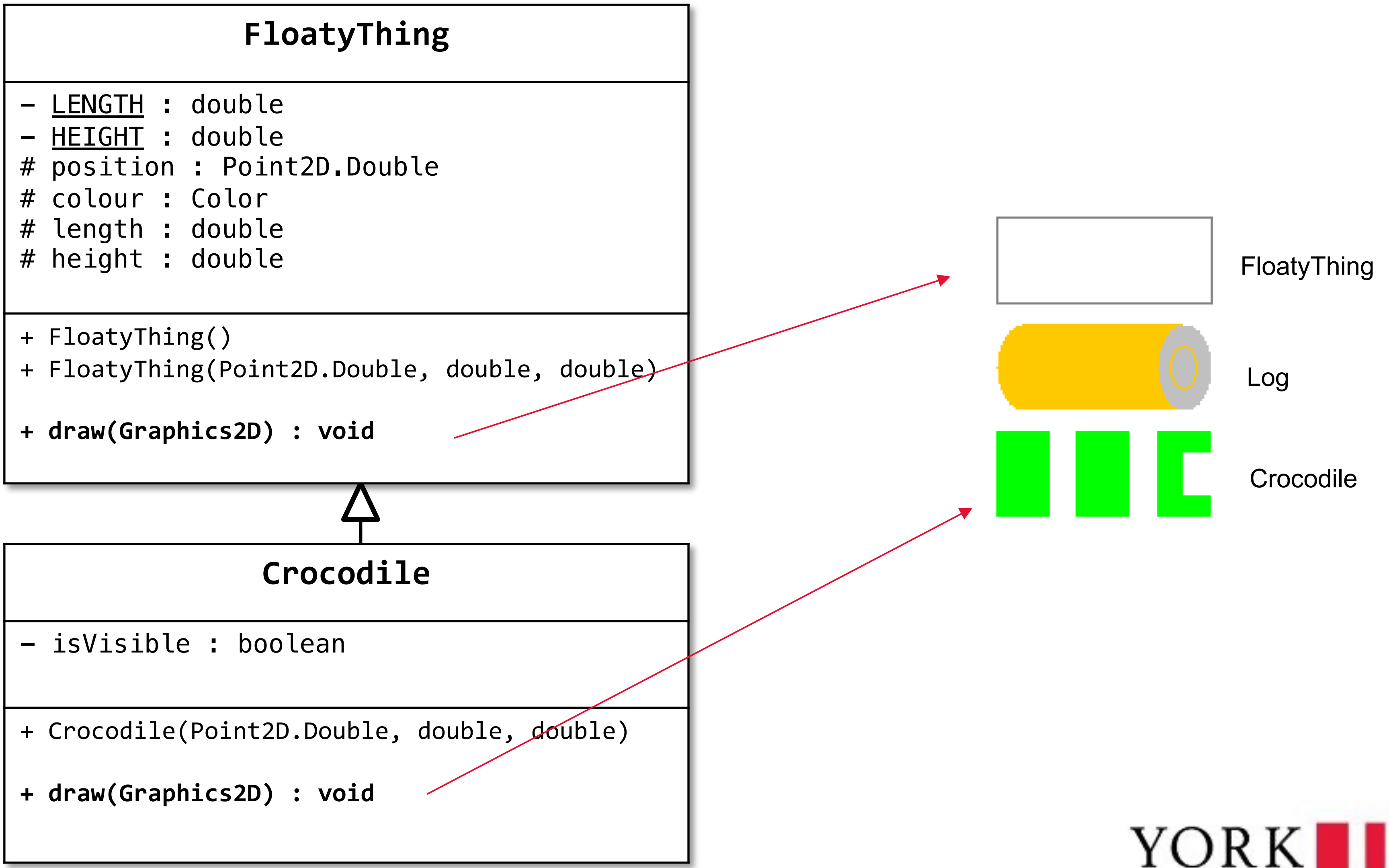
```
Crocodile

- isVisible : boolean

+ Crocodile(Point2D.Double, double, double)
```

```
public class Crocodile extends FloatyThing {

     private boolean isVisible;


     public Crocodile(Point2D.Double position, double length, double height) {

          super(position, length, height);

          this.colour = Color.GREEN;
          this.isVisible = true;

     }

}
```

subclasses:

- contain any fields/methods/ctors it introduces plus any inherited features from its superclass
- It can override methods it inherits!

