
CS2030 Lecture 1

Programming as Communication Across an Abstraction Barrier

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

Semester 2 2019 / 2020

Imperative Programming Concepts

- Imperative — specifies **how** computation proceeds using *statements that change program state*
- Data (Memory)
 - Primitive data-types: numerical, character, boolean
 - Composite data-types:
 - Homogeneous: array (multi-dimensional)
 - Heterogeneous: record (or structure)
- Process (Mechanism)
 - Primitive operations: arithmetic, relational, logical, ...
 - Control structures: sequence, selection, repetition
 - **Modular** programming: functions, procedures, recursion
 - Input and output

Static Typing vs Dynamic Typing

- Dynamic (e.g. JavaScript):

```
var a;  
var b = 5.0;  
var c = "Hello";  
  
b = "This?"; // ok
```
- Static (e.g. Java):

```
int a;  
double b = 5.0;  
String c = "Hello";  
  
b = "This?"; // error
```

- As Java is a type-safe language, it is very strict when it comes to type checking
- Need to develop a sense of “type awareness” by maintaining type-consistency
- Incompatible typing throws off an error

JShell — Interactive Shell for Java

- Uses REPL to provide an immediate feedback loop

```
jshell> double b = 5.0
b ==> 5.0

jshell> b = "This?";
| Error:
| incompatible types: java.lang.String cannot be converted to double
| b = "This?";
|   ^-----^

jshell> var c = "Hello" /* var has been introduced since java 10.. */
c ==> "Hello"

jshell> /var /* .. not to be confused with the /var jshell command */
| double b = 5.0
| String c = "Hello" /var is a jshell command, it does not belong to java

jshell> c = 1
| Error:
| incompatible types: int cannot be converted to java.lang.String
| c = 1
|   ^

jshell> /exit
| Goodbye
```

- We shall be using JShell as a “testing framework” for unit or integrated (incremental) testing throughout the module

Exercise: Point Within a Circle

“Given a 2D *point*, and a *circle* represented by it's *centre* and *radius*, determine if the circle *contains* the point within it”

- What are the data items?
 - point represented by x and y coordinates
 - circle represented by a centre (i.e. a point) and the radius
- Make use of two **double** values to store each point; one **double** value to store the radius
- Modularity: design a function `contains` that
 - takes in the point and circle, and
 - returns **true** if the point lies within the circle, or **false** otherwise

An Imperative (Procedural) Solution

```
jshell> boolean contains(double point_x, double point_y,  
...> double centre_x, double centre_y, double radius) {  
...> double dx = point_x - centre_x;  
...> double dy = point_y - centre_y;  
...> double distance = Math.sqrt(dx * dx + dy * dy);  
...> return distance < radius;  
...> }  
| created method contains(double,double,double,double,double)  
  
jshell> contains(1, 1, 2, 2, 1)  
$.. ==> false  
  
jshell> contains(1, 1, 2, 2, 2)  
$.. ==> true
```

- Notice that the definition of the `contains` function requires
 - knowledge of the point and circle in terms of five **double** variables
 - knowledge of computing the distance between two points, in order to determine containment

Abstraction

- Ideally the function should take in a point and a circle, i.e.
`boolean contains(Point point, Circle circle) {`
 - **Data abstraction:** abstract away lower level data
- The implementation of the function can be as simple as
`return distanceBetween(point, circle.centre) < circle.radius;`
`}`
 - **Functional abstraction:** abstract away lower level computation
- Point and Circle are different types of objects
- Use a **class** to define each individual type of object

Data Abstraction

- Point object has properties of x and y

```
jshell> class Point {  
...> double x;  
...> double y;  
...> }  
| created class Point
```

- Circle object has properties of centre and radius

```
jshell> class Circle {  
...> Point centre;  
...> double radius;  
...> }  
| created class Circle
```

