
CS2030 Lecture 2

Abstraction and Encapsulation

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2022 / 2023

Outline and Learning Outcomes

- Be able to transition from data-process to object-oriented modeling and programming
- Understand the first two OOP principles:
 - **Abstraction**: data and functional abstraction
 - **Encapsulation**: packaging and information hiding
- Appreciate **good OOP design**
 - Guiding principle: *Tell-Don't-Ask*
 - Bottom-up testing to avoid cyclic dependencies
- Appreciate the importance of maintaining an **abstraction barrier** when writing object-oriented programs

Abstraction in Object-Oriented Design

- Consider a point object:
 - data abstraction
 - ▷ e.g. a point comprises two floating-point values
 - **double** x; **double** y; or
 - **double**[] coord; or
 - List<Double> coord; ...
 - functional abstraction
 - ▷ e.g. a point can determine the distance from *itself* to another given point
 - p.distanceTo(q) or q.distanceTo(p),
where p and q are referring to Point objects

Modeling an Object-Oriented (OO) Solution

□ Object

- an abstraction of *closely-related data and behaviour*
- Both **properties** and **methods** of a specific type of object is specified within a **class** — a blue-print of the object
 - instance property/field/variable:
 - ▷ every object has the same set of properties, but possibly different property values
 - instance method:
 - ▷ functionality specific to the object
 - constructor:
 - ▷ a special method to create or *instantiate* an object

Point Class

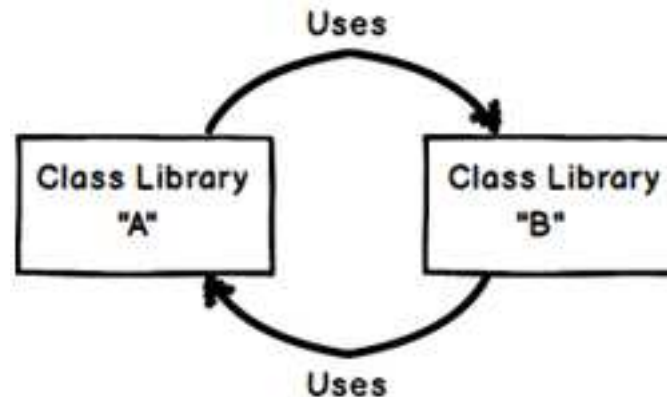
```
class Point {  
    /* properties */  
    final double x;  
    final double y;  
  
    /* constructor */  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    /* method */  
    double distanceTo(Point otherpoint) {  
        double dispX = this.x - otherpoint.x;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

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?

Avoid Cyclic Dependencies

- Example: cyclic dependency between two classes:



- Assuming that Circle defines two properties: Point centre and **double** radius, which design is better?

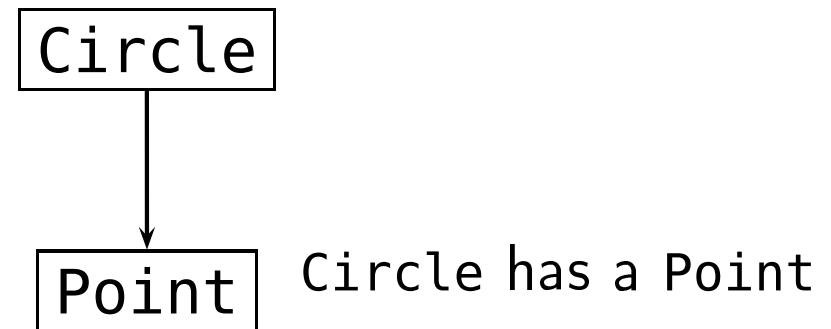
```
class Point {  
    final double x;  
    final double y;  
  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    boolean isContainedIn(Circle c) {  
        ...  
    }  
}
```

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

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
```

