CS2030 Lecture 1

Refresher in Programming

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

Semester 1 2022 / 2023

Outline and Learning Outcomes

- Refresh on basic programming constructs and the *data–process* model of computational problem solving
- Familiarity with *statically-typed* values and variables
- Instill a sense of *type awareness* when developing programs using a strongly-typed language
- Understand the concept of *abstraction*
  - *data abstraction* and *functional abstraction*
- Understand program execution using the Java memory model
- Appreciate the motivation behind effect-free programming
- Appreciate the difference between program *interpretation* (*translation*) and program *compilation*

Data–Process Model

- Data (Memory)
  - Primitive (Singleton): numerical, character, boolean
  - Reference: for composite data
    - Homogeneous: array (multi-dimensional)
    - Heterogeneous: record
- Process (Mechanism)
  - Primitive operations: arithmetic, relational, logical, ...
  - Control structures: sequence, selection, repetition
  - Modularity: value-returning function and procedure
  - Input and output
- Coding Style: `https://www.comp.nus.edu.sg/~cs2030/style/`

Interpreter for Java — JShell

- `JShell` (introduced since Java 9) provides an interactive shell
  - uses REPL to provide an immediate feedback loop

```
$ jshell
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro

jshell> 1 + 1
$1 ==> 2

jshell> /exit
|  Goodbye
```

- Useful for testing language constructs and prototyping
- JShell will be used extensively as a testing framework
  - unit testing
  - integrated (incremental) testing

Assignment with Typed Values and Variables

- Dynamic Typing (e.g. Python):

```
>>> a = 5.0
>>> b = "Hello"
>>> a = b
>>> a
'Hello'
```

- Static Typing (e.g. Java):

```
jshell> double a = 5.0
a ==> 5.0

jshell> String b = "Hello"
b ==> "Hello"

jshell> a = b
|  Error:
|  incompatible types: java.lang.String cannot be converted to double
|  a = b
```

- Java is a type-safe language — strict type checking
- Need to develop a sense of *type awareness*

Recurring Theme — Abstraction Principle

- Abstraction
  - reduces complexity by filtering out unnecessary details
  - develop programs at different levels of abstraction
- Abstraction Principle

  Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts. — *Benjamin C. Pierce*

Exercise: Distance between two points

- A point comprises of two **double** floating point values representing the x and y coordinates
- To compute the Euclidean distance d of (x_1, y_1) and (x_2, y_2)

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
jshell> double x1 = 0.0
x1 ==> 0.0

jshell> double y1 = 0.0
y1 ==> 0.0

jshell> double x2 = 1.0
x2 ==> 1.0

jshell> double y2 = 1.0
y2 ==> 1.0

jshell> double dx = x2 - x1
dx ==> 1.0

jshell> double dy = y2 - y1
dy ==> 1.0

jshell> double distance = Math.sqrt(dx * dx + dy * dy)
distance ==> 1.4142135623730951
```

Data Abstraction

- Create a `Point` *class* as a *user-defined type* comprising two **`double`** values

```
jshell> class Point {
   ...>     double x; // fields/properties
   ...>     double y;
   ...>
   ...>     Point(double x, double y) {  // constructor
   ...>         this.x = x;
   ...>         this.y = y;
   ...>     }
   ...> }
|  created class Point
```

- To create a `Point` object at the origin,

```
jshell> Point origin = new Point(0.0, 0.0)
origin ==> Point@28feb3fa
```

```
jshell> origin.x
$.. ==> 0.0
```

```
jshell> origin.y
$.. ==> 0.0
```

Functional Abstraction

- *Modularity*: define a *generalized* and *cohesive* task
- A *module/function* (or *method* in Java) can take the form of
  - a *function* that returns *exactly one* value; or

```
jshell> double distanceBetween(Point p1, Point p2) {
   ...>     double dx = p2.x - p1.x;
   ...>     double dy = p2.y - p1.y;
   ...>     return Math.sqrt(dx * dx + dy * dy);
   ...> }
|  created method distanceBetween(Point,Point)

