EECS 1720
Building Interactive Systems

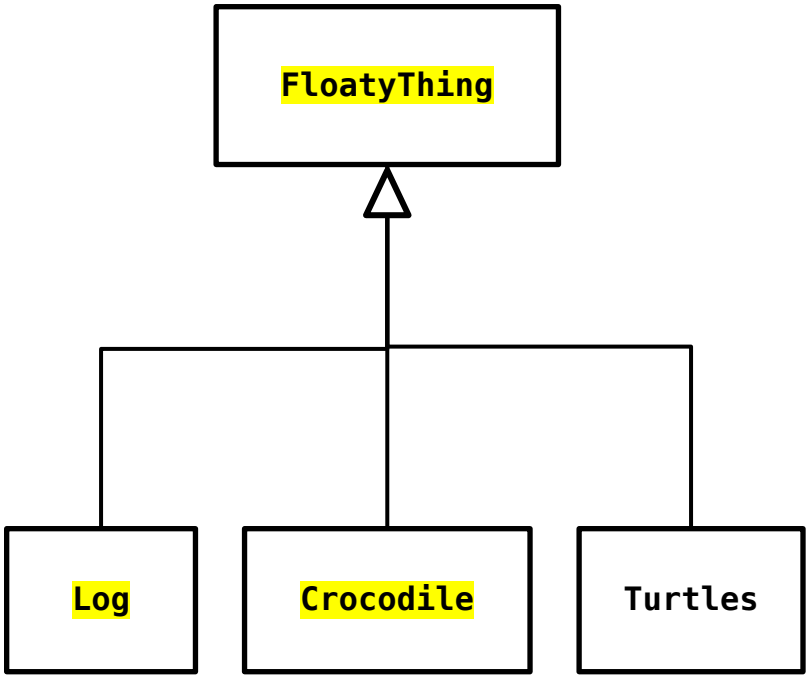
Lecture 10 :: Object Hierarchies 2

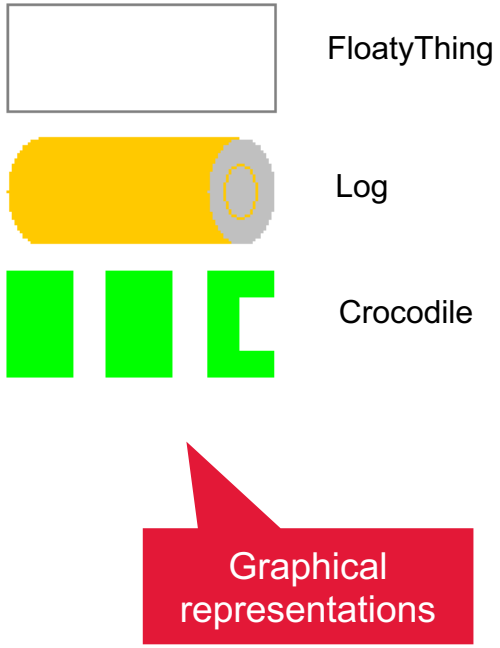
- Inheritance (continued)

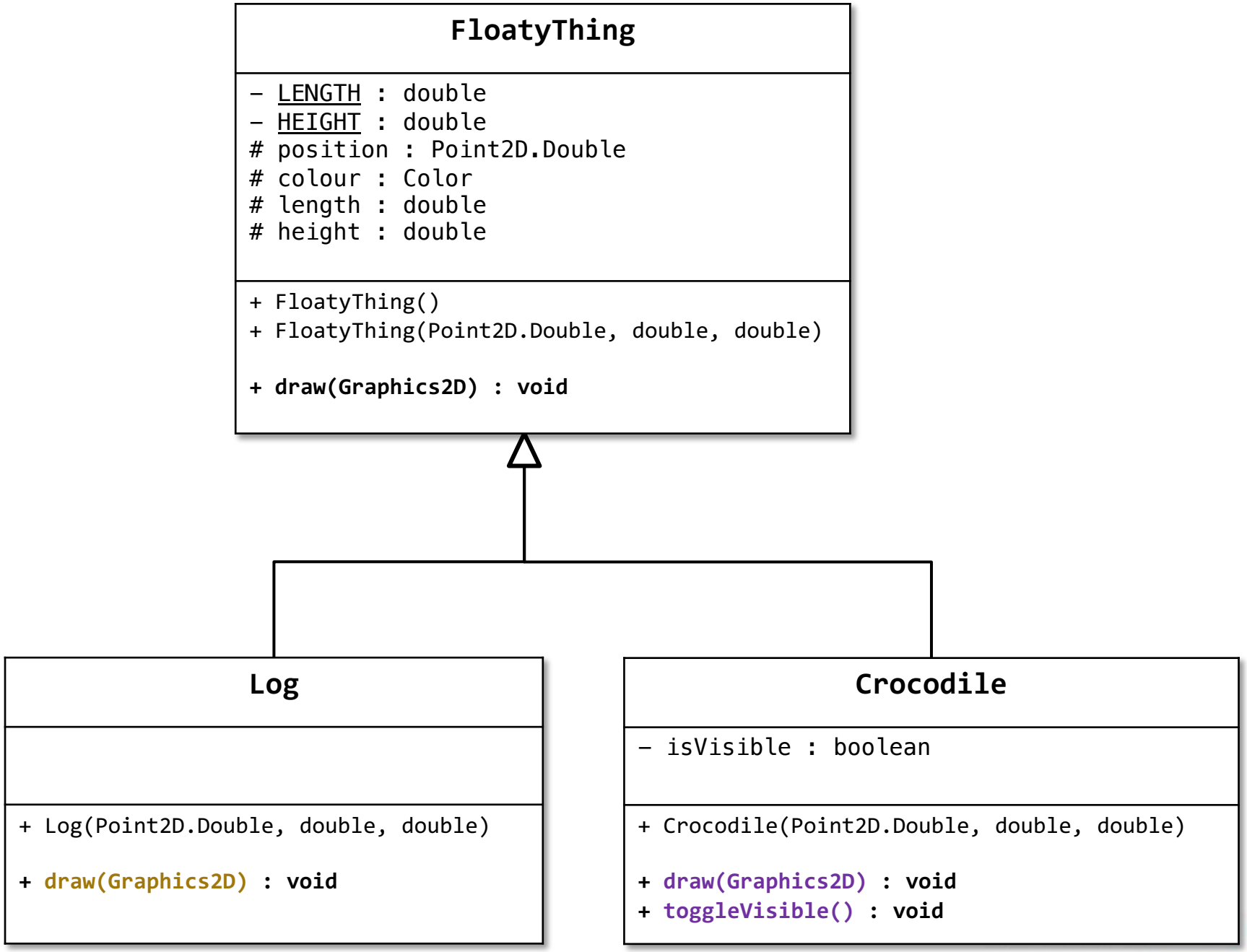
YORK UNIVERSITÉ UNIVERSITY

Recall: INHERITANCE

Inheritance (is-a → “is substitutable for”)






FloatyThing
– LENGTH : double
– HEIGHT : double
position : Point2D.Double
colour : Color
length : double
height : double
+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)
+ draw(Graphics2D) : void
Log
+ Log(Point2D.Double, double, double)
+ draw(Graphics2D) : void
Crocodile
– isVisible : boolean
+ Crocodile(Point2D.Double, double, double)
+ draw(Graphics2D) : void
+ toggleVisible() : void

Why use class hierarchies?

- Code re-use + 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)

- example:
  - container holds type FloatyThing, then it can also hold any sub-type of FloatyThing

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
```

How is Polymorphic Behaviour possible?

Recall:

- constructors of sub-classes must call constructors of immediate super-classes

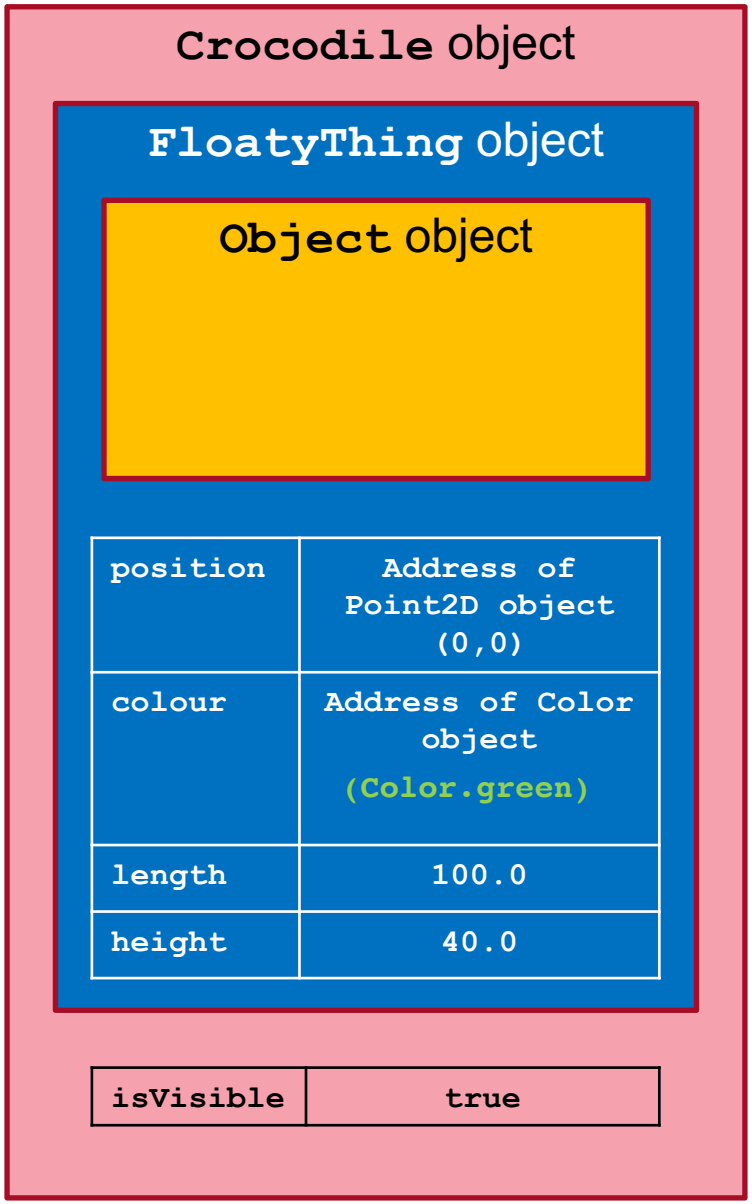
- why is the constructor call to the superclass needed?
  - because **`Crocodile`** is-a **`FloatyThing`** and the **`FloatyThing`** part of **`Crocodile`** needs to be constructed

- depending on which actual reference is ”holding” the object, this gives a ”lens” on related sub-properties of the object

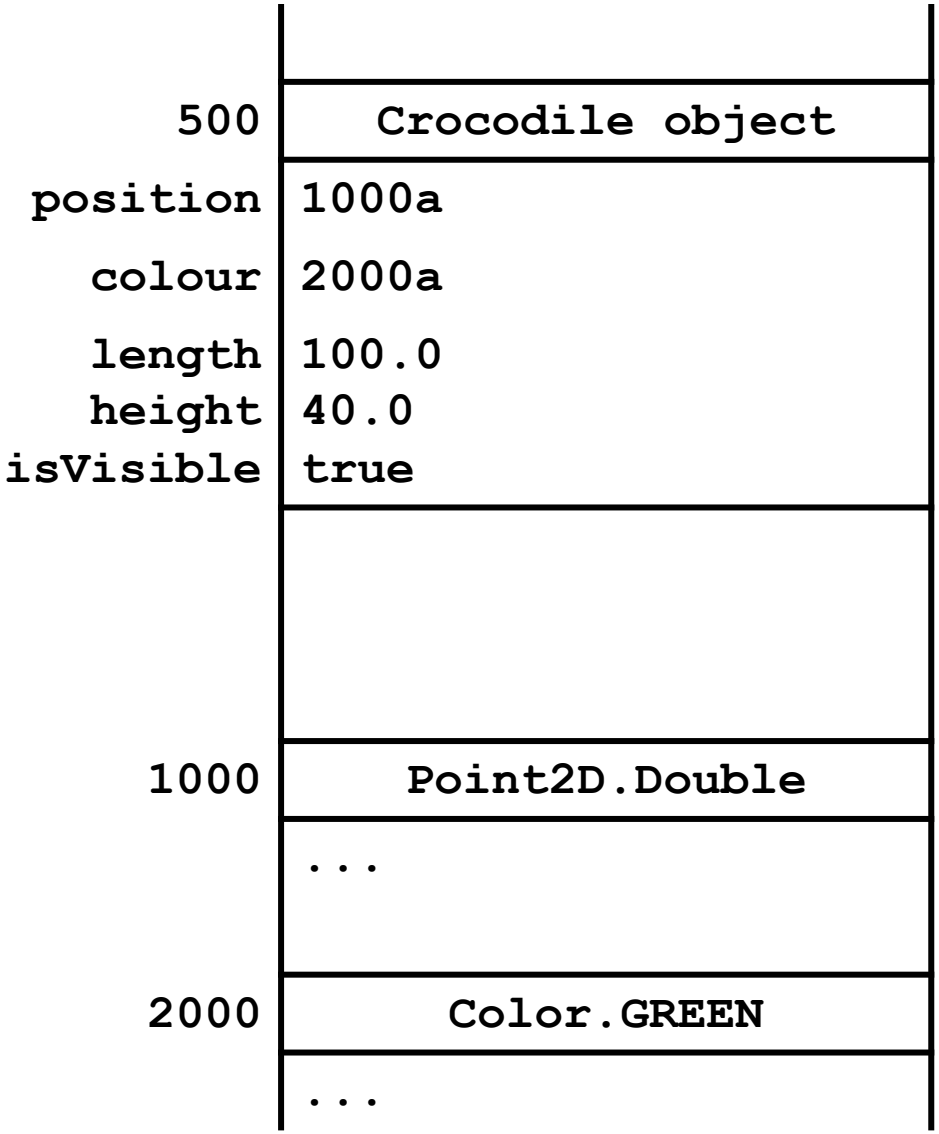
What's happening in memory during instantiation of a sub-class object??