- Creating a point and circle (*though not the desirable way..*)

```
jshell> Point p = new Point()  
p ==> Point@6fc6f14e
```

```
jshell> p.x = 1; p.y = 1  
$.. ==> 1.0  
$.. ==> 1.0
```

```
jshell> Circle c = new Circle()  
c ==> Circle@2286778
```

```
jshell> c.centre = new Point()  
$.. ==> Point@6d7b4f4c
```

```
jshell> c.centre.x = 2; c.centre.y = 2; c.radius = 1  
$.. ==> 2.0  
$.. ==> 2.0  
$.. ==> 1.0
```


Functional Abstraction

□ Include distanceBetween function

```
jshell> double distanceBetween(Point p, Point q) {  
    ...> double dx = p.x - q.x;  
    ...> double dy = p.y - q.y;  
    ...> return Math.sqrt(dx * dx + dy * dy);  
    ...> }  
| created method distanceBetween(Point,Point)  
  
jshell> distanceBetween(c.centre, p)  
$.. ==> 1.4142135623730951
```

□ Redefine contains function

```
jshell> boolean contains(Circle circle, Point p) {  
    ...> return distanceBetween(circle.centre, point) < circle.radius;  
    ...> }  
| created method contains(Circle,Point)  
  
jshell> contains(c, p)  
$.. ==> false  
  
jshell> c.radius = 2  
$.. ==> 2.0  
  
jshell> contains(c, p)  
$.. ==> true
```

Encapsulation: Packaging

- There are two aspects of encapsulation – **packaging** and **information hiding**; let's focus on the first aspect
- Classes provide a way to package low level data
- In addition, low level functionality should also be packaged
- With `Point` and `Circle` objects,
 - where should `distanceBetween` be packaged?
 - ▷ Distance is a computation over two points; it should be packaged within the `Point` class
 - ▷ Let the function be invoked through a `Point` object, i.e. if `p` and `q` are points, then `p.distanceTo(q)` or `q.distanceTo(p)` should give the same result
 - where should `contains` be packaged? *Think dependency..*

Modeling an Object-Oriented (OO) Solution

- Object — an abstraction of *closely-related data and behavior*
- An object-oriented model is a programming solution based on interacting objects:
 - a point has two **double** attributes representing the x- and y-coordinates of the point
 - a circle has a point as it's centre and a radius
 - these are **properties** / **attributes** / **fields** of the object
- To determine if a circle contains a point,
 - a circle takes in a point to check for containment
 - a circle's centre (i.e. a point) takes in another point to get its distance with respect to this other point
 - there are **methods** of the object

Point Class

- The **properties** and **methods** of a specific type of object is specified within a **class** — a blue-print of the object

```
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);  
    }  
}
```

Point Class

□ Properties:

- a Point comprises two **double** values x and y
- every Point object has the same set of properties, but different property values

□ Constructor:

- a method to create or **instantiate** a point
- the Point constructor takes in two **double** values as arguments and assigns them to its properties

□ Method:

- a method `distanceTo` that returns the distance between a given Point and **itself**