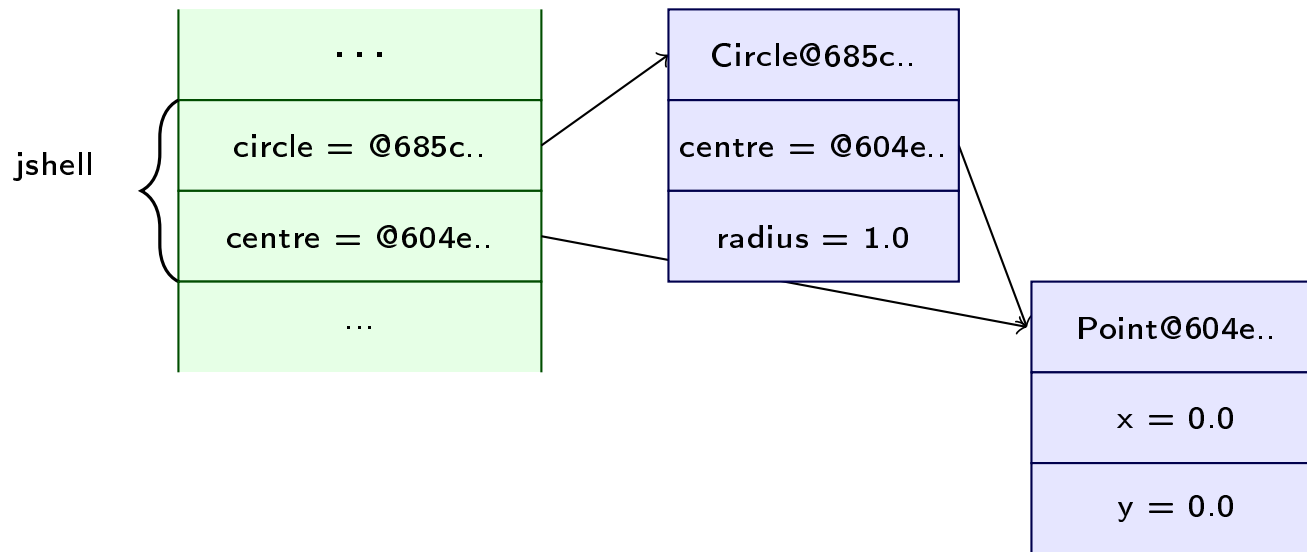


Modeling the Association Between Objects

- Consider modeling the following statements:

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

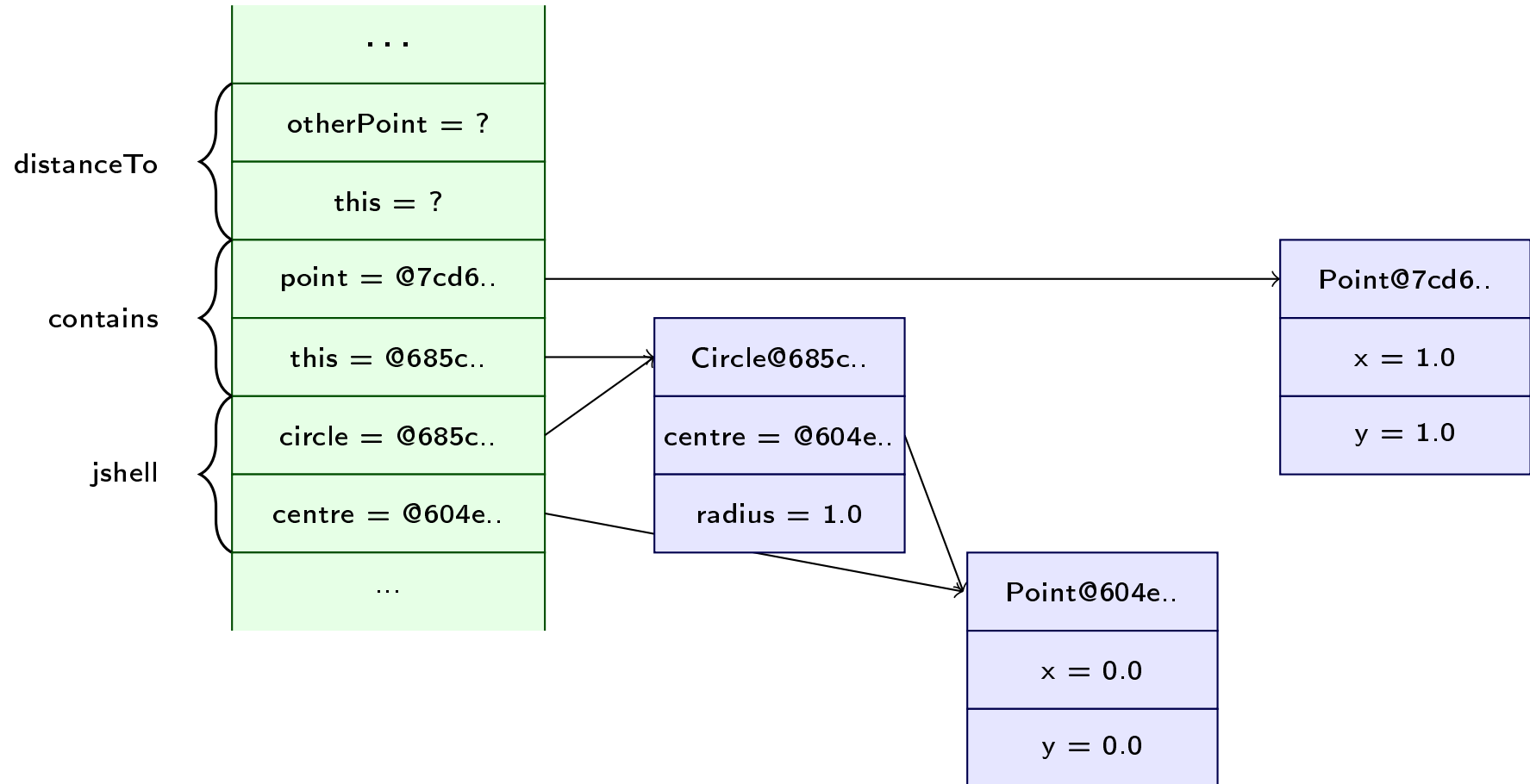
```
jshell> Circle circle = new Circle(centre, 1.0)  
circle ==> Circle centered at (0.0, 0.0) with radius 1.0
```