```
Crocodile croc =
   new Crocodile(new Point2D.Double(), 100.0, 40.0);
```

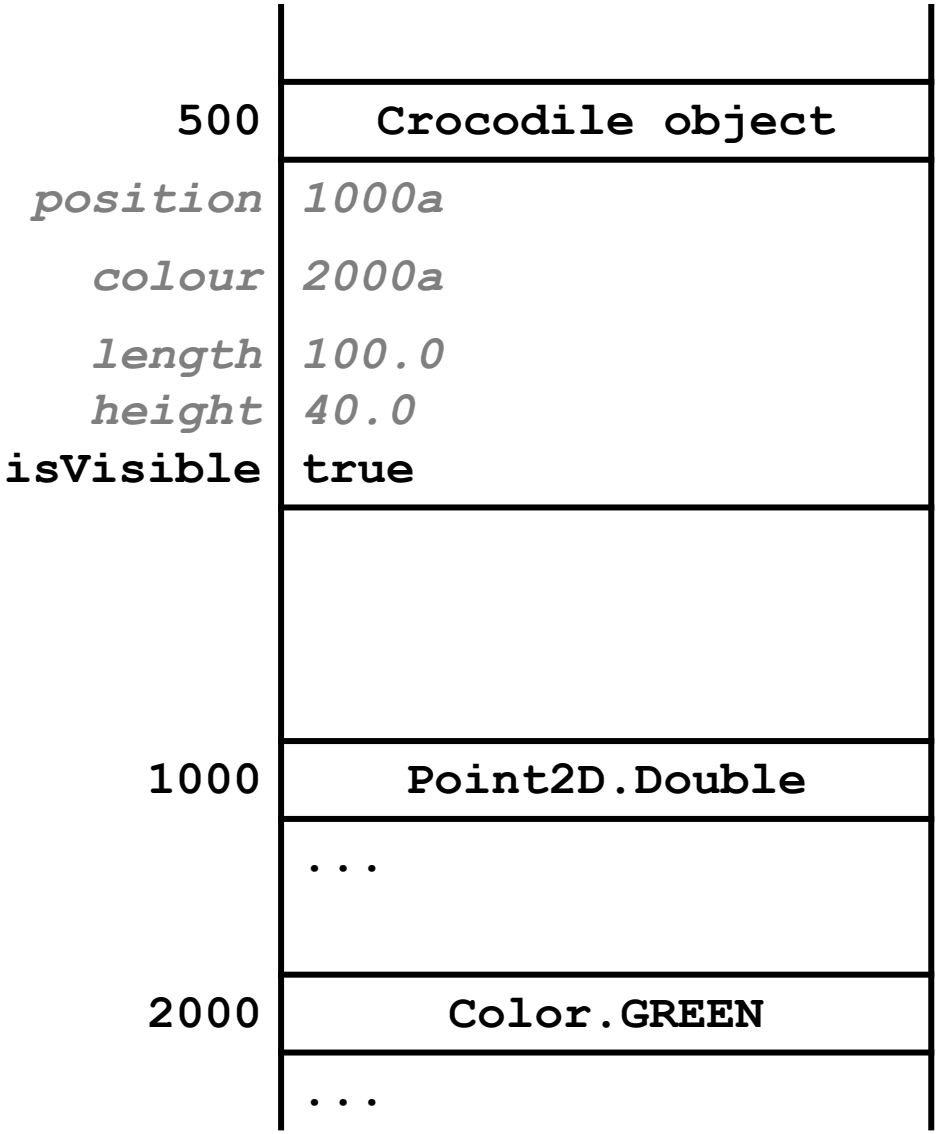
1. **`Crocodile`** constructor starts running
- creates new **`FloatyThing`** subobject by invoking the **`FloatyThing`** constructor
    2. **`FloatyThing`** constructor starts running
    - creates new **`Object`** subobject by (silently) invoking the **`Object`** constructor
        3. **`Object`** constructor runs
        - and finishes
    - sets **`position, colour, length`** and **`height`**
    - and finishes
- sets **`colour and isVisible`**
- and finishes



Crocodile Memory Diagram



Crocodile Memory Diagram



If FloatyThing fields were private?

accessible through inherited API (if there are getters/setters)

Storing and accessing polymorphic objects

(a little deeper look at polymorphism)

YORK
UNIVERSITÉ
UNIVERSITY

- Polymorphism
  - Substitutability & visibility
  - Overriding & dynamic dispatch
  - Exception hierarchy (is-a/substitution implications)

Recall: is-a == “is substitutable for”

- Hence we can assign instances of Log or Crocodile (subclasses of FloatyThing) to a FloatyThing reference:

```
FloatyThings[][] items = new FloatyThings[MAXROWS][MAXTHINGS];

items[0][0] = new Log(p, x, y);      // p is a Point2D.Double
                                     // x,y are doubles

items[0][2] = new Crocodile(p, x, y);
```

or we could also do this:

```
FloatyThing f1 = items[0][0];
FloatyThing f2 = items[0][2];

Object obj = f1;
obj = f2;

obj = new StringBuilder();  // why? Object is an ancestor of all classes
```

Declared vs. Actual Types

Declared (compile-time) vs Actual (run-time) types

```
Object obj = new Log();
```

- Declared type is the type of the reference variable
- Actual type is the type of the instance assigned to a reference variable

- Declared type is static – fixed.. i.e. cannot change
- Actual type is “dynamic” (i.e. can change)

Dynamic type

- Change is based on (determined by) a valid inheritance hierarchy!!
  - Classes:
    - A parent reference can be assigned any **instance** of a class that is in its sub-hierarchy (i.e. is a child of)
  - Interfaces:
    - An interface reference can be assigned any **instance** of a class that implements that interface

Object **obj** ← is “polymorphic”

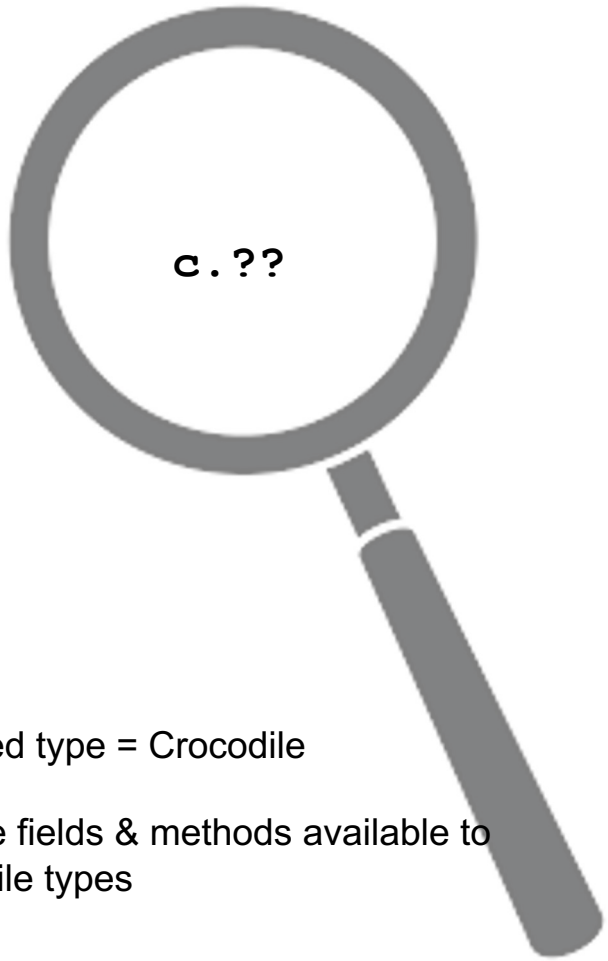
```
Object obj = new Log(…);

obj = new Crocodile(…);

obj = new StringBuilder(…);
```

- Polymorphism
  - having the ability to take on different “form”
  - having the ability to behave/act differently

What can we access after substitution?

- i.e. if parent reference is assigned a subclass instance.. what can we access exactly?
  - Think of the declared type as a “lens” on the object it is assigned …
  - The declared type determines what can be accessed
    - Specifically, the API of the reference type determines what can be “seen” or “accessed” via the reference