Circle Class

```
class Circle {
```

```
jshell> Point p = new Point(1, 1)
p ==> Point@604eb137

jshell> Point centre = new Point(2, 2)
centre ==> Point@7cd62f43

jshell> Circle c = new Circle(centre, 1)
c ==> Circle@5622fdf

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

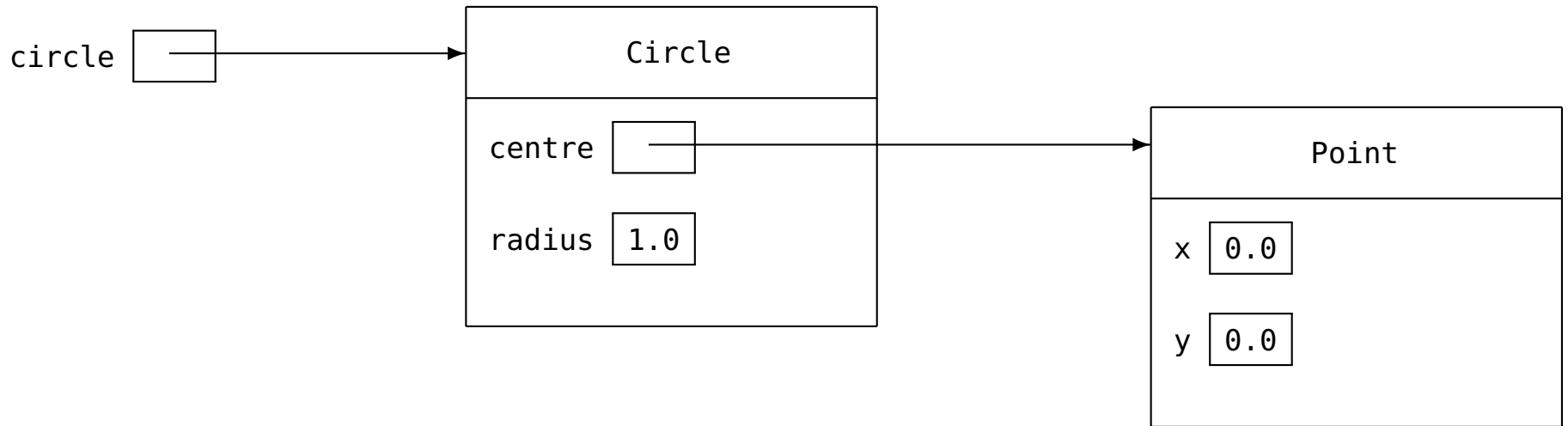
jshell> c.radius = 2
$.. ==> 2.0

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

Modeling the Association between Objects

- Learn to develop a mental model that is **correct**, **consistent** and **complete**
- Consider modeling the following statement:

```
Circle circle = new Circle(new Point(0, 0), 1);
```



- Notice the use of **references** that refers (points) to objects

Reference

- When an object is created using **new**, a **reference** to the instantiated object is returned

- What happens in the following?

```
Point p = new Point(1, 1);  
Point q = p;  
p.x = 2;
```

- How about this?

