

<div data-bbox="53 159 613 284" data-label="Section-Header"> <h1>CS2030 Lecture 2</h1> <h2>Abstraction and Encapsulation</h2> </div> <div data-bbox="53 363 618 405" data-label="Text"> <p>Henry Chia (hchia@comp.nus.edu.sg)</p> </div> <div data-bbox="53 467 369 502" data-label="Text"> <p>Semester 2 2022 / 2023</p> </div>	<div data-bbox="1173 19 2027 67" data-label="Section-Header"> <h1>Abstraction in Object-Oriented Design</h1> </div> <div data-bbox="1173 132 2141 705" data-label="List-Group"> <ul style="list-style-type: none"> □ Consider a point object: <ul style="list-style-type: none"> – data abstraction <ul style="list-style-type: none"> ▸ e.g. a point comprises two floating-point values <ul style="list-style-type: none"> ▪ double x; double y; or ▪ <code>ImList<Double> coord;</code> or ▪ <code>Pair<Double, Double> pair; ...</code> – functional abstraction <ul style="list-style-type: none"> ▸ e.g. a point can determine the distance from <i>itself</i> to another given point <ul style="list-style-type: none"> ▪ <code>p.distanceTo(q)</code> or <code>q.distanceTo(p)</code>, where p and q are referring to Point objects </div>
<div data-bbox="1025 746 1086 770" data-label="Page-Footer"> <p>1 / 16</p> </div>	<div data-bbox="2145 746 2206 770" data-label="Page-Footer"> <p>3 / 16</p> </div>
<div data-bbox="53 817 761 865" data-label="Section-Header"> <h2>Outline and Learning Outcomes</h2> </div> <div data-bbox="53 930 990 1428" data-label="List-Group"> <ul style="list-style-type: none"> □ Be able to transition from data-process to object-oriented modeling and programming □ Understand the first two OOP principles: <ul style="list-style-type: none"> – Abstraction: data and functional abstraction – Encapsulation: packaging and information hiding □ Appreciate good OOP design <ul style="list-style-type: none"> – Guiding principle: <i>Tell-Don't-Ask</i> – Bottom-up testing to avoid cyclic dependencies □ Appreciate the importance of maintaining an abstraction barrier when writing object-oriented programs </div>	<div data-bbox="1173 817 2161 865" data-label="Section-Header"> <h2>Modeling an Object-Oriented (OO) Solution</h2> </div> <div data-bbox="1173 906 2186 1503" data-label="List-Group"> <ul style="list-style-type: none"> □ Object <ul style="list-style-type: none"> – an abstraction of <i>closely-related data and behaviour</i> □ Both properties and methods of a specific type of object is specified within a class — a blue-print of the object <ul style="list-style-type: none"> – instance property/field/variable: <ul style="list-style-type: none"> ▸ every object has the same set of properties, but possibly different property values – instance method: <ul style="list-style-type: none"> ▸ functionality specific to the object – constructor: <ul style="list-style-type: none"> ▸ a special method to create or <i>instantiate</i> an object </div>
<div data-bbox="1025 1544 1086 1568" data-label="Page-Footer"> <p>2 / 16</p> </div>	<div data-bbox="2145 1544 2206 1568" data-label="Page-Footer"> <p>4 / 16</p> </div>

Point Class

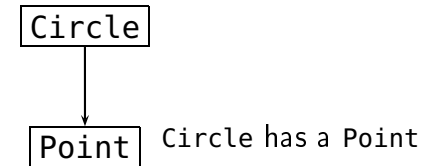
```
class Point {  
    /* properties */  
    final double x;  
    final double y;  
  
    /* constructor */  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    /* method */  
    double distanceTo(Point otherpoint) {  
        double dispX = this.x - otherpoint.x;  
        double dispY = this.y - otherpoint.y;  
        return Math.sqrt(dispX * dispX + dispY * dispY);  
    }  
  
    /* method */  
    public String toString() {  
        return "(" + this.x + ", " + this.y + ")";  
    }  
}
```