```
Crocodile c =
    new Crocodile(new Point2D.Double(), 100.0, 40.0);
```



Declared type = Crocodile

can see fields & methods available to Crocodile types

Crocodile object

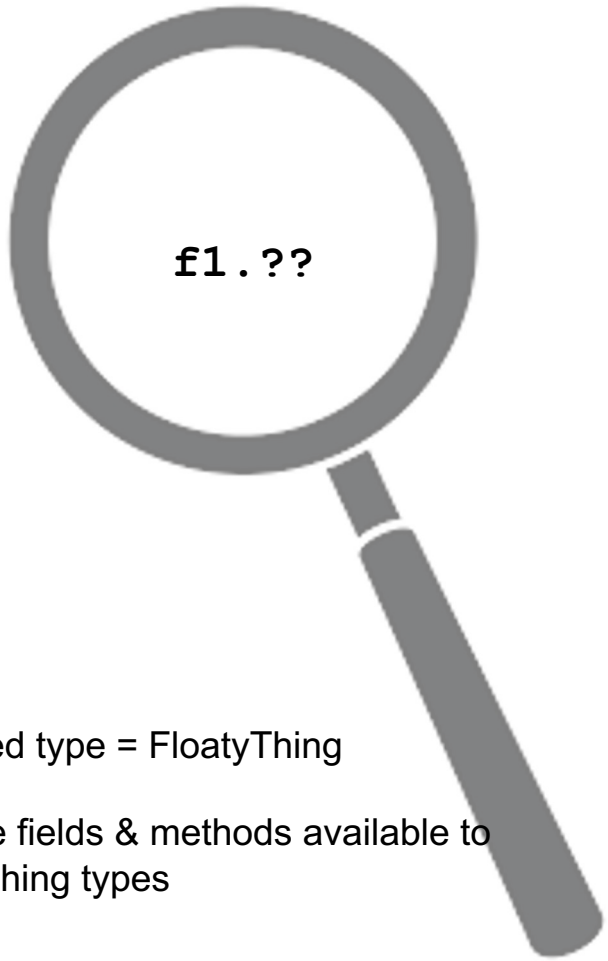
FloatyThing object

Object object

position	Address of Point2D object (0,0)
colour	Address of Color object (Color.green)
length	100.0
height	40.0

isVisible	true

```
FloatyThing f1 =
    new Crocodile(new Point2D.Double(), 100.0, 40.0);
```



Declared type = FloatyThing

can see fields & methods available to FloatyThing types

Crocodile object

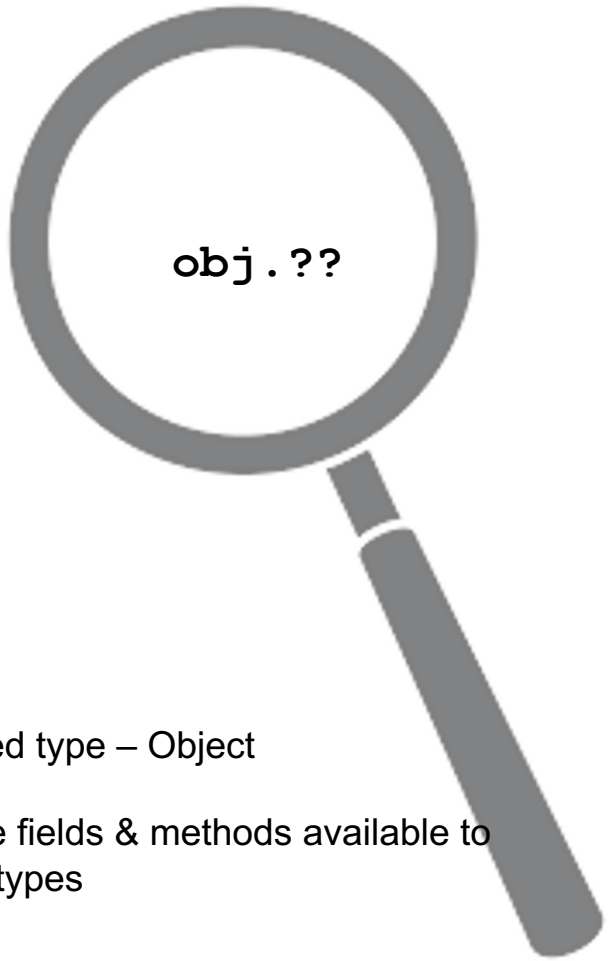
FloatyThing object

Object object

position	Address of Point2D object (0,0)
colour	Address of Color object (Color.green)
length	100.0
height	40.0

isVisible	true

```
Object obj =
    new Crocodile(new Point2D.Double(), 100.0, 40.0);
```



Declared type – Object

can see fields & methods available to Object types

Crocodile object

FloatyThing object

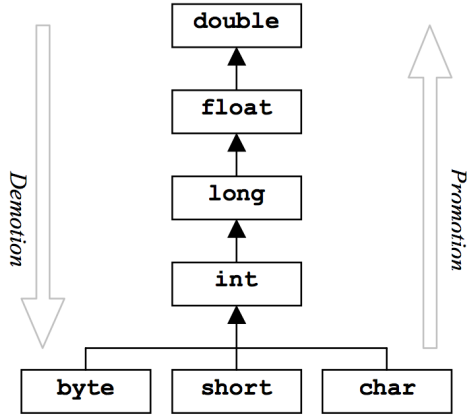
Object object

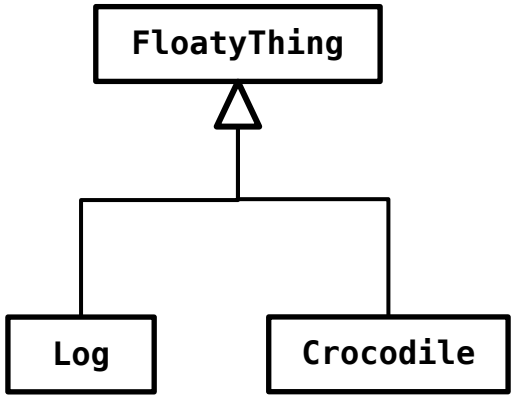
position	Address of Point2D object (0,0)
colour	Address of Color object (Color.green)
length	100.0
height	40.0

isVisible	true

Ok, so I store FloatyThing types, how do I access their specific instance properties?

- We note from the previous example:
  - subclass instances are automatically substitutable for parent/ancestor class references
  - Analogy: this is similar to idea of promotion (in primitive numeric types)..



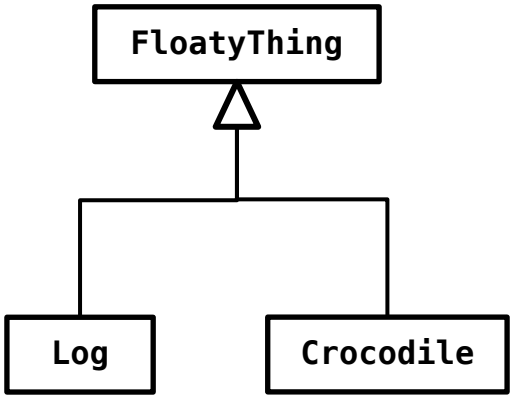


  - Is there a parallel to demotion??

Yes..
casting is possible with is-a relationships

… if certain conditions are met

- Can cast from one reference type to another if and only if:
  - The parent reference is referring to a valid child instance
  - We are casting down from the parent reference to the child instance (actual type) or an appropriate ancestor in between



Casting:
a parent reference to a child instance type

```
items[0][0] = new Log(p, x, y);          // items is a FloatyThing[][] type
                                         // p is a Point2D.Double
                                         // x,y are doubles

items[0][2] = new Crocodile(p, x, y);


// …

Crocodile c = (Crocodile) items[0][2];
    // can now access all Crocodile features through the reference c

Log l = (Log) items[0][0];
    // can now access all Log features through the reference l
```

```
Object obj =
    new Log(new Point2D.Double(), 100.0, 40.0);
```

Log object

FloatyThing object

Object object

position	Address of Point2D object (0,0)
colour	Address of Color object (Color.orange)
length	100.0
height	40.0

```
Object obj =
    new Log(new Point2D.Double(), 100.0, 40.0);
```


```
FloatyThing myFT =
      (FloatyThing) obj;
```

Log object

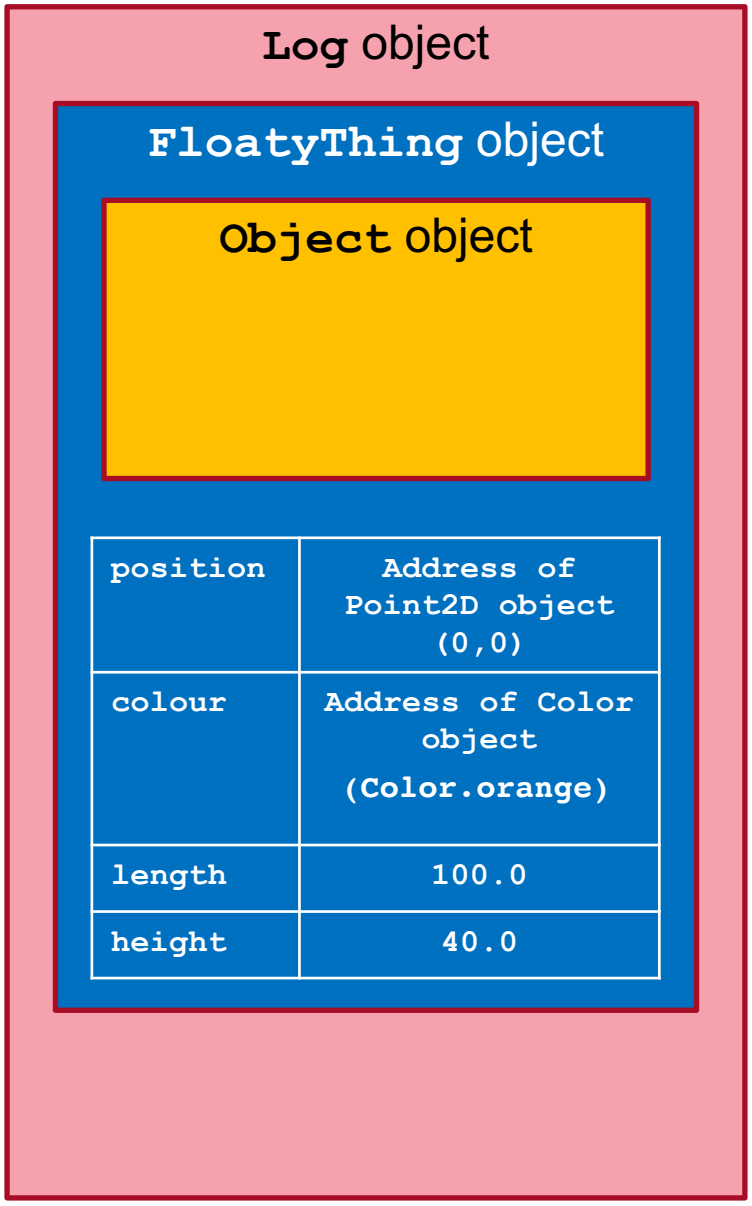
FloatyThing object

Object object

position	Address of Point2D object (0,0)
colour	Address of Color object (Color.orange)
length	100.0
height	40.0

```
Object obj =
    new Log(new Point2D.Double(), 100.0, 40.0);
```


```
Log myLog = (Log) obj;
```



How to ensure a cast will be possible? use “instanceof” operator

- Keyword “instanceof” allows you to test whether or not a reference is pointing to an instance of a specific class

