CS2030 Lecture 2

Abstraction and Encapsulation

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

Semester 1 2023 / 2024

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** between the client and implementation classes

Abstraction in Object-Oriented Design

- Consider a point as an object:
  - data abstraction
    - e.g. a point comprises two floating-point values
      - `double x; double y;` or
      - `ImList<Double> coord;` or
      - `Pair<Double, Double> pair;` ...
  - 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 */
    double x;
    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 + ")";
    }
}
```

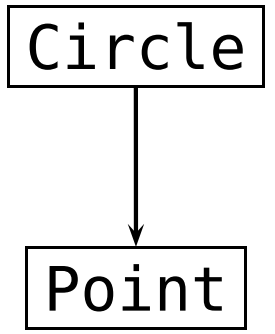
Packaging

- 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?

Has–A Relationship

```
class Circle {
    Point centre; // Circle has a Point as the centre
    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
```



Circle has a Point

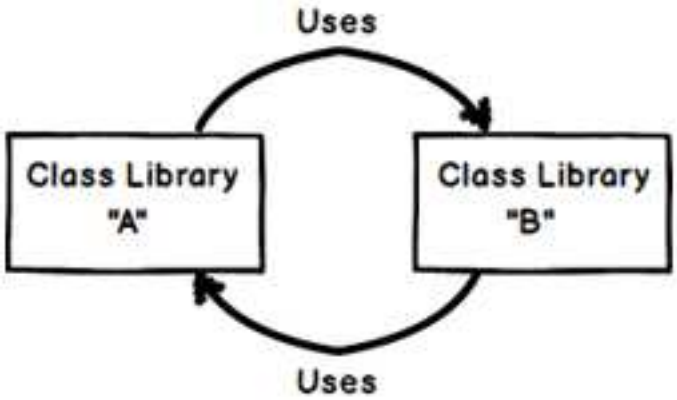
Avoid Cyclic Dependencies

- How about the following alternative design?

```
class Point {
    double x;
    double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    boolean isContainedIn(Circle c) {
        return c.centre.distanceTo(this) < c.radius;
    }
    ...
```