5 / 16

Composition: Has-A Relationship

```
class Circle {  
    final Point centre; // Circle has a Point as the centre  
    final double radius; // Circle has a radius  
  
    Circle(Point centre, double radius) {  
        this.centre = centre;  
        this.radius = radius;  
    }  
  
    boolean contains(Point point) {  
        return this.centre.distanceTo(point) < this.radius;  
    }  
  
    public String toString() {  
        return "Circle centered at " + this.centre + " with radius " + this.radius;  
    }  
}
```

jshell> Point p = new Point(1.0, 1.0)
p ==> (1.0, 1.0)
jshell> Circle c = new Circle(new Point(0.0, 0.0), 1.0)
c ==> Circle centered at (0.0, 0.0) with radius 1.0
jshell> c.contains(p)
\$.. ==> false
jshell> c = new Circle(new Point(0.0, 0.0), 2.0)
c ==> Circle centered at (0.0, 0.0) with radius 2.0
jshell> c.contains(p)
\$.. ==> true



7 / 16

Packaging

- Classes provide a way to package
 - lower-level data
 - e.g. data representation of the coordinate values should be packaged within **Point** class
 - lower-level functionality
 - e.g. distance is a computation over two points, hence it should be packaged within the **Point** class
- Exercise: determine if a **Point** is contained within a **Circle**
 - two types of objects: **Point** and **Circle**
 - what are the properties and methods of **Circle**?
 - where should containment be packaged?

6 / 16

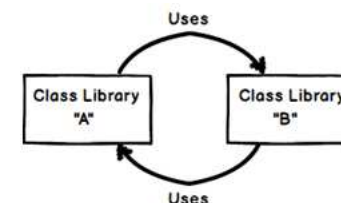
Avoid Cyclic Dependencies

- How about the following alternative design?

```
class Point {  
    final double x;  
    final double y;  
  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    boolean isContainedIn(Circle c) {  
        return c.centre.distanceTo(this) < c.radius;  
    }  
    ...  
}
```

```
class Circle {  
    final Point centre;  
    final double radius;  
  
    Circle(Point centre, double radius) {  
        this.centre = centre;  
        this.radius = radius;  
    }  
    ...  
}
```

- Example: cyclic dependency between two classes:



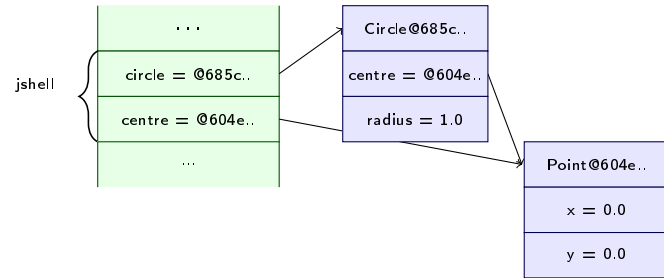
8 / 16

Modeling the Association Between Objects

- Consider modeling the following statements:

```
jshell> Point centre = new Point(0.0, 0.0)
centre ==> (0.0, 0.0)

jshell> Circle circle = new Circle(centre, 1.0)
circle ==> Circle centered at (0.0, 0.0) with radius 1.0
```



- circle references Circle object
- centre in Circle object references a Point object

9 / 16

Encapsulation

- Packaging (discussed earlier) and information hiding
- Consider the method `Circle::contains(Point)` below:

```
boolean contains(Point point) {
    double dx = centre.x - point.x; // properties x and y of Point
    double dy = centre.y - point.y; // class are exposed !!!
    return Math.sqrt(dx * dx + dy * dy) < this.radius;
}
```

- Accessor methods allow for different internal representations

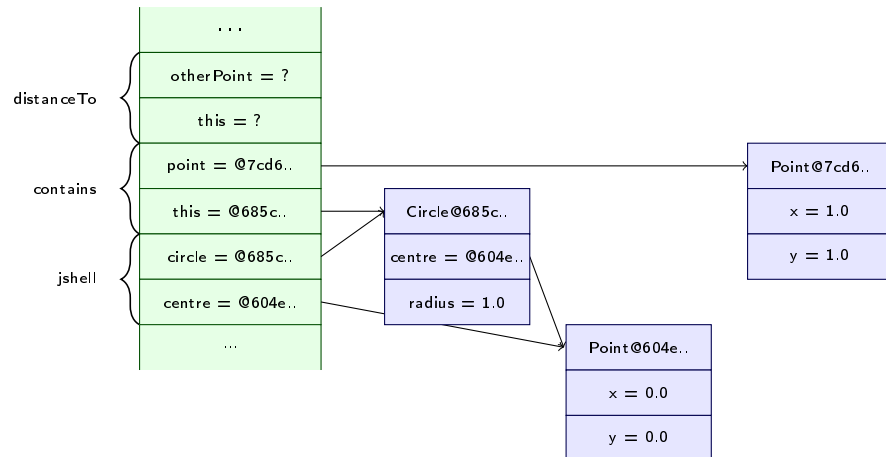
```
class Point {
    final IList<Double> coord;
    Point(double x, double y) {
        this.coord = new IList<Double>()
        .add(x).add(y);
    }
    double getX() { // accessor
        return this.coord.get(0);
    }
    double getY() { // accessor
        return this.coord.get(1);
    }
    ...
}

class Circle {
    private final Point centre;
    private final double radius;
    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }
    boolean contains(Point point) {
        double dx = centre.getX() - point.getX();
        double dy = centre.getY() - point.getY();
        return Math.sqrt(dx * dx + dy * dy) < radius;
    }
}
```