```
Object obj = items[0][0];

if (obj instanceof  Log)       {
        // you can cast obj to Log safely
        Log l = (Log) obj;
}
```

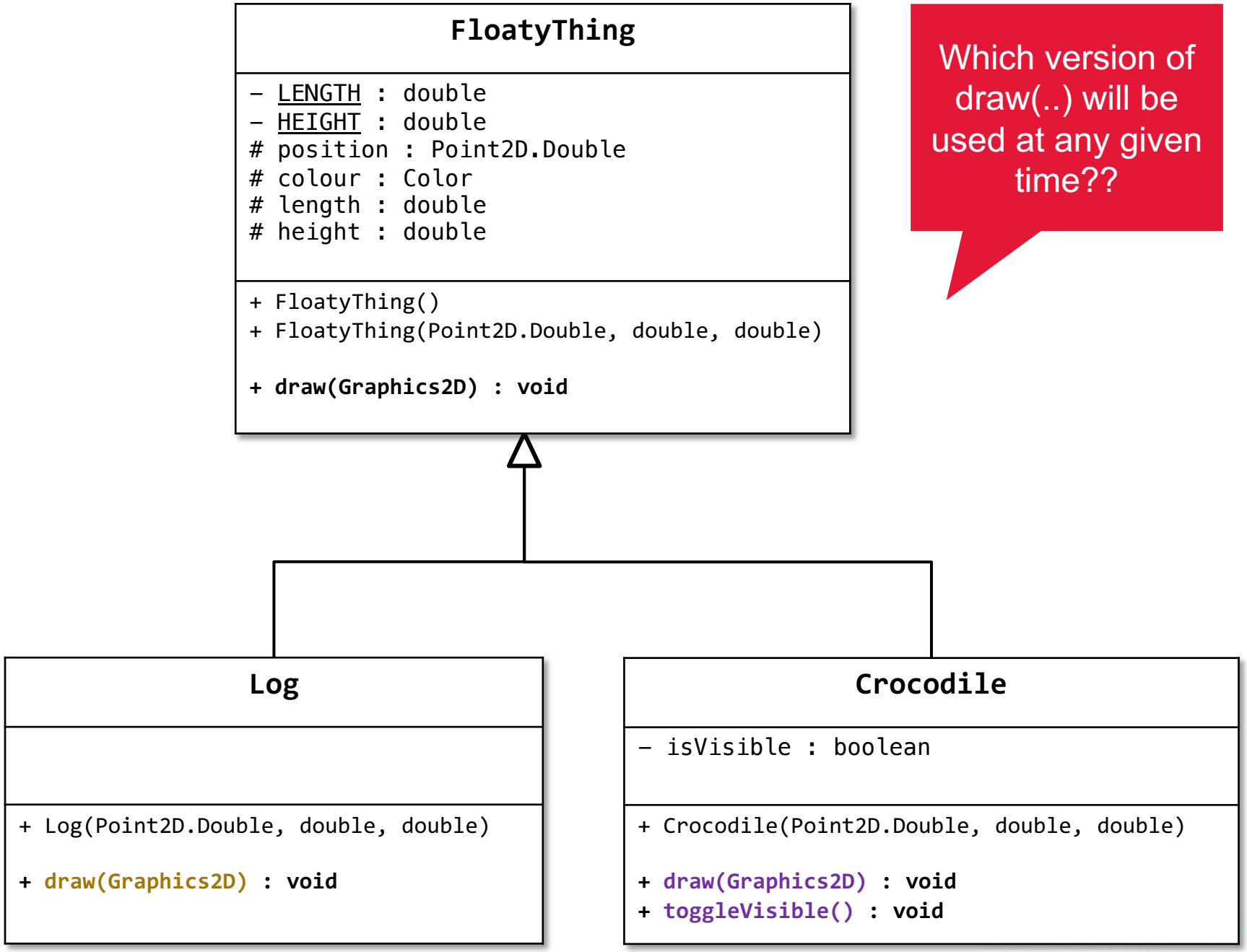
Checks if obj is referencing an actual Log instance (as opposed to Crocodile or other instance)

YORK UNIVERSITÉ UNIVERSITY

Dynamic Dispatch

Dynamic behaviour?

- We have seen examples of dynamic form (substitution of instances into parent references)
- We have seen that it is possible to override methods (a subclass can override a parent's method)
- What if all classes in an inheritance hierarchy override a given method?
    - Which one is used at any given time?


FloatyThing
- LENGTH : double
- HEIGHT : double
position : Point2D.Double
colour : Color
length : double
height : double
+ FloatyThing()
+ FloatyThing(Point2D.Double, double, double)
+ draw(Graphics2D) : void
Which version of draw(..) will be used at any given time??
Log
+ Log(Point2D.Double, double, double)
+ draw(Graphics2D) : void
Crocodile
- isVisible : boolean
+ Crocodile(Point2D.Double, double, double)
+ draw(Graphics2D) : void
+ toggleVisible() : void

Dynamic dispatch

- If the declared type is assigned a child instance and is used to invoke a method..

1. The method must exist in the declared type's API
    - draw(..) method exists in FloatyThing api, so can be invoked from a FloatyThing reference (i.e. it has to be a visible feature)

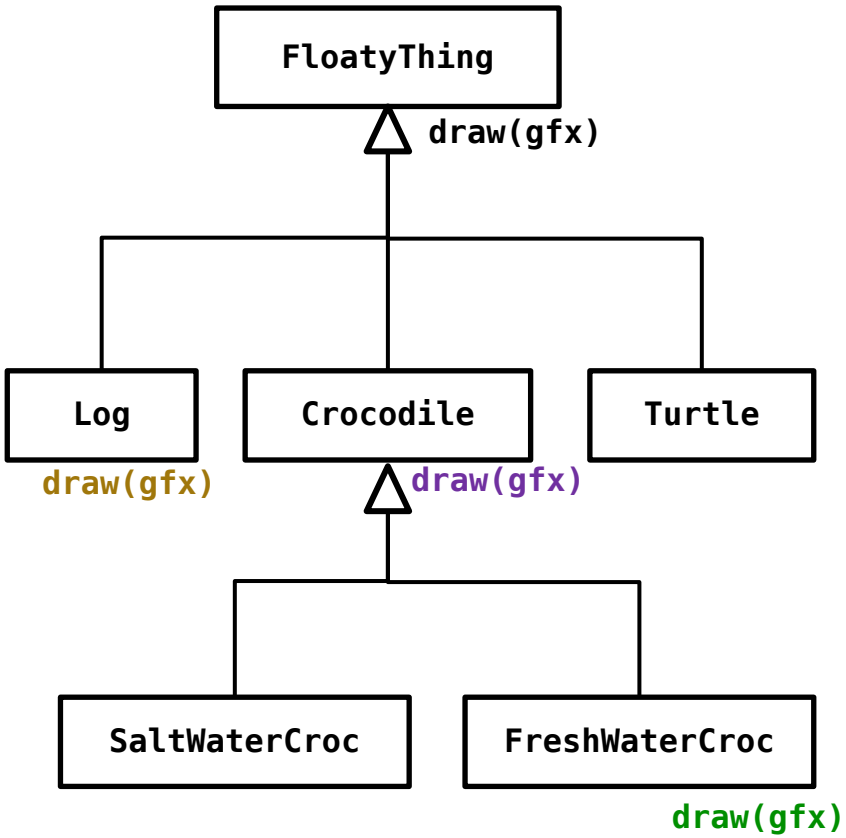
2. If the method is overridden, the version of the method in actual type's API is run
    - draw(…) method is overridden by Crocodile class, so the invocation of FloatyThing's draw(..) method will be re-routed to Crocodile's version of the draw(…) method

    - This re-routing of the method call is termed "dynamic dispatch"

Example:

```
items[0][0] = new Log(p, x, y);
items[0][1] = new FloatyThing();
items[0][2] = new Crocodile(p, x, y);


// note, items[i][j] are always FloatyThing references
// assume we have access to a Graphics2D reference called gfx


items[0][0].draw(gfx);   // dispatched to Log version of draw(..)

items[0][1].draw(gfx);   // no dispatch->FloatyThing version of draw(..)

items[0][2].draw(gfx);   // dispatched to Crocodile version of draw(..)
```

Implications?

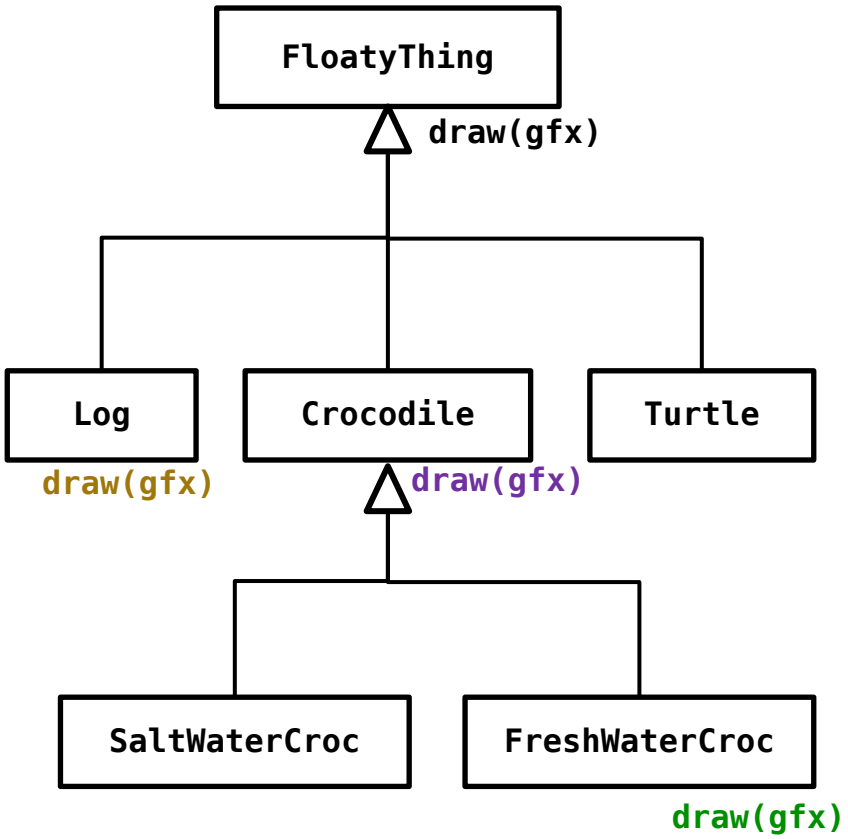



```
FloatyThing f = new Crododile(..);
f.draw(gfx);  // which version?

f = new Turtle(..);
f.draw(gfx);  // which version?

f = new FreshWaterCroc(..);
f.draw(gfx);  // which version?