```
jshell> new Point(1.0, 1.0).
   ...> isContainedIn(
   ...>     new Circle(
   ...>         new Point(0.0, 0.0), 2.0))
$.. ==> true
```

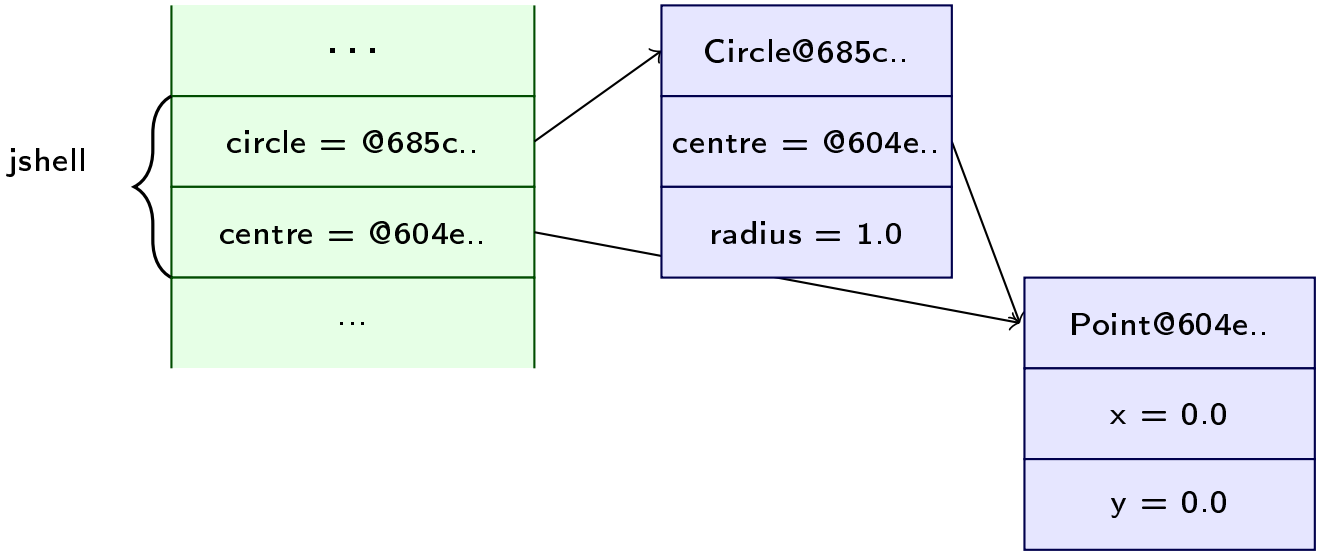
- Avoid cyclic dependencies between classes, e.g.



Modeling the Association Between Objects

- Consider modeling the following statements:

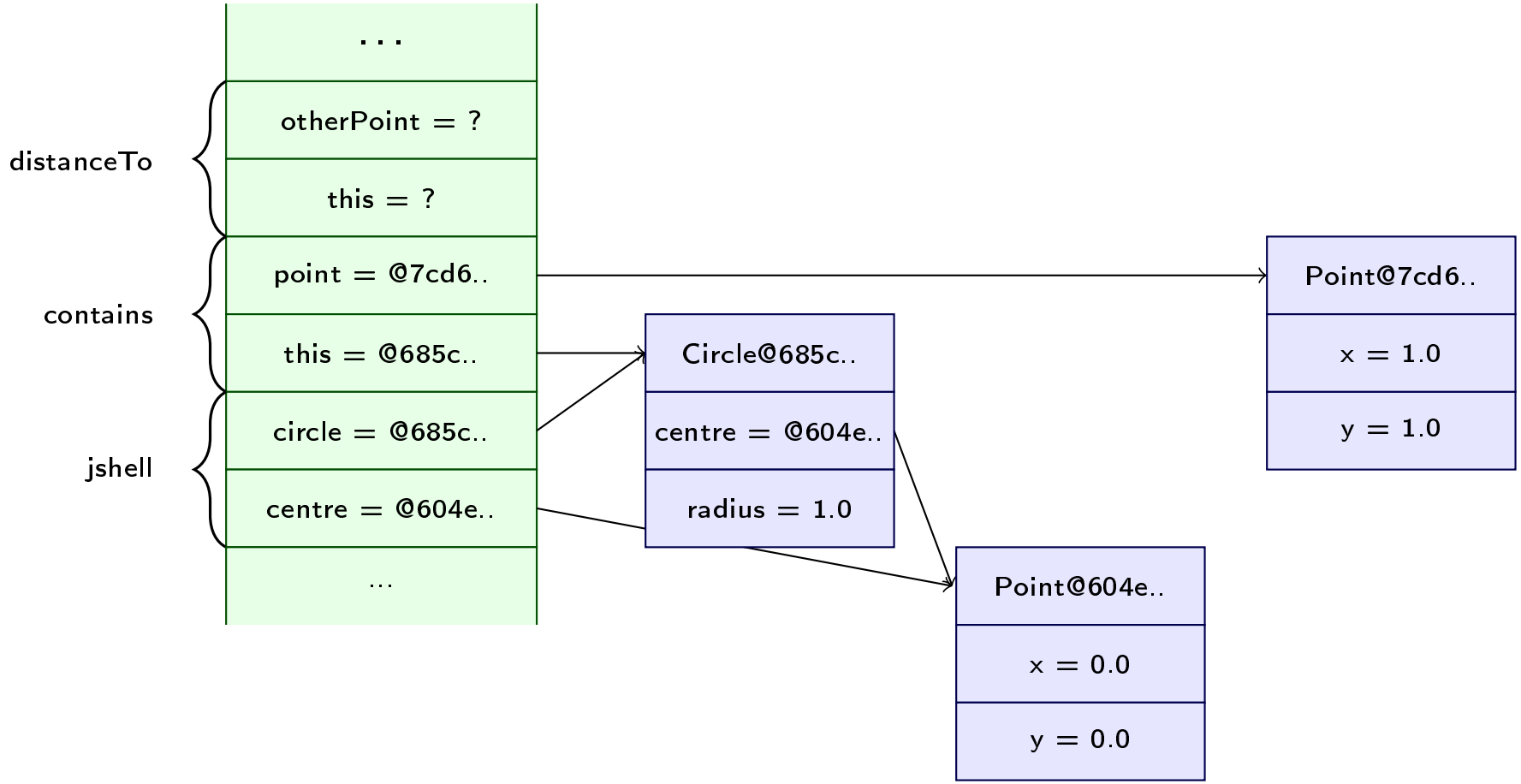
```
jshell> Point centre = new Point(0.0, 0.0)
centre ==> (0.0, 0.0)

