Advanced Object-Oriented Programming

CPT204 – Lecture 1

Erick Purwanto

Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 1

Basic Java Review, Checking 1, Testing 1

Welcome to your first online lecture !

- Welcome to Lecture 1 !
  - It is recommended that you finish Lecture 0 and Lab 0 first

- In this lecture we are going to
  - review some basic concepts you've learned in Intro Java and Data Structure
  - learn the first part about checking
  - learn the first part about testing

- We will continue learning about checking and testing in future lectures

Hailstone Sequence

- As an example in this lecture, we use the hailstone sequence

- Hailstone sequence starts with a number **n**
  - the next number in the sequence is **n/2** if n is even, or **3n+1** if n is odd
  - it ends when it reaches **1**
  - for example:
    - hailstone(5) = [5 16 8 4 2 1]
    - hailstone(3) = [3 10 5 16 8 4 2 1]
    - hailstone(8) =
    - hailstone(2^n) =

- Interestingly, we still don't know if every hailstone sequences always reaches 1

Printing Hailstone

- Here's a Java code to simply print a hailstone sequence

```
int n = 5;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n/2;
    }
    else {
        n = 3*n + 1;
    }
}
System.out.println(n);
```

Types

- In Java, you have to declare the type of a variable before starting using it
  - not in Python

- A **type** is a set of values, along with operations that can be performed on those values
- Java has **8 primitive types**
  - `boolean, byte, char, short, int, long, float and double`
- Java also has **object types**
  - `String, BigInteger`

- By Java convention, primitive types are lowercase, while object types start with a capital letter

Operations

- Operations are functions that take input values and produce output values
- There are three different syntaxes for an operation in Java:
  - As an infix, prefix, or postfix operator
    - e.g., `a + b` is infix operation `+ : int × int → int`
  - As a method of an object
    - e.g., `bigint1.add(bigint2)` calls the operation `add : BigInteger × BigInteger → BigInteger`
  - As a function
    - e.g., `Math.sin(x)` calls the operation `sin : double → double`

- Some operations are **overloaded**, same operation name is used for different types
  - e.g., arithmetic operators +, -, *, / are overloaded for numeric primitive types

Static Typing

- Java is a **statically-typed** language
- The types of all variables have to be known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well
  - For example, if `a` and `b` are declared as `int`s, then the compiler concludes that `a+b` is also an `int`
  - In fact, you will shortly see in our lab, IntelliJ environment does this *while you're still typing* your code !

- In **dynamically-typed** languages like Python or Javascript, this kind of checking is deferred until runtime (while the program is running)

Static Checking

- Static typing is a particular kind of **static checking**, which means checking for bugs at **compile time**
- Static typing prevents a large class of bugs from infecting your program, in particular, bugs caused by applying an operation to the wrong types of arguments
  - for example, if you write a broken line of code like:

    `"5" * "6"`

    that tries to multiply two strings, then static typing will catch this error *while you're still coding*, rather than waiting until the line is reached during execution

- We will explore **checking** again in more detail next week

Review: String

- A string is a series of characters gathered together
  - Create a string by writing its chars out between double quote

    ```
    String hello = "Hello!";
    ```

- Operations on String types include:
  - length `hello.length()`
  - indexing `hello.charAt(0)`
  - concatenation `"a" + "bc"`
  - substring `hello.substring(3)`

    `hello.substring(3, 5)`
  - check presence `hello.contains("lo")`
  - search `hello.indexOf("lo")`
  - equality test `hello.equals("hello!")`

Review: Arrays

- Arrays are fixed-length sequences of another type

- To declare an array variable and construct an array value to assign to it:
  - `int[] a = new int[100];`
  - it includes all possible int array values, but once an array is created, we can never change its length

- Operations on array types include:
  - indexing `a[2]`
  - assignment `a[2] = 0`
  - length `a.length`

Hailstone with Array

- We want to store the sequence in an array, instead of just printing it out

```
int[] a = new int[100];
int i = 0;
int n = 5;
while (n != 1) {
    a[i] = n;
    i++;
    if (n % 2 == 0) {
        n = n/2;
    }
    else {
        n = 3*n + 1;
    }
}
a[i] = n;
```

Review: List and ArrayList

- Instead of a fixed-length array, let's use the List type!