f = new SaltWaterCroc(..);
f.draw(gfx);  // which version?
```

Diagram indicates classes for which draw(..) is overridden

YORK UNIVERSITÉ UNIVERSITY

Implications?

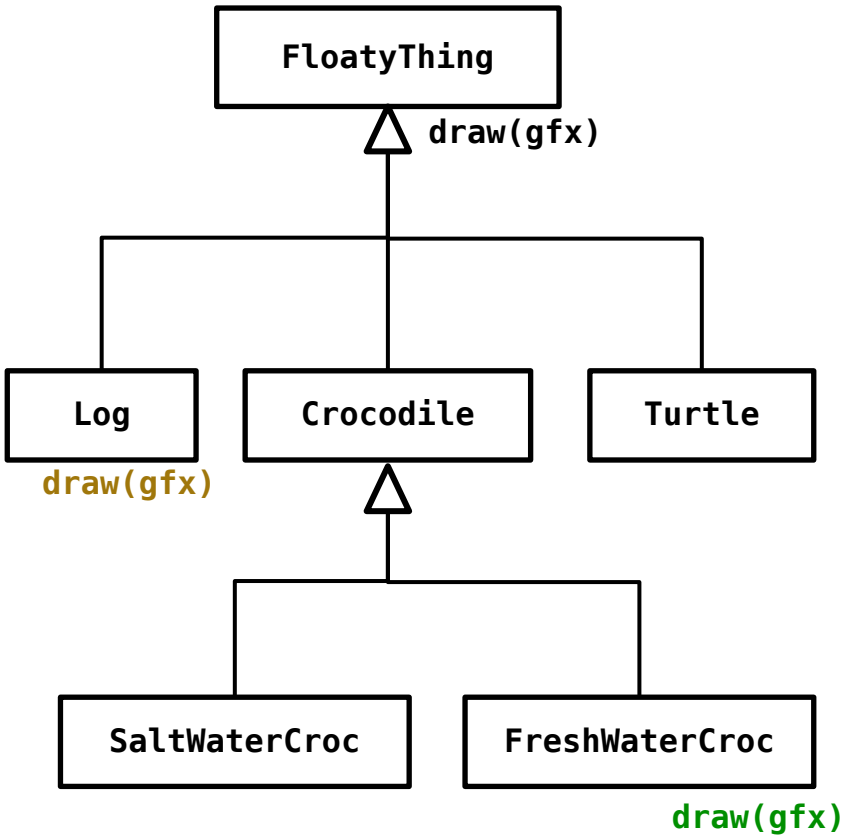



```
FloatyThing f = new Crododile(..);
f.draw(gfx);  // which version?

f = new Turtle(..);
f.draw(gfx);  // which version?

f = new FreshWaterCroc(..);
f.draw(gfx);  // which version?

f = new SaltWaterCroc(..);
f.draw(gfx);  // which version?
```

Diagram indicates classes for which draw(..) is overridden

YORK UNIVERSITÉ UNIVERSITY

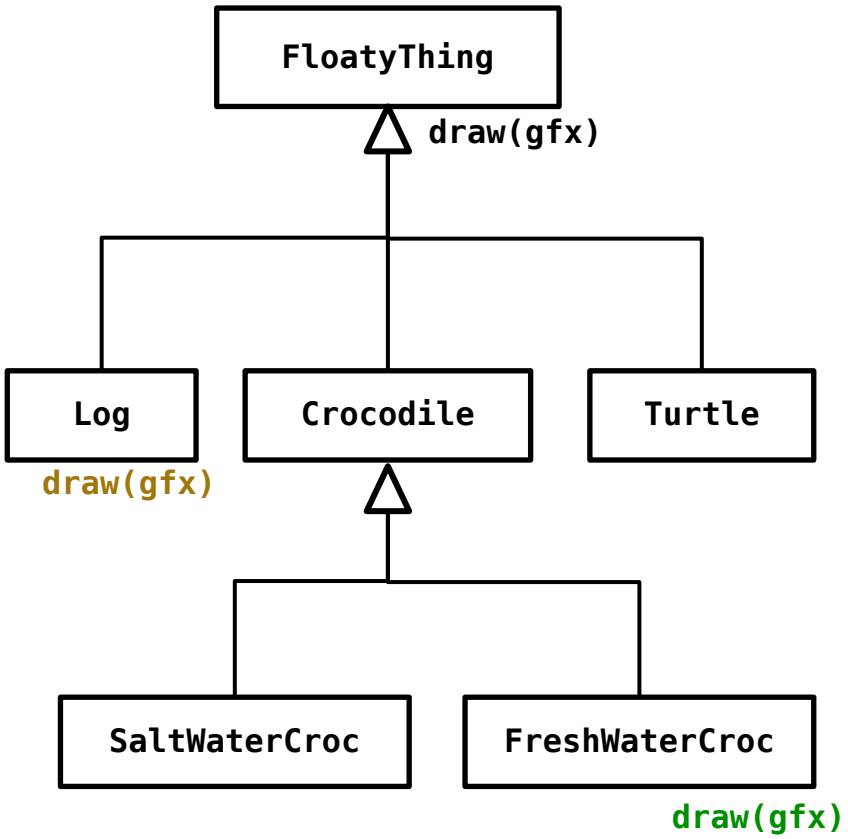
Implications?




```
FloatyThing f = new Crododile(..);
f.draw(gfx);  // which version?

f = new Turtle(..);
f.draw(gfx);  // which version?

f = new FreshWaterCroc(..);
f.draw(gfx);  // which version?

f = new SaltWaterCroc(..);
f.draw(gfx);  // which version?
```

Diagram indicates classes for which draw(..) is overridden

Implications?




```
FloatyThing f = new Crododile(..);
f.draw(gfx);  // which version?

f = new Turtle(..);
f.draw(gfx);  // which version?

f = new FreshWaterCroc(..);
f.draw(gfx);  // which version?

f = new SaltWaterCroc(..);
f.draw(gfx);  // which version?
```

Diagram indicates classes for which draw(..) is overridden

YORK UNIVERSITÉ UNIVERSITY

Binding

- We say that a method is “bound” to a reference
  - Early binding (at compile time)
    - Fixed.. Always bound to same method,
    - No dynamic dispatch
  - Late binding (at run time)
    - Dynamic dispatch
    - Depends on actual type assigned to reference at run time

Recap: Polymorphism

- *inheritance* allows you to define a base class that has fields and methods
  - classes derived from the base class can use the public and protected base class fields and methods

- ***polymorphism*** allows the implementer to change the ***form*** and ***behaviour*** of the derived class methods

INTERFACES
(more details)

Recall: Interfaces

- An interface is like a class, except it can only contain method signatures, and fields
  - While it *can* contain fields, we usually use interfaces to encapsulate *methods* only
- A class that "*implements*" an interface must provide definitions for all methods declared by the interface
  - Interfaces only "declare" these features, never implements
  - We create a class that then does the implementation of the shell methods declared by the interface

Interfaces (how they are specified)

```
public interface MyInterface {

    public String hello = “Hello”;
    public void sayHello();

}
```

Note:

- keyword **interface** used instead of **class**

- no constructors (cannot instantiate an interface type)
- if there are fields, they are forced to be public static (even if not specified)
- no method definitions (method declarations/signature only)

Interfaces **may not be instantiated**

- However …
  - A class may **implement** an interface
  - The class may then be instantiated

- What does *implementing* mean?
  - It means that the class is expected to define (provide) the methods specified by the **interface**

  - In this way, the interface can be used to force a class to adhere to a set of behaviours
    - An interface effectively declares a desired API
    - A class that implements an interface, fulfills that API

A class implements an interface

```
public class MyInterfaceImpl implements MyInterface {

     public void sayHello() {
              System.out.println(MyInterface.hello);
     }

}
```

Note:

- If the interface uses fields, they must be accessed via the interface name
  - Fields are always public (no defined methods)
  - Fields are always static (no instantiation)

- All methods have no implementation (considered public & are “abstract”)
  - Abstract means the definition is missing (so the method implementation is not defined/concrete)

- An interface may not really be used until it is implemented!

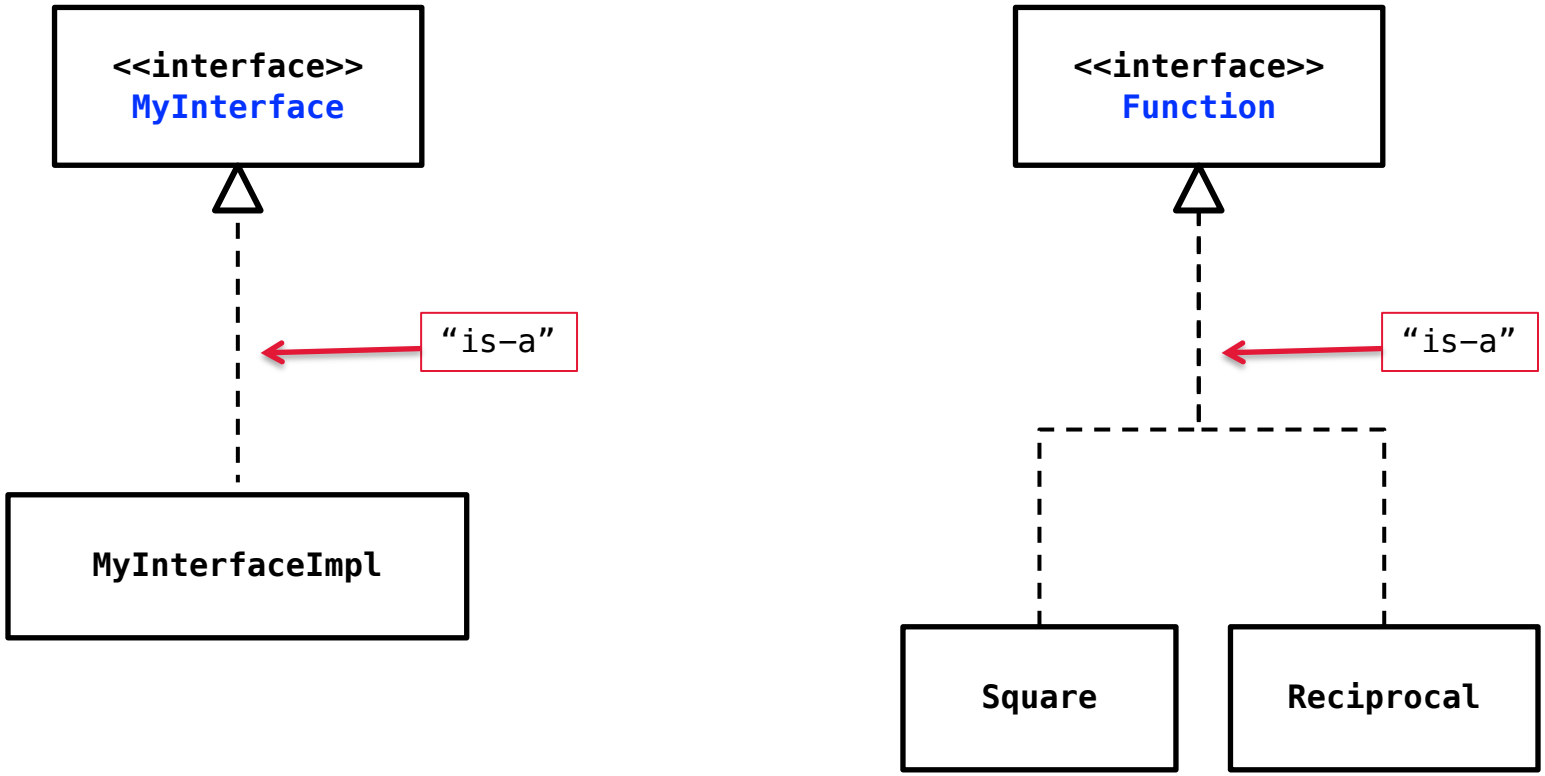
Interfaces (example)

- consider an interface for mathematical functions of the form

$F(x) = x^2$
$F(x) = 1/x$

(want to handle single or multiple x's)

Recall: (other interface examples)



```
public interface Function {

    /**
     * Evaluate the function at x.
     *
     * @param x the value at which to evaluate the function
     * @return the value of the function evaluated at x
     */
    public double eval(double x);