```
Point p = new Point(1, 1);  
void foo(Point p) {  
    p.x = 3;  
}
```

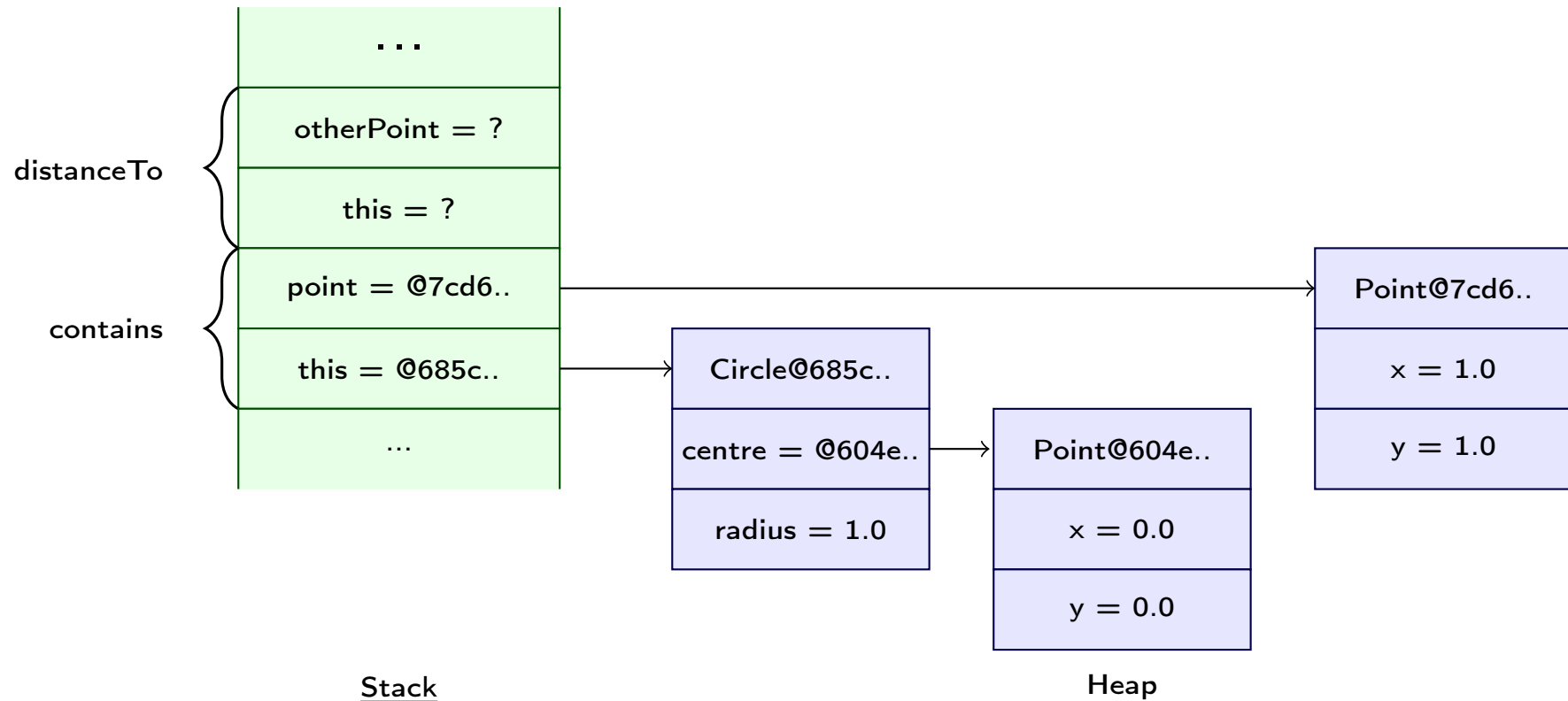
- What if the parameter name in method foo is changed to r?
- References are passed **as a value** to a function

Modeling Object Behavior

- Modeling behaviour requires knowledge on the mechanism of Java method calls and activations
- The Java **memory model** comprises three areas:
 - Stack
 - ▷ LIFO stack for storing activation records of method calls
 - ▷ method local variables are stored here
 - Heap
 - ▷ for storing Java objects upon invoking **new**
 - ▷ *garbage collection* is done here
 - Non-heap (*Metaspace* since Java 8)
 - ▷ for storing loaded classes, and other meta data
 - ▷ *we shall revisit this when we discuss **static** fields*

Java Memory Model

```
jshell> Point centre = new Point(0, 0)
centre ==> Point@604eb137
jshell> Circle circle = new Circle(centre, 1)
circle ==> Circle@685c2f43
jshell> Point point = new Point(1, 1)
point ==> Point@7cd62f43
jshell> circle.contains(point)
$.. ==> false
```



Encapsulation: Information Hiding

- Consider an alternative implementation of the Point class

```
class Point {  
    double[] coord;  
  
    Point(double x, double y) {  
        coord = new double[]{x, y};  
    }  
  
    double distanceTo(Point otherpoint) {  
        double dispX = coord[0] - otherpoint.coord[0];  
        double dispY = coord[1] - otherpoint.coord[1];  
        return Math.sqrt(dispX * dispX + dispY * dispY);  
    }  
}
```

- Not knowing the lower level data of Point, Circle still works!
- However, the following *know-too-much* implementation fails

```
boolean contains(Point point) {  
    double dx = centre.x - point.x;  
    double dy = centre.y - point.y;  
    return Math.sqrt(dx * dx + dy * dy) < radius;  
}
```

Access Modifiers

- Prevent access to lower level details by another object using **private** access modifiers