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

Java Memory Model — **this** reference

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



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

```
import java.util.List;  
  
class Point {  
    final List<Double> coord;  
    Point(double x, double y) {  
        this.coord = List.of(x, 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;  
    }  
}
```

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 List<Double> coord;  
  
    Point(double x, double y) {  
        this.coord = List.of(x, 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);  
    }  
    ...  
}
```

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

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)  
c ==> Circle centered at (0.0, 0.0) with radius 1.0
```

```
jshell> Point p = new Point(1.0, 1.0)  
p ==> (1.0, 1.0)
```

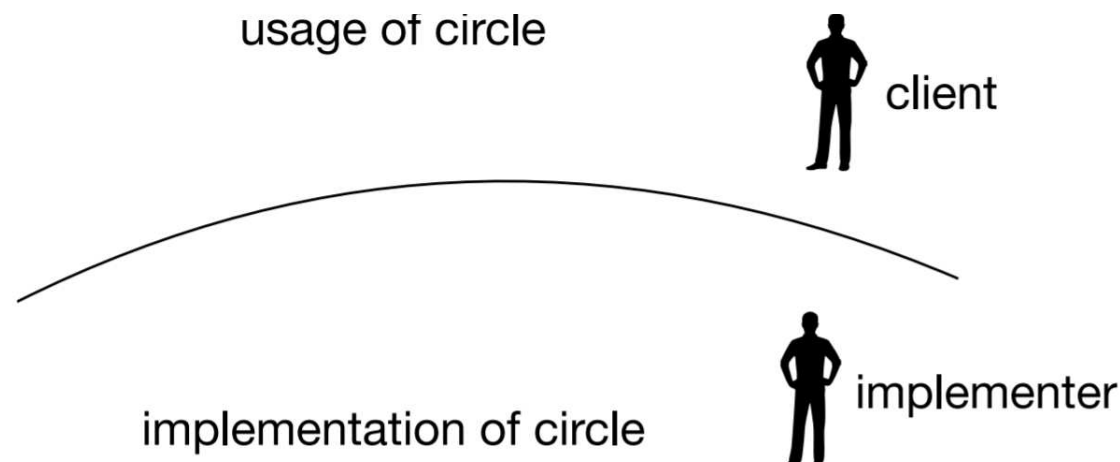
```
jshell> c.contains(p)  
$.. ==> false
```

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

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

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



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...*