    /**
     * Evaluate the function at each value of x in the given list.
     *
     * @param x an array of values at which to evaluate the function
     * @return the array of values of the function evaluated at the given
     * values of x
     */
    public Double[] eval(Double[] x);
}
```

semicolon, and no method body

semicolon, and no method body

Interfaces

- notice that the interface declares which methods exist and specifies expectations for its inputs/outputs
  - but it does *not* specify *how* the methods are implemented

- the method implementations are defined by **classes** that ***implement*** the interface

Interfaces are types

- an interface is a reference data type
  - if you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface
    (https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html)

```
Function f1 = new Square();
Function f2 = new Reciprocal();
```

interface

classes that implement the interface

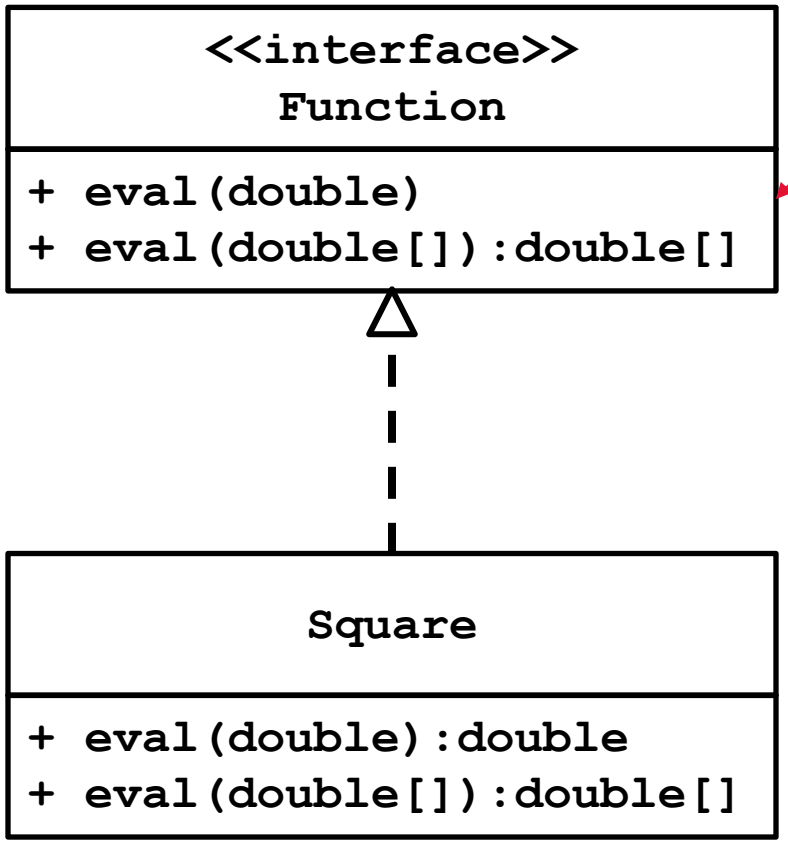
```
public class Square implements Function {

    public Square() {}


    // override
    public double eval(double x) {
        return x * x;
    }

    // override
    public double[] eval(double[] x) {
        double[] result = new double[x.length];
        for (int i=0; i<x.length;  i++) {
            result[i] = this.eval(x[i]);
        }
        return result;
    }

}
```

Square implements the **Function** interface

Square must provide an implementation of **eval(double)**

Square must provide an implementation of **eval(double[])**

Interfaces (UML)



these methods are declared only (only signature & return type is specified)

their “implementation” is left to the class that ***implements***

these methods ARE defined concretely (i.e. implementation is defined)

They “override” the methods specified in the interface

Overriding vs. Overloading

- Recall: method "**overloading**"
  - Several methods of the same name (but different signature)
  - i.e. we have several "versions" of the same method/constructor that support alternative arguments

- method "**overriding**" ?
  - A (new) implementation of a declared/existing method
  - Occurs in interfaces & inheritance

  - In interfaces:
    - a class implementing an interface must provide a full definition of a declared method
    - the new definition replaces/"overrides" the existing declaration

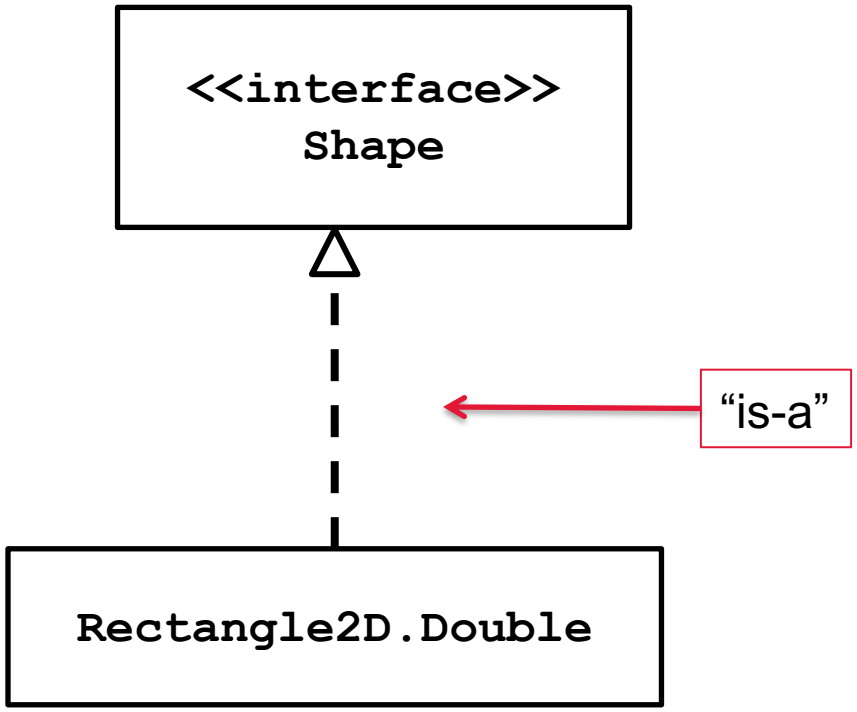
```
public class Reciprocal implements Function {

    public Reciprocal() {}


    // override
    public double eval(double x) {
        return 1.0 / x;
    }


    // override
    public Double[] eval(Double[] x) {
        Double[] result = new Double[x.length];
        for (int i=0; i<x.length;  i++) {
            result[i] = this.eval(x[i]));
        }
        return result;
    }

}
```

Reciprocal implements the **Function** interface

Reciprocal must provide an implementation of **eval(double)**

Reciprocal must provide an implementation of **eval(Double[])**

Interfaces (UML)



Rectangle2D.Double has the ability to "override" methods declared in Shape

A Shape reference can safely be used to invoke a method declared by the Shape interface..

Because we know that the implementing class has provided an implementation of that method that can be run!

```java
public interface Function {
    /**
     * Evaluate the function at x.
     */
    public double eval(double x);

    /**
     * Evaluate the function at each
     * value of x in the given list.
     */
    public Double[] eval(Double[] x);
}
```

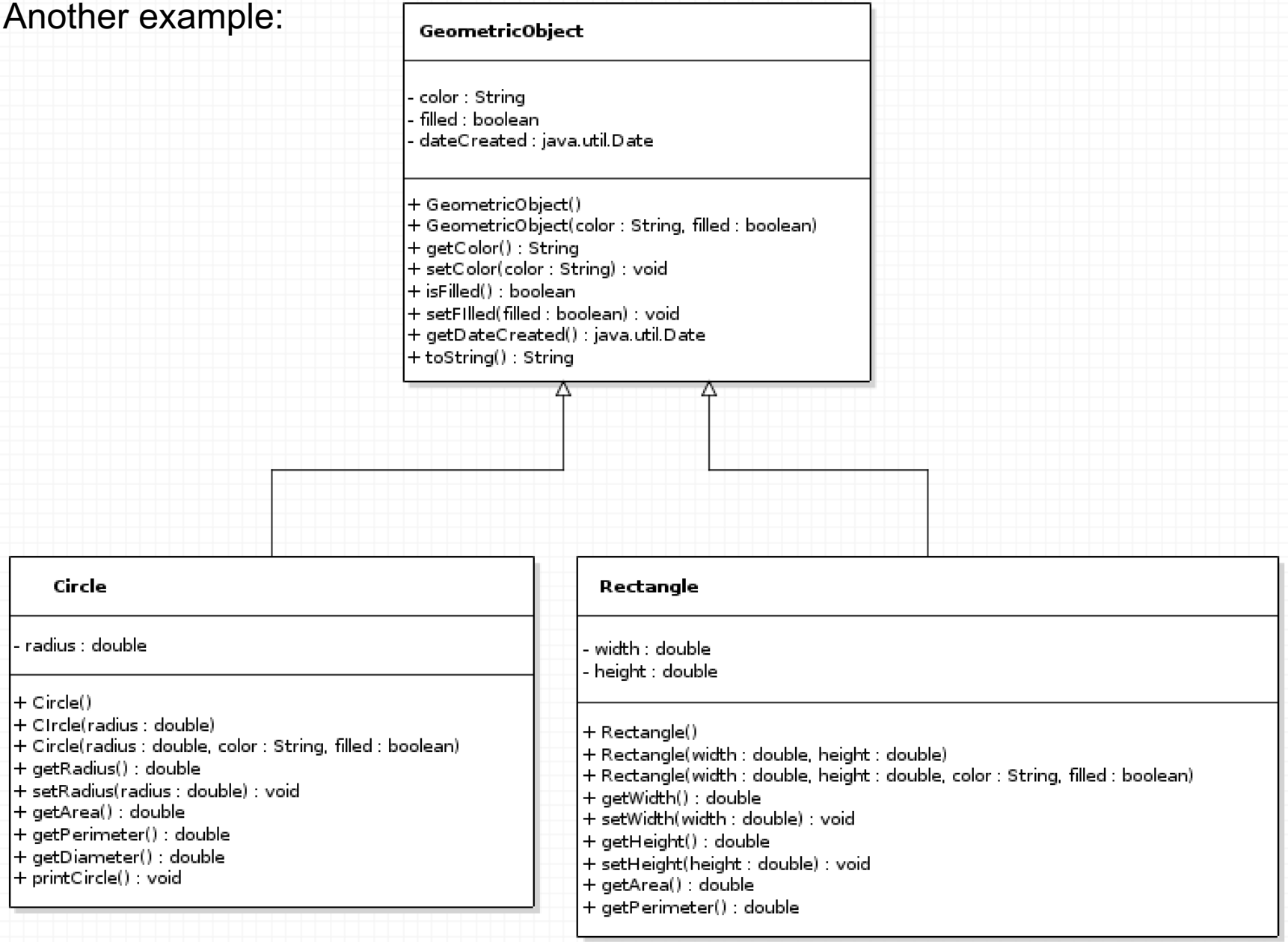
```java
public class Square implements Function {

    public Square() {}

    // override
    public double eval(double x) {
        return x * x;
    }

    // override
    public Double[] eval(Double[] x) {
        Double[] result = new Double[x.length];
        for (int i=0; i<x.length;  i++) {
            result[i] = this.eval(x[i]);
        }
        return result;
    }
}
```

- Recall: method "**overloading**"
  - Several methods of the same name (but different signature)
  - i.e. we have several "versions" of the same method/constructor that support alternative arguments

- method "**overriding**" ?
  - A (new) implementation of a declared/existing method
  - In interfaces: a class implementing an interface must provide a full definition of a declared method. The new definition replaces/"overrides" the existing declaration

INTERFACES vs. INHERITANCE?

Another example:



Base (super) class :: GeometricObject