```
class Point {  
    private double x;  
    private double y;  
  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    double distanceTo(Point otherpoint) {  
        double dispX = this.x - otherpoint.x;  
        double dispY = this.y - otherpoint.y;  
        return Math.sqrt(dispX * dispX + dispY * dispY);  
    }  
}
```

- This prevents the contains method from accessing point.x (point.y) or centre.x (centre.y) directly

Abstraction Barrier

- Interaction between two objects is viewed as communication across an **abstraction barrier**
- Provides a separation between the implementation an object, and how it's used by a client
- OOP Principle #1: **Abstraction**
 - *Implementer 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

Tell–Don't–Ask

- Encapsulation isn't merely about restricting access to properties and providing *getters* (possible even *setters*), e.g.

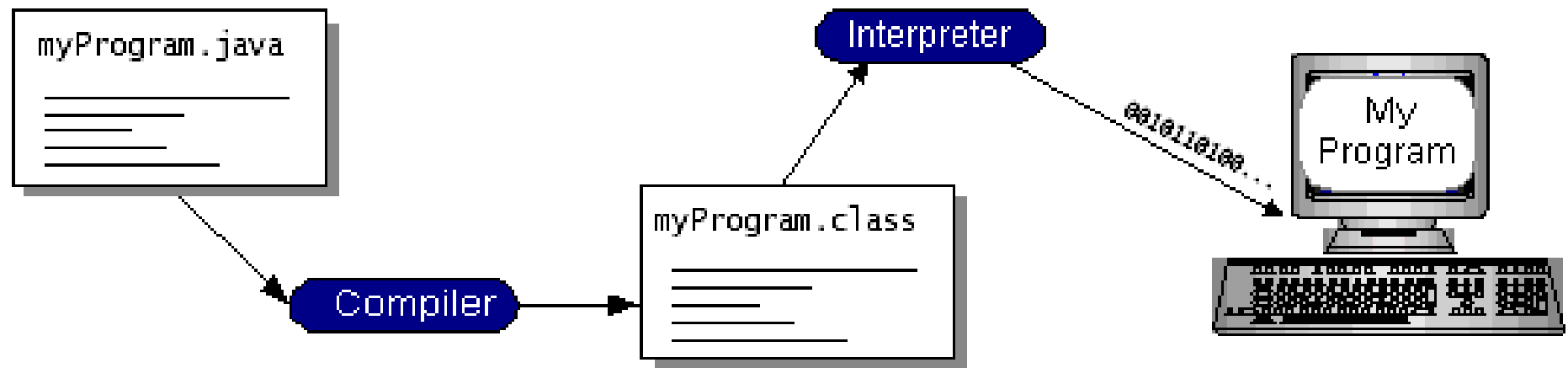
```
class Point {  
    private double x;  
    private double y;  
  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
}
```

```
class Circle {  
    Point centre;  
    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;  
    }  
}
```

- Although implementation details of `Point` is hidden away from `Circle` via *getters*, it violates the **Tell–Don't–Ask** principle
 - Tell an object what to do, rather than asking an object for data and acting on it

Java Compilation and Interpretation

- So far, we have been using the JShell as an interpreter to test our programs
- Java programs are typically compiled to **bytecode** first, and subsequently interpreted by the JVM residing on the machine



- We shall take a closer look at compiling and running Java programs in the next lecture

Lecture Summary

- ❑ Develop a sense of type awareness when writing Java programs
- ❑ Employ object-oriented modeling to convert a process-oriented solution to one that involves the interaction between objects
- ❑ Understand memory management using Java Memory Model
- ❑ Understand the OO principles of abstraction and encapsulation
- ❑ Appreciate the importance of maintaining an abstraction barrier when writing object-oriented programs
- ❑ Appreciate java compilation and interpretation

Difference between **CS2030** and **CS2040**

*While CS2040 trains you to be efficient,
CS2030 trains you to be human.. 😊*