Overriding methods



FloatyThing

− LENGTH : double
− HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

+ draw(Graphics2D) : void

Crocodile

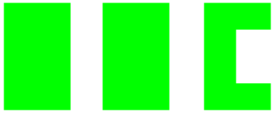
− isVisible : boolean

+ Crocodile(Point2D.Double, double, double)

+ draw(Graphics2D) : void

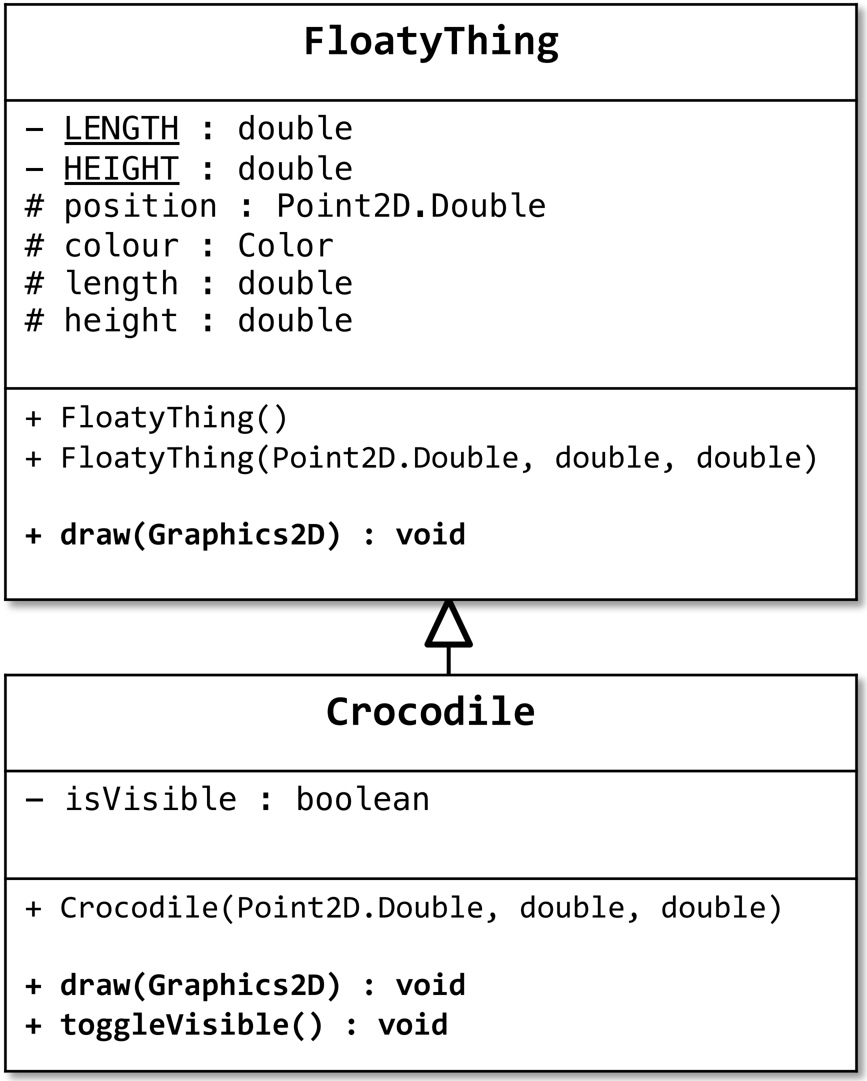
```
// in FloatyThing:

public void draw(Graphics2D gfx) {

     Color origCol = gfx.getColor();
     gfx.setColor(this.colour);

     gfx.drawRect(  (int)this.position.getX(), (int)this.position.getY(),
                    (int)this.length,(int)this.height                     );

     gfx.setColor(this.colour);


}
```

```
// in Crocodile

@Override
public void draw(Graphics2D gfx) {

     Color origCol = gfx.getColor();
     gfx.setColor(this.colour);

     gfx.fillRect(  (int)this.position.getX(), (int)this.position.getY(),
                    (int)(this.length/4), (int)this.height);

     gfx.fillRect(  (int)(this.position.getX()+3*this.length/8), (int)this.position.getY(),
                    (int)(this.length/4), (int)this.height);

     gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                    (int)(this.length/4), (int)(this.height/4));

     gfx.fillRect(  (int)(this.position.getX()+6*this.length/8),
                    (int)(this.position.getY()+3*this.height/4), (int)(this.length/4),
                    (int)(this.height/4));

     gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                    (int)(this.length/8), (int)this.height);


     gfx.setColor(this.colour);


}
```



```
// in Log

@Override
public void draw(Graphics2D gfx) {

     Color origCol = gfx.getColor();
     gfx.setColor(this.colour);


     gfx.fillOval(  (int)this.position.getX(), (int)this.position.getY(),
                    (int)(this.length/4), (int)this.height);

     gfx.fillRect(  (int)(this.position.getX()+1*this.length/8),
                    (int)this.position.getY(), (int)(3*this.length/4), (int)this.height);


     gfx.setColor(Color.LIGHT_GRAY);

     gfx.fillOval(  (int)(this.position.getX()+3*this.length/4),
                    (int)this.position.getY(), (int)(this.length/4), (int)this.height);

     gfx.setColor(this.colour);

     gfx.drawOval(  (int)(this.position.getX()+13*this.length/16),
                    (int)(this.position.getY()+this.height/4),
                    (int)(this.length/8), (int)(this.height/2));

     gfx.setColor(this.colour);

}
```