```
public class GeometricObject {

    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;


    /** Construct a default geometric object */
    public GeometricObject() {
        this.filled = false;
        this.dateCreated = new java.util.Date();
    }

    /** Construct a custom geometric object */
    public GeometricObject(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
        this.dateCreated = new java.util.Date();
    }


    // …
```

Base (super) class :: GeometricObject

```
// …

    /** Return color */
    public String getColor() {
        return this.color;
    }

    /** Set a new color */
    public void setColor(String color) {
        this.color = color;
    }

    /** Return filled. Since filled is boolean, its getter method
    is named isFilled */
    public boolean isFilled() {
        return this.filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }
```

Base (super) class :: GeometricObject

```
// …


    /** Get dateCreated */
    public java.util.Date getDateCreated() {
        return this.dateCreated;
    }


    /** Return a string representation of this object */
    public String toString() {
        return "created on " + this.dateCreated + "\ncolor: " +
        this.color + " and filled: " + this.filled;

    }

}
```

Sub-class :: Circle

```
public class Circle extends GeometricObject {

    private double radius;



    /** Construct a default Circle object */
    public Circle() {

    }

    /** Construct a custom Circle object */
    public Circle (double radius, String color, boolean filled) {
        this.radius = radius;

        this.color = color;
        this.filled = filled;
    }

    // …

}
```

Implicit call to super() if not explicitly included

PROBLEM – cannot access private fields

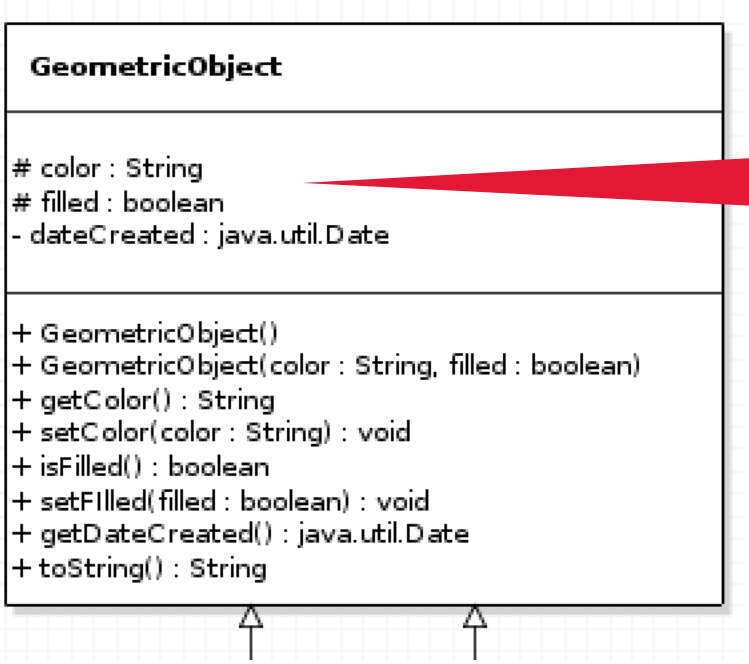
YORK UNIVERSITÉ UNIVERSITY

Sub-class :: Circle (solution 1)

```
public class Circle extends GeometricObject {

    private double radius;



    /** Construct a default Circl
    public Circle() {

    }

    /** Construct a custom Circle object */
    public Circle (double radius, String color, boolean filled) {
        this.radius = radius;

        this.color = color;
        this.filled = filled;
    }

    // …

}
```

```
public class GeometricObject {

    protected String color = "white";
    protected boolean filled;
    private java.util.Date dateCreated;

    // …
```

Make fields “protected” in base class (they will act like “public” fields to all derived sub-classes, but act “private” to any external client of GeometricObject)



"protected" fields represented with access as "#" in UML diagram

Sub-class :: Circle (solution 2)

```
public class Circle extends GeometricObject {

    private double radius;



    /** Construct a default Circl
    public Circle() {

    }

    /** Construct a custom Circle object */
    public Circle (double radius, String color, boolean filled) {
        this.radius = radius;
        this.setColor(color);
        this.setFilled(filled);
    }

    // …

}
```

```
public class GeometricObject {

    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    // …
```

Keep base class fields private & Use public API (inherited) to mutate color and filled fields of the Circle object

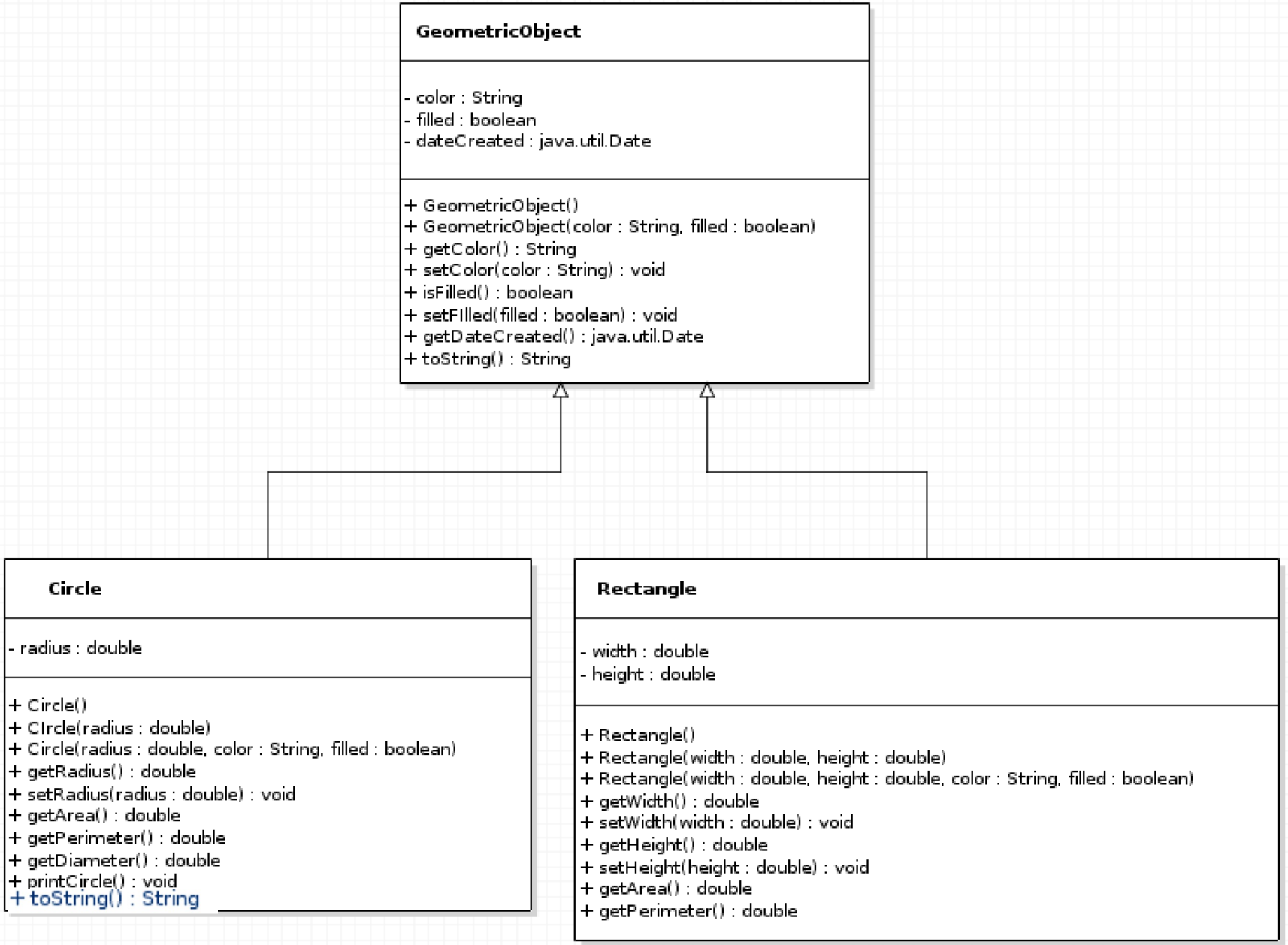
```
    // …

    /** Return radius */
    public double getRadius() {
        return this.radius;
    }
    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }
    /** Return area */
    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
    /** Return diameter */
    public double getDiameter() {
        return 2 * this.radius;
    }
    /** Return perimeter */
    public double getPerimeter() {
        return 2 * this.radius * Math.PI;
    }
    /** Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created " +
                this.getDateCreated() + " and the radius is " +
                this.getRadius() );
    }
}
```

Note: This method is part of API inherited from super class

sub-classes can override super-class methods

- E.g. Want to run a different version of toString() that is specific to Circle objects?

```
public class Circle extends GeometricObject {

// … implementation shown on previous slides


    /** Override the toString() method */
    /** Return a string representation of a Circle object */

    @Override
    public String toString() {
        return "Circle (r=" + radius + ") created on " +
        this.getDateCreated() + "\ncolor: " + this.getColor() + "
        and filled: " + this.getFilled();

    }
}
```


GeometricObject
- color : String
- filled : boolean
- dateCreated : java.util.Date
+ GeometricObject()
+ GeometricObject(color : String, filled : boolean)
+ getColor() : String
+ setColor(color : String) : void
+ isFilled() : boolean
+ setFIlled(filled : boolean) : void
+ getDateCreated() : java.util.Date
+ toString() : String
Circle
- radius : double
+ Circle()
+ CIrcle(radius : double)
+ Circle(radius : double, color : String, filled : boolean)
+ getRadius() : double
+ setRadius(radius : double) : void
+ getArea() : double
+ getPerimeter() : double
+ getDiameter() : double
+ printCircle() : void
+ toString() : String
Rectangle
- width : double
- height : double
+ Rectangle()
+ Rectangle(width : double, height : double)
+ Rectangle(width : double, height : double, color : String, filled : boolean)
+ getWidth() : double
+ setWidth(width : double) : void
+ getHeight() : double
+ setHeight(height : double) : void
+ getArea() : double
+ getPerimeter() : double

A subclass can also invoke the super class version of a method using the “super” keyword

- Recall, “super” is a reference to the superclass part of the current object, so through it we can access super class methods
- Code below achieves same result as previous slide!