11 / 16

Java Memory Model — **this** reference

```
jshell> circle.contains(new Point(1.0, 1.0)) // contains method calls distanceTo
$.. ==> false
```



10 / 16

private Access Modifier

- Prevent client access to lower level details of the implementer
 - use **private** access modifiers when declaring properties
 - Circle class **must not** access `point.x`
- Guiding principle: **Tell–Don't–Ask**
 - tell an object what to do, *don't ask* an object for data
 - Circle *should not* access `point.getX()`

```
class Point {
    private final IList<Double> coord;
    Point(double x, double y) {
        this.coord = new IList<Double>().add(x).add(y);
    }
    double distanceTo(Point otherpoint) { // tell -- method is exposed to other client classes
        double dispX = this.getX() - otherpoint.getX();
        double dispY = this.getY() - otherpoint.getY();
        return Math.sqrt(dispX * dispX + dispY * dispY);
    }
    private double getX() { // don't ask -- use as a private helper method
        return this.coord.get(0);
    }
    private double getY() { // don't ask -- use as a private helper method
        return this.coord.get(1);
    }
    ...
}
```

12 / 16

Mutating Objects

- Consider `setRadius` as a *mutator* method in `Circle`

```
class Circle {
    private final Point centre;
    private final double radius;

    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    boolean contains(Point point) {
        return this.centre.distanceTo(point) < this.radius; // tell, don't ask
    }

    void setRadius(double newRadius) {
        this.radius = newRadius;
    }

    public String toString() {
        return "Circle centered at " + this.centre + " with radius " + this.radius;
    }
}
```

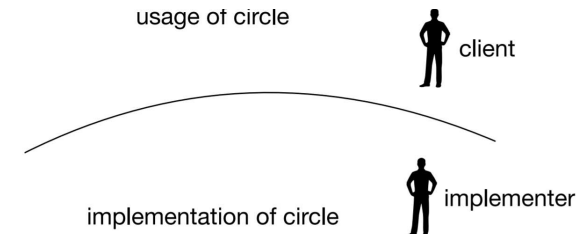
- `Circle` class is uncompileable as `getRadius` attempts to modify the `radius` property which is declared **final**

```
$ javac Circle.java
Circle.java:15: error: cannot assign a value to final variable radius
    this.radius = newRadius;
    ^
1 error
```

13 / 16

Abstraction Barrier

- Provides a separation between the implementation an object, and how it is used by a client across the barrier
 - client calls implementer by *assigning* arguments to method parameters of the implementer
 - implementer returns a value to the client which is then either *assigned* to a variable in the client, or passed to (*assigned* to parameters of) another method



15 / 16

Mutation via Creation of New Objects

- Avoid state-mutating **void** methods; return new object instead

```
class Circle {
    private final Point centre;
    private final double radius;
    ...
    Circle setRadius(double newRadius) {
        return new Circle(this.centre, newRadius);
    }
}

jshell> Circle c = new Circle(new Point(0.0, 0.0), 1.0) // test setup
c ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> Point p = new Point(1.0, 1.0) // test setup
p ==> (1.0, 1.0)

jshell> c.contains(p) // testing the contains method
$.. ==> false

jshell> c.setRadius(2.0).contains(p) // write test via method chaining
$.. ==> true

jshell> c.contains(p) // immutable object c results in same outcome
$.. ==> false
```

14 / 16

Abstraction Barrier

- Adherence to OOP principles sets up an **abstraction barrier** between the client and implementer
- OOP Principle #1: **Abstraction**
 - Implementor* defines the data/functional abstractions using lower-level data and processes
 - Client* uses the high-level data-type and methods
- OOP Principle #2: **Encapsulation**
 - Package* related data and behaviour in a self-contained unit
 - Hide* information/data from the client and allowing access only through methods provided by the implementer
- Two other OOP principles of inheritance and polymorphism will be discussed in the next lecture...

16 / 16