Overriding + new methods



FloatyThing

− LENGTH : double
− HEIGHT : double
# position : Point2D.Double
# colour : Color
# length : double
# height : double

+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)

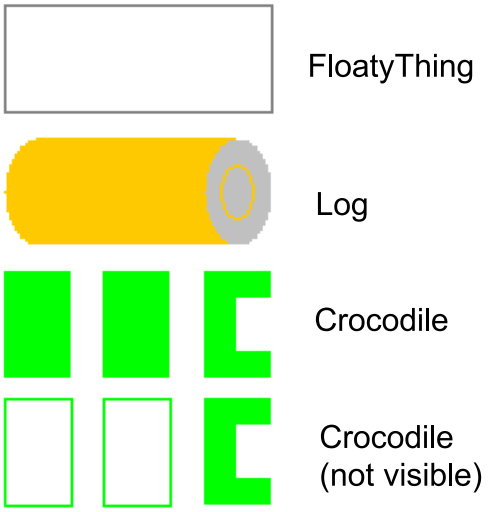
+ draw(Graphics2D) : void

Crocodile

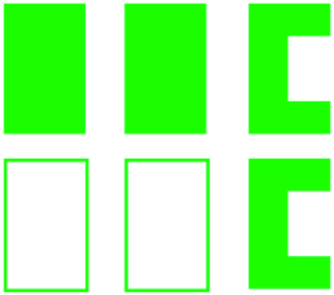
− isVisible : boolean

+ Crocodile(Point2D.Double, double, double)