```
public class Circle extends GeometricObject {

// … implementation shown on previous slides


    /** Override the toString() method */
    /** Return a string representation of a Circle object */

    @Override
    public String toString() {
        return "Circle (r=" + this.radius + ") " + super.toString();

    }
}
```

Forces invocation of GeometricObject’s version of the toString() method !!

Sub-class :: Rectangle

```
public class Rectangle extends GeometricObject {

    private double width;
    private double height;

    /** Construct a default Rectangle object */
    public Rectangle() {

    }
    /** Construct a custom Rectangle object */
    public Rectangle (double height, double width) {
        this.width = width;
        this.height= height;
    }
    /** Construct a custom Rectangle object */
    public Rectangle (double height, double width, String color,
    boolean filled) {
        this.width = width;
        this.height= height;
        this.setColor(color);
        this.setFilled(filled);
    }

    // …
}
```

Note: in all 3 ctor's, super() is implicitly invoked first

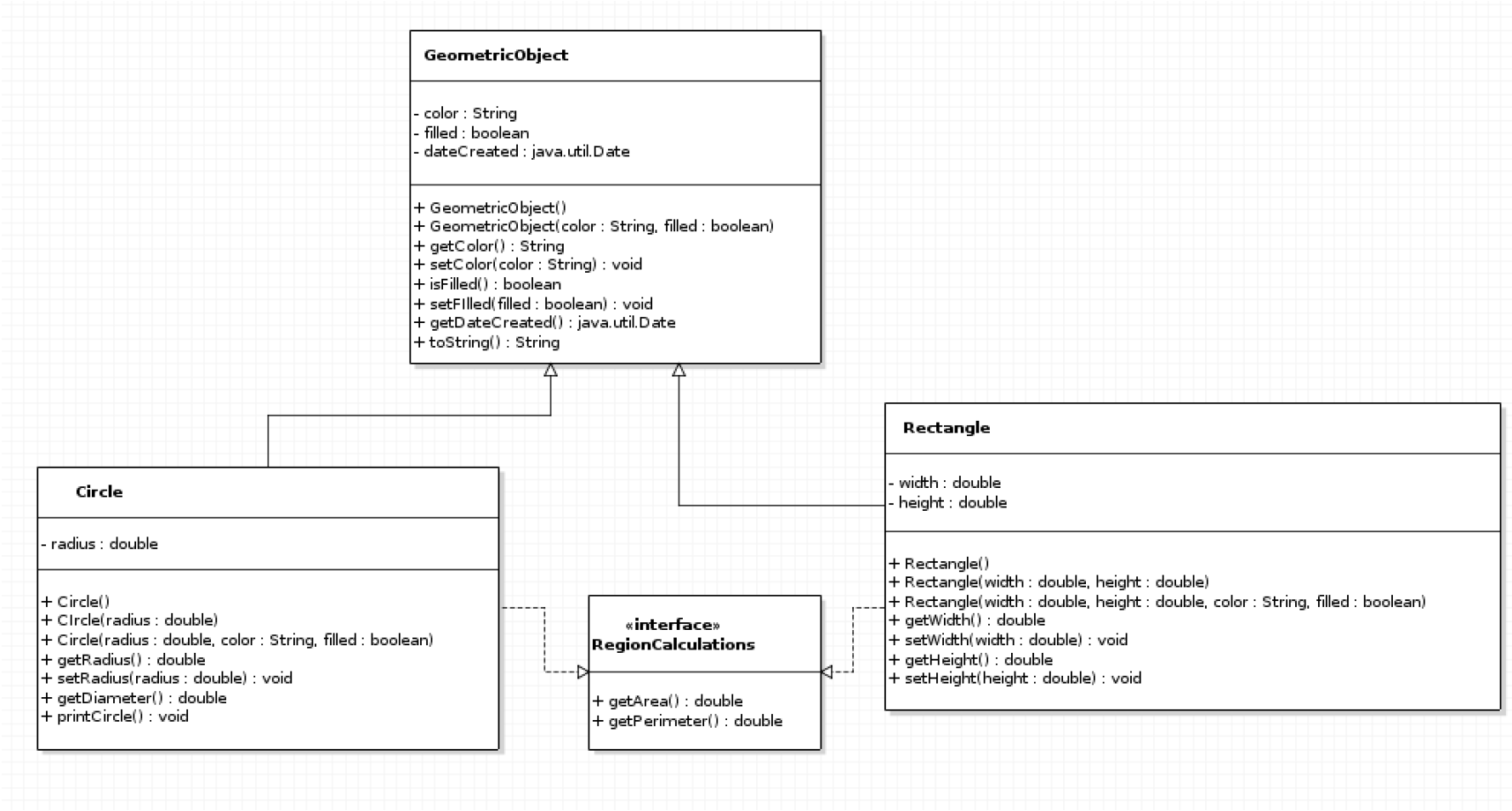
YORK UNIVERSITÉ UNIVERSITY

```
    // …

    /** Return width */
    public double getWidth() {
        return width;
    }
    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }
    /** Return height */
    public double getHeight() {
        return height;
    }
    /** Set a new width */
    public void setHeight(double height) {
        this.height = height;
    }
    /** Return area */
    public double getArea() {
        return this.width*this.height;
    }
    /** Return perimeter */
    public double getPerimeter() {
        return 2*(this.width + this.height);
    }
}
```

getArea and getPerimeter appear to be part of a common *API (i.e. common interface)*

YORK UNIVERSITÉ UNIVERSITY

getArea and getPerimeter as interface



YORK UNIVERSITÉ UNIVERSITY

Interface approach

```
public class Circle extends GeometricObject implements
RegionCalculations {
    // …
    /** Return area */
    public double getArea() {
        return (this.radius * this.radius * Math.PI);
    }
    /** Return perimeter */
    public double getPerimeter() {
        return 2 * this.radius * Math.PI;
    }
    // …

}
```

```
public class Rectangle extends GeometricObject implements
RegionCalculations {
    // …
    /** Return area */
    public double getArea() {
        return (this.width*this.height);
    }
    /** Return perimeter */
    public double getPerimeter() {
        return 2*(this.width + this.height);
    }
    // …
}
```

What can the interface reference “access”?

```
RegionCalculations r = new Circle();
r.getArea();
r.getPerimeter();

r = new Rectangle();
r.getArea();
r.getPerimeter();
```

- Only these?
  - Yes because r is a reference to a RegionCalculations type
  - r only has access to its visible API (two methods)
  - r can invoke these methods because they will be dynamically dispatched to the implemented versions of these methods (implemented by the actual types)

Exception hierarchy (polymorphism example)

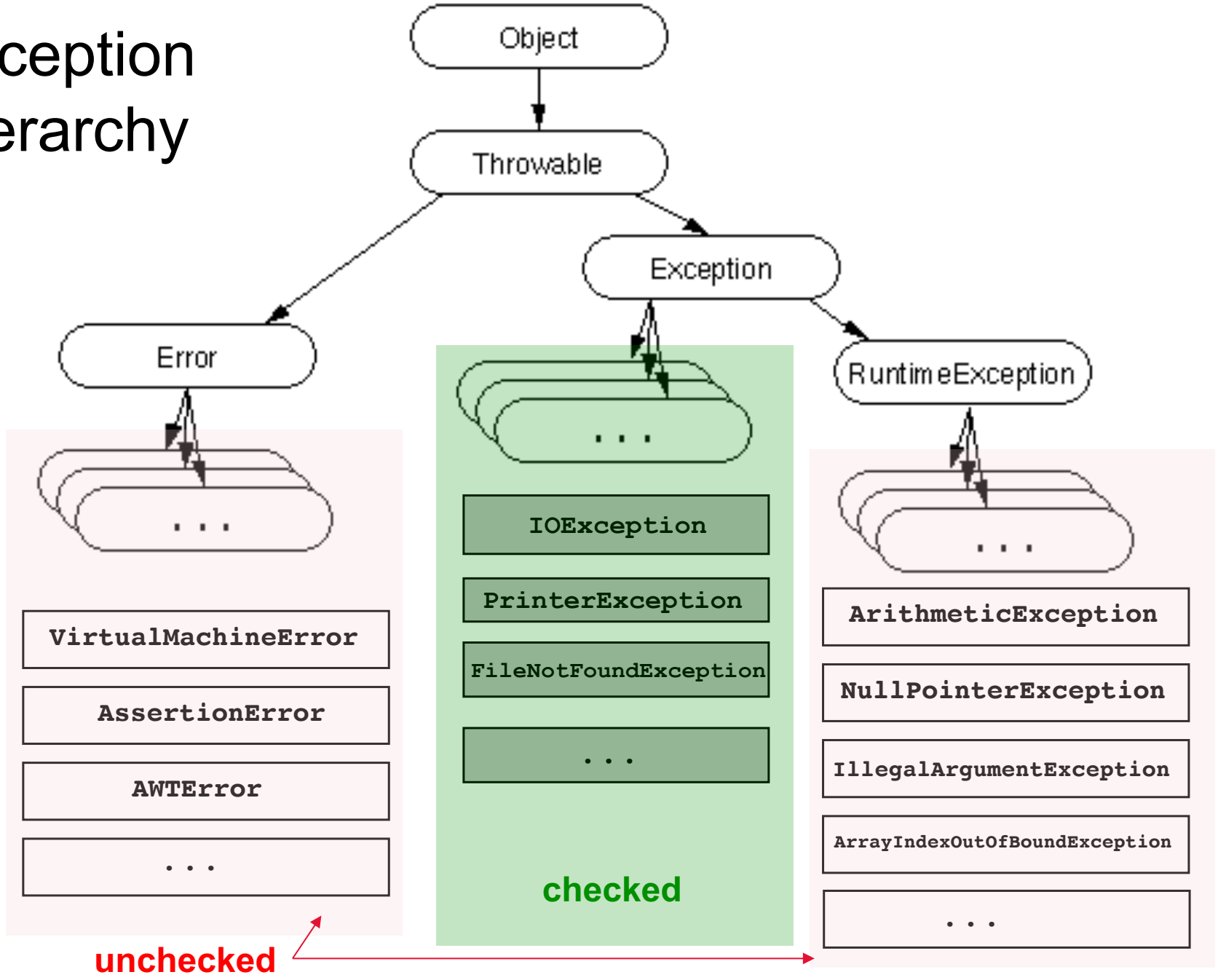
Exceptions (revisited): insert try/catch block

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // do an echo of input file (i.e. read all lines and output them to screen)
               Scanner inF = new Scanner(inFile);
               String oneLineText;
               System.out.println("Contents of file:");
               System.out.println("***************************");

               while (inF.hasNextLine()) {
                    oneLineText = inF.nextLine();
                    System.out.println(oneLineText);
               }
               inF.close();  // close the file after reading!!
          }
          catch (FileNotFoundException e) {
               // handle it
          }
     }
}
```

… previously:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // code to read file not shown

               inF.close();  // close the file after reading!!
          }
          catch (FileNotFoundException e) {
               // handle it
          }
          catch (NullPointerException e) {
               // handle it
          }
          catch (NoSuchElementException e) {
               // handle it
          }

     }
}
```

Exception Hierarchy



Catching a super-class (e.g. Exception)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // code to read file not shown

               inF.close();  // close the file after reading!!
          }
          catch (Exception e) {
               // captures any Exception
          }
          catch (NullPointerException e) {
               // captures NullPointerException only
          }
          catch (NoSuchElementException e) {
               // captures NoSuchElementException only
          }

     }
}
```

Catching a super-class (e.g. Exception)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // code to read file not shown

               inF.close();  // close the file after reading!!
          }
          catch (NullPointerException e) {
               // captures NullPointerException only
          }
          catch (NoSuchElementException e) {
               // captures NoSuchElementException only
          }
          catch (Exception e) {
               // captures any Exception (not yet caught
          }

     }
}
```

Order matters!

Catching a super-class (e.g. Exception)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // code to read file not shown
               inF.close();  // close the file after reading!!
          }
          catch (Exception e) {
               // captures any Exception (not yet caught)

               if (e instanceOf NullPointerException) {
                    // handle null pointer exception
               }
               if (e instanceOf FileNotFoundException) {
                    // handle file not found exception
               }
               // etc
          }
     }
}
```

Can catch any, then test for actual type

We can extend Exception classes

- Say we want to make a new RuntimeException?

```
public class MyRuntimeException extends RuntimeException {

     // add some new and additional fields here
     // to extend functionality of a typical exception


     public MyRuntimeException() {

          // implicitly invokes RuntimeException()
     }


     public MyRuntimeException(String message) {

          super(message);     // invokes RuntimeException(message);
     }

}
```