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

```
class Point {
    ImList<Double> coord;

    Point(double x, double y) {
        this.coord = new ImList<Double>()
            .add(x).add(y);
    }

    double x() { // accessor
        return this.coord.get(0);
    }

    double y() { // accessor
        return this.coord.get(1);
    }
    ...
```

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

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

    boolean contains(Point point) {
        double dx = centre.x() - point.x();
        double dy = centre.y() - point.y();
        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
  - e.g. client `Circle` **must not** access `point.x`
- Guiding principle: **Tell–Don't–Ask**
  - *tell* an object what to do, *don't ask* an object for data
  - e.g. client `Circle` *should not* access `point.x()`

```
class Point {
    private ImList<Double> coord; // don't allow client direct access

    Point(double x, double y) {
        this.coord = new ImList<Double>().add(x).add(y);
    }

    double distanceTo(Point otherpoint) { // tell -- method is exposed to other client classes
        double dispX = this.x() - otherpoint.x();
        double dispY = this.y() - otherpoint.y();
        return Math.sqrt(dispX * dispX + dispY * dispY);
    }

    private double x() { // don't ask -- use as a private helper method
        return this.coord.get(0);
    }
    ...
```

Mutating Objects

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

```
class Circle {
    private Point centre;
    private 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 scale(double factor) {
        this.radius = this.radius * factor;
    }

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

```
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.scale(2.0)

jshell> c
c ==> Circle centered at (0.0, 0.0) with radius 2.0
```

Mutation via Creation of New Objects

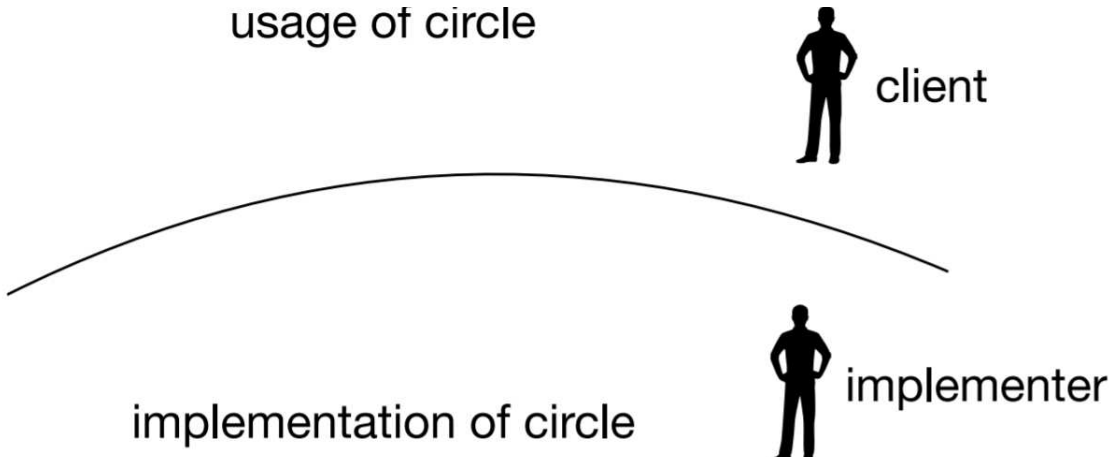
- Make objects immutable by making properties **final**
- Avoid state-mutating **void** methods; return new object instead

```
class Circle {
    private final Point centre;
    private final double radius;
    ...
    Circle scale(double factor) {
        return new Circle(this.centre, this.radius * factor);
    }
```

```
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.scale(2.0).contains(p) // write 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 subsequent lectures...*