+ draw(Graphics2D) : void
+ toggleVisible() : void

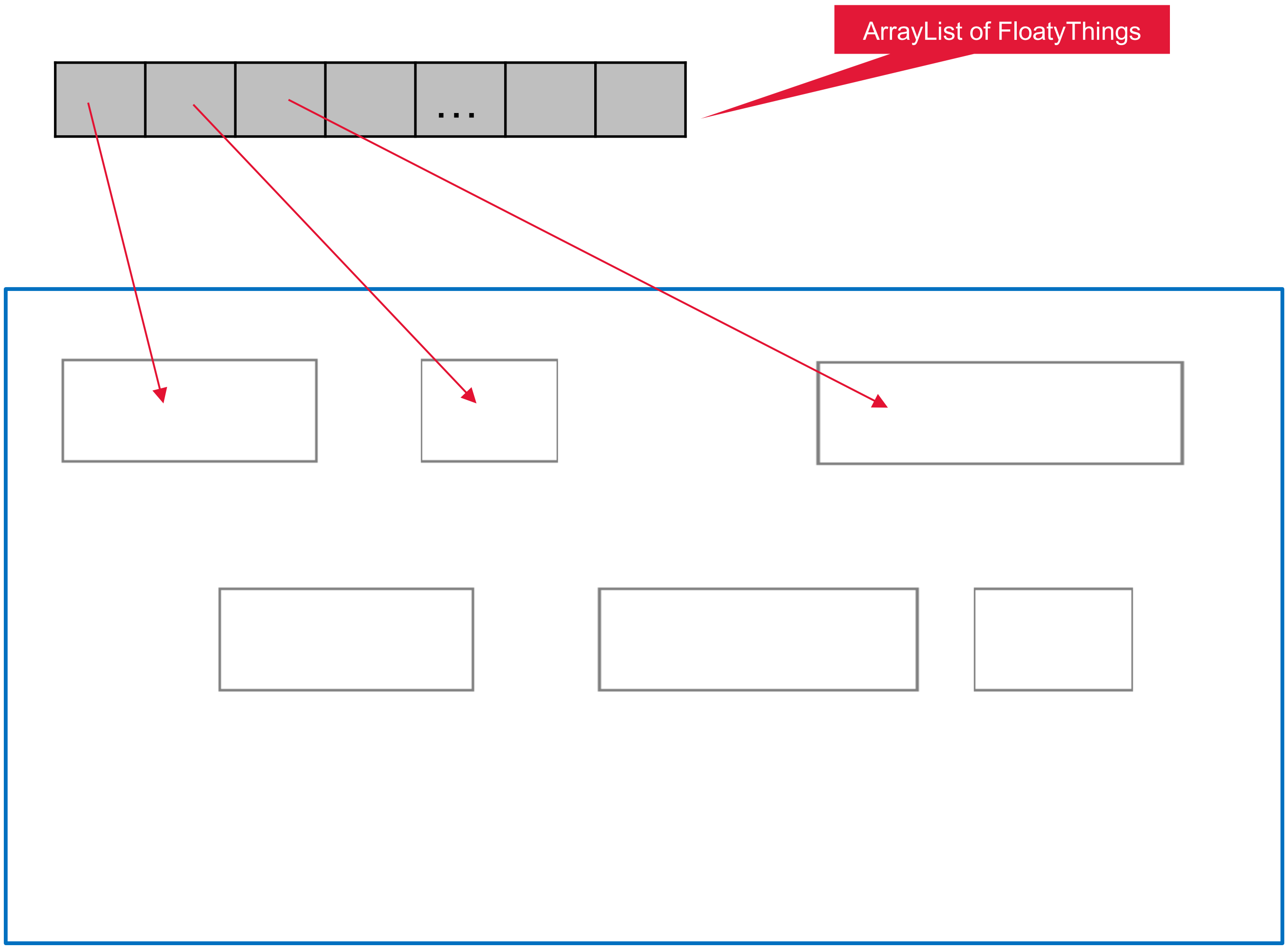


```
\\ in Crocodile (note MAGIC numbers used due to lack of space on slide)

public void toggleVisible() {
    this.isVisible = !this.isVisible;
}


@Override
public void draw(Graphics2D gfx) {

    Color origCol = gfx.getColor();
    gfx.setColor(this.colour);

    if (this.isVisible) {
        gfx.fillRect(  (int)this.position.getX(), (int)this.position.getY(), (int)(this.length/4),
                       (int)this.height);
        gfx.fillRect(  (int)(this.position.getX()+3*this.length/8), (int)this.position.getY(),
                       (int)(this.length/4), (int)this.height);
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                       (int)(this.length/4), (int)(this.height/4));
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8),
                       (int)(this.position.getY()+3*this.height/4), (int)(this.length/4),
                       (int)(this.height/4));
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                       (int)(this.length/8), (int)this.height);
    }
    else {
        gfx.drawRect(  (int)this.position.getX(), (int)this.position.getY(),
                       (int)(this.length/4), (int)this.height);
        gfx.drawRect(  (int)(this.position.getX()+3*this.length/8), (int)this.position.getY(),
                       (int)(this.length/4), (int)this.height);
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                       (int)(this.length/4), (int)(this.height/4));
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8),
                       (int)(this.position.getY()+3*this.height/4), (int)(this.length/4),
                       (int)(this.height/4));
        gfx.fillRect(  (int)(this.position.getX()+6*this.length/8), (int)this.position.getY(),
                       (int)(this.length/8), (int)this.height);
    }
    gfx.setColor(this.colour);

}
```