jshell> double distance = distanceBetween(origin, new Point(1.0, 1.0))
distance ==> 1.4142135623730951
```

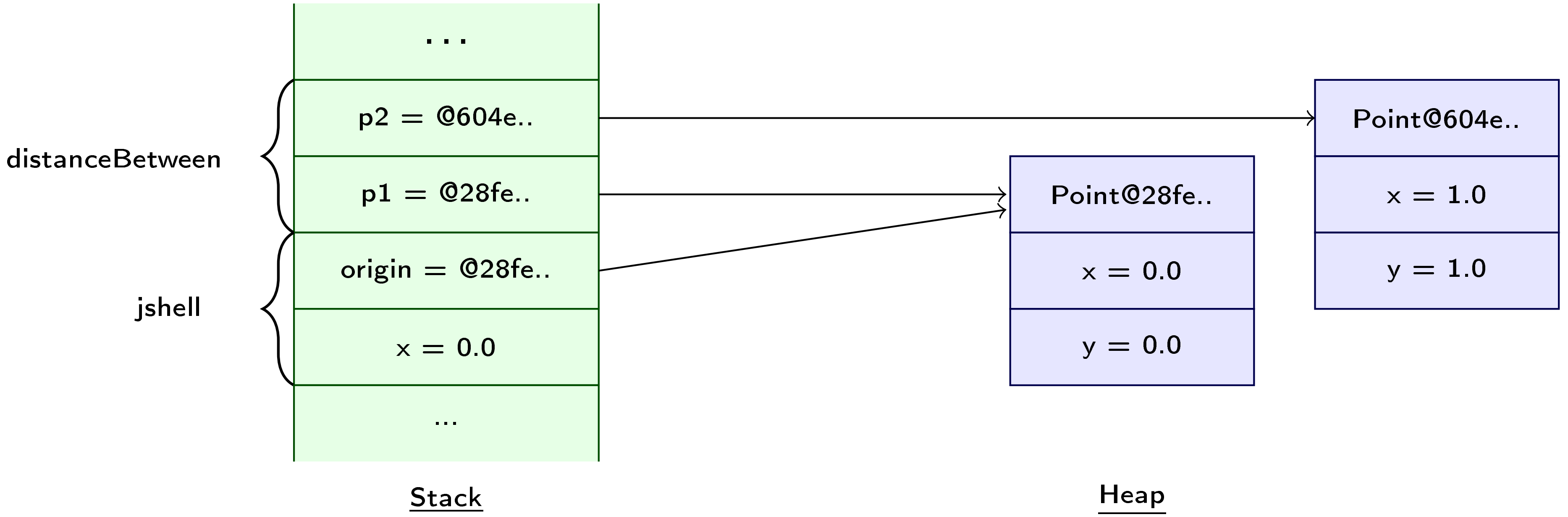
  - a *procedure* that does something but returns nothing (**void**)

```
jshell> void printHello() {
   ...>    System.out.println("Hello");
   ...> }
|  created method printHello()

jshell> printHello()
Hello
```

Java Memory Model

- LIFO *stack* for storing activation records of method calls
- *Heap* for storing Java "objects" created via **new**
- E.g. memory model *just before* `distanceBetween` returns

```
jshell> double x = 0.0
x ==> 0.0

jshell> Point origin = new Point(x, 0.0)
origin ==> Point@28feb3fa

jshell> double distance = distanceBetween(origin, new Point(1.0, 1.0)) // pass-by-(address)-value
distance ==> 1.4142135623730951
```



Immutability and Effect-free Programming

- Declare properties (e.g. `x` and `y` of `Point`) as **`final`**

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

- Prevents the property values being modified once initialized

```
jshell> Point origin = new Point(0.0, 0.0)
origin ==> Point@28feb3fa

jshell> origin.x = 2.0
|  Error:
|  cannot assign a value to final variable x
|  origin.x = 2.0
|  ^------^

jshell> void foo(Point p) {
   ...> p.x = 2.0;
   ...> }
|  Error:
|  cannot assign a value to final variable x
|  p.x = 2.0;
```

  – `p` cannot be modified in `foo`; `foo(origin)` is invalid!

- Facilitates code *maintainability* and *testability*

Composite Data: `List`

- A `List` can be used to *represent an abstraction* of an array — *ordered* collection of *homogeneous* elements
  - need not know how the collection is implemented
- Represent all points using a list of points: `List<Point>`

```
jshell> List<Point> points = List.of(new Point(0.0, 0.0),
   ...> new Point(1.0, 1.0), new Point(2.0, 2.0))
points ==> [Point@28feb3fa, Point@675d3402, Point@51565ec2]

jshell> points.get(1)
$.. ==> Point@675d3402

jshell> points.get(1).x
$.. ==> 1.0
```

- `List` is a *generic* collection – a collection of *any specified* type

```
jshell> List<Integer> ints = List.of(1, 2, 3)
ints ==> [1, 2, 3]
```

Immutable Collection

- `List.of` creates an *immutable collection*
- Write-access is disallowed: `add`, `remove`, `set`, ...