  Lists are variable-length sequences of another type

- To declare a List variable and make a list value:
  - `List<Integer> list = new ArrayList<Integer>();`

- Some of its operations:
  - indexing `list.get(2)`
  - assignment `list.set(2, 5)`
  - add `list.add(5)`
  - length `list.size()`

- Create a list from an array

  `List<Integer> list = Arrays.asList(10, 20, 30)`

Review: List and ArrayList

```
List<Integer> list = new ArrayList<Integer>();
```

- List is an **interface**, a type that can't be constructed directly with `new`, but that instead *specifies the operations* that a List must provide
  - ArrayList is a class, a concrete type that provides *implementations of those operations*
  - ArrayList isn't the only implementation of the List type, though it's the most commonly used one
    - LinkedList is another implementation

Review: List and ArrayList

```
List<Integer> list = new ArrayList<Integer>();
```

- We wrote `List<Integer>` instead of `List<int>`
  Lists only know how to deal with *object types*, not *primitive types*
  - In Java, each of the primitive types (lowercase, abbreviated) has an equivalent object type (capitalized, fully spelled out)
  - Java requires us to use these object type equivalents when we **parameterize** a type with <angle brackets>

Hailstone with List

- Here is the hailstone code written with Lists:

```
List<Integer> list = new ArrayList<Integer>();
int n = 5;
while (n != 1) {
    list.add(n);
    if (n % 2 == 0) {
        n = n/2;
    }
    else {
        n = 3*n + 1;
    }
}
list.add(n);
```

Iterate through List

- You can use the enhanced for loops to iterate through a list
  - for example:

```
// print elements of a hailstone sequence stored in list
for (int x : list) {
        System.out.print(x + " ");
}
```

- As a simple exercise, write a code to find the maximum element in a list of integers using the enhanced for loop !

Methods

- In Java, statements are inside a method, and every methods has to be in a class; so, the simplest code of our hailstone program is:

```
public class Hailstone {
    /**
     * Compute a hailstone sequence.
     * For example, hailstone(5) = [5 16 8 4 2 1].
     * @param n starting number for sequence. Assumes n > 0.
     * @return hailstone sequence starting at n and ending with 1.
     */
    public static List<Integer> hailstone(int n) {
        List<Integer> list = new ArrayList<Integer>();
        while (n != 1) {
            list.add(n);
            if (n % 2 == 0) {
                n = n/2;
            } else {
                n = 3*n + 1;
            }
        }
        list.add(n);
        return list;
    }
}
```

Testing

- Now suppose you have written a code like the one in the previous slide
- How do you know that you have written a **correct** code?
  - You write a test code!
- You may add a main method containing a test code to print the **actual** result against the **expected** output like the following:

```
public static void main(String[] args) {
    int n = 5;
    List<Integer> list = hailstone(n);
    System.out.println("Expected: 5 16 8 4 2 1");
    System.out.print("Actual:   ");
    for (int x : list) {
         System.out.print(x + " ");
    }
}
```

Test with JUnit

- Automate your tests using JUnit test

```
import java.util.Arrays;
import java.util.List;
import org.junit.Test;
import static org.junit.Assert.*;

public class HailstoneTest {
    @Test
    public void testHailstone() {
        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
        List<Integer> actual = Hailstone.hailstone(5);
        assertEquals(expected, actual);
    }

    @Test
    public void testMaxHailstone() {
        int expectedMax = 16;
        assertEquals(expectedMax, Hailstone.maxHailstone(5));
    }
}
```

Multiple assertEquals

- You can create many test methods, and many `assertEquals` inside a test method

```
import java.util.Arrays;
import java.util.List;
import org.junit.Test;
import static org.junit.Assert.*;

public class HailstoneTest {
    @Test
    public void testHailstone() {
        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
        List<Integer> actual = Hailstone.hailstone(5);
        assertEquals(expected, actual);

        expected = Arrays.asList(3, 10, 5, 16, 8, 4, 2, 1);
        actual = Hailstone.hailstone(3);
        assertEquals(expected, actual);
    }
}
```

- We will explore testing more in future lectures!

Thank you for your attention !

- In this lecture, you have learned:
  - Review 1
    - Types, Operators, Arrays, Lists
  - Checking 1
    - Static Typing, Static Checking
  - Testing 1
    - Expected vs Actual, JUnit Testing, assertEquals

- Please continue to Lab 1,
  - to practice testing with JUnit,
  - to do Lab Exercise 1.1, and
  - to do Exercise 1.1, 1.2