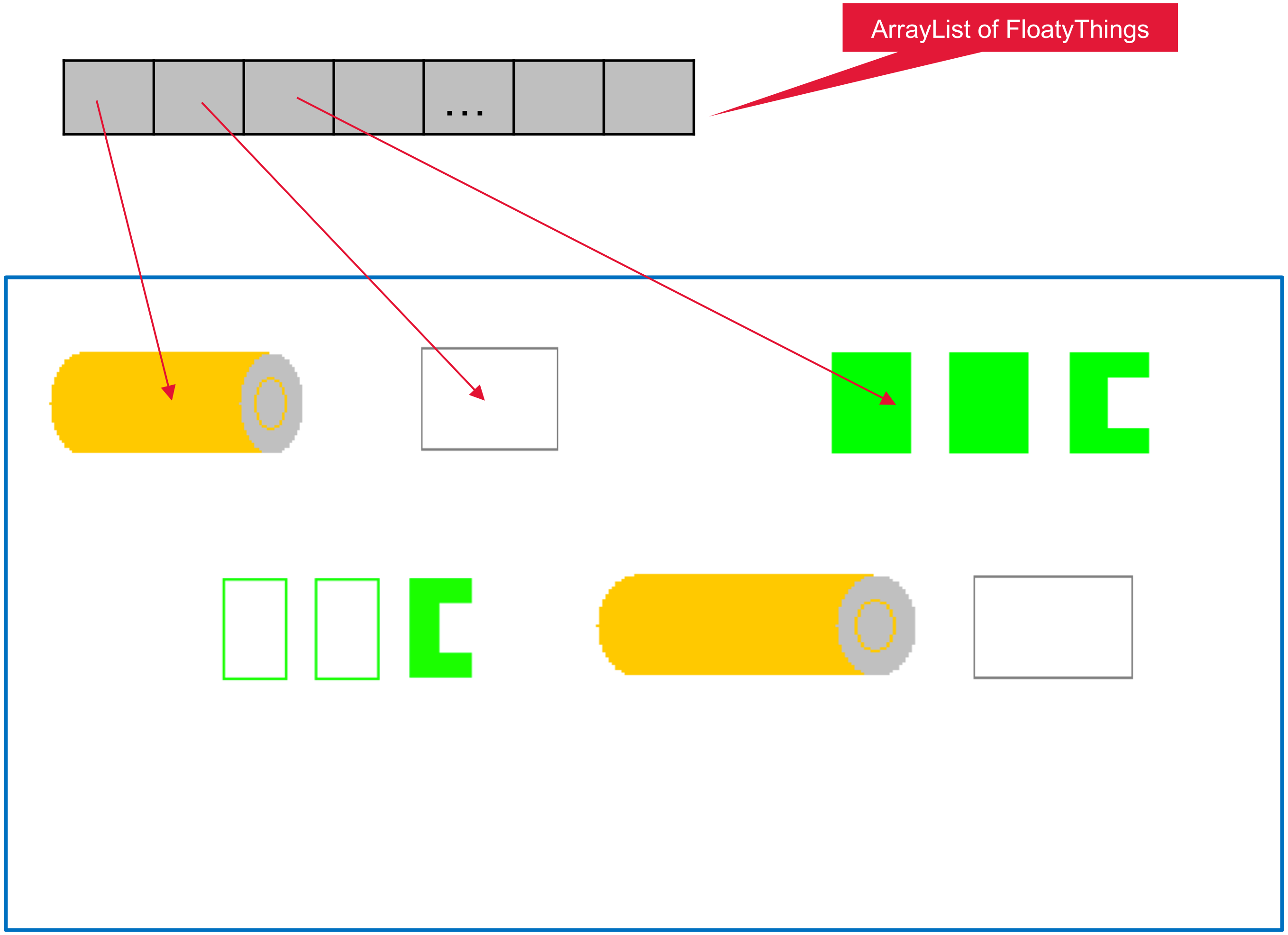
Crocodile

Crocodile (not visible)



ArrayList of FloatyThings

```
ArrayList<FloatyThings> topRow = new ArrayList<FloatyThings>();
topRow.add(new FloatyThing());
topRow.add(new FloatyThing(new Point2D.Double(0,40),10,30));

// …
```



```
// imagine we have added multiple different sub-type objects to topRows
// assume gfx is the Graphics2D reference from RasterImage

for (FloatyThing thing : topRow ) {
          thing.draw(gfx);
}
```

Take aways:

- 2 types of “is-a” relationship
  - Interfaces & Inheritance
  - Each define “types”, these types can hold objects of any sub-type defined in their hierarchy

- Both are forms of “is-a” relationship & support object “substitution”
  - Handy for storing different (related) objects in a common container (e.g. array or ArrayList)
  - Handy for enforcing commonalities amongst state & behaviour of a “family” or “hierarchy” of classes (and specifically their objects)

- Inheritance (class extension):
  - organizes abstractions of *objects*
  - abstracts aspects of their state (data)
  - abstract aspects of their behavior (methods)
  - more specialized (sub-types) can override methods & add fields