```
jshell> List.of(1, 2, 3).add(4)
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:72)
|        at ImmutableCollections$AbstractImmutableCollection.add (ImmutableCollections.java:76)
|        at (#1:1)

jshell> List.of(1, 2, 3).remove(1)
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:72)
|        at ImmutableCollections$AbstractImmutableList.remove (ImmutableCollections.java:108)
|        at (#2:1)

jshell> List.of(1, 2, 3).set(1, 4)
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:72)
|        at ImmutableCollections$AbstractImmutableList.set (ImmutableCollections.java:110)
|        at (#3:1)
```

- Read-access is allowed: `get`, `size`, `isEmpty`, ...

```
jshell> List.of(1, 2, 3).get(0)
$.. ==> 1

jshell> List.of(1, 2, 3).size()
$.. ==> 3

jshell> List.of().isEmpty()
$.. ==> true
```

Exercise

- Find the maximum distance between any two points from a given list of points, `pts`

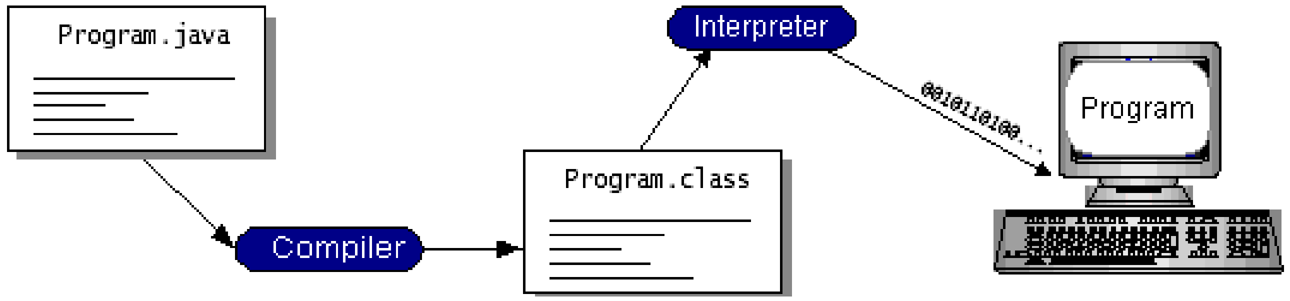
```
jshell> double findMaxDistance(List<Point> pts) {
   ...>    double maxDistance = 0.0;
   ...>
   ...>    for (int i = 0; i < pts.size() - 1; i++) {
   ...>       for (int j = i + 1; j < pts.size(); j++) {
   ...>          double distance = distanceBetween(pts.get(i), pts.get(j));
   ...>          if (distance > maxDistance) {
   ...>             maxDistance = distance;
   ...>          }
   ...>       }
   ...>    }
   ...>    return maxDistance;
   ...> }
|  created method findMaxDistance(List<Point>)

jshell> List<Point> points = List.of(new Point(0.0, 0.0),
   ...> new Point(1.0, 1.0), new Point(2.0, 2.0))
points ==> [Point@28feb3fa, Point@675d3402, Point@51565ec2]

jshell> double maxDistance = findMaxDistance(points)
maxDistance ==> 2.8284271247461903
```

Interpretation vs Compilation

- While JShell *interprets* Java code snippets, Java programs can also be *compiled* and executed via a driver class

```java
import java.util.List; // import List from the library

class Program { // driver class with a main method

    static double distanceBetween(Point p1, Point p2) { // note static modifier
        double dx = p2.x - p1.x;
        double dy = p2.y - p1.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    static double findMaxDistance(List<Point> pts) { // note static modifier
        double maxDistance = 0.0;

        for (int i = 0; i < pts.size() - 1; i++) {
            for (int j = i + 1; j < pts.size(); j++) {
                double distance = distanceBetween(pts.get(i), pts.get(j));
                if (distance > maxDistance) {
                    maxDistance = distance;
                }
            }
        }
        return maxDistance;
    }

    public static void main(String[] args) { // first method to run
        List<Point> pts = List.of(new Point(0.0, 0.0), new Point(1.0, 1.0), new Point(2.0, 2.0));
        double maxDistance = findMaxDistance(pts);
        System.out.println(maxDistance);
    }
}
```

Compiling and Running a Java Program



- To compile (assuming saved in `Program.java`):

  ```
  $ javac Program.java
  ```

  – Syntax errors or incompatible typing throws off a compile-time error

- Bytecode created (`Program.class`) translated and executed/run on the Java Virtual Machine (JVM) using:

  ```
  $ java Program
  2.8284271247461903
  ```