

INFO1113

Object- Oriented Programming

Notes from 2018 Semester 2

Contents

| | |
|--|-----------|
| General notes about INFO1113 | 4 |
| Week 1 — Java fundamentals | 5 |
| Compiling Java | 5 |
| Java syntax fundamentals | 6 |
| Standard output and printf() | 7 |
| Java identifiers | 7 |
| Scoping | 7 |
| Types and variables | 8 |
| Arithmetic operations | 9 |
| Command line arguments | 10 |
| Scanner and input | 11 |
| Generating random numbers | 11 |
| Week 2A — Loops and static methods | 12 |
| Operator precedence | 12 |
| Loops in Java | 12 |
| Static variables | 13 |
| Static methods | 14 |
| Call stack | 14 |
| Week 2B — Arrays and strings | 15 |
| Arrays | 15 |
| Multi-dimensional arrays | 16 |
| <i>String</i> | 17 |
| Mutating <i>String</i> with <i>StringBuilder</i> | 18 |
| Week 3A — Objects, classes, and UML | 19 |
| Classes and objects | 19 |
| Encapsulation | 20 |
| Unified Modelling Language (UML) | 21 |
| <i>this</i> keyword | 22 |
| toString() method | 22 |

| | |
|--|-----------|
| Weeks 3B, 4A — I/O | 23 |
| Streams | 23 |
| Scanner | 23 |
| PrintWriter | 24 |
| Binary files | 25 |
| Binary I/O with FileInputStream | 25 |
| Bitwise operators | 26 |
| Converting byte arrays to an integer | 27 |
| Week 4A — Memory | 28 |
| Stack | 28 |
| Heap | 28 |
| Garbage collector | 29 |
| Week 4B — Collections | 30 |
| Abstract Data Type (ADT) | 30 |
| ArrayList | 30 |
| LinkedList | 31 |
| Maps and Sets | 32 |
| Checked and unchecked operations | 33 |
| Week 5 — Class inheritance and overloading | 34 |
| Inheritance | 34 |
| Is-a and has-a relationships | 35 |
| Overloading | 36 |
| Overloading in the Java API | 37 |
| Week 6 — Abstract classes, interfaces, overriding, and polymorphism | 38 |
| Abstract classes | 38 |
| Interfaces | 39 |
| Overriding | 40 |
| Polymorphism | 41 |
| Final methods and classes | 41 |
| Weeks 7, 11B, 12A — Generics and iterators | 42 |
| Generics | 42 |
| Wildcards | 44 |
| Iterators | 45 |
| Inner classes | 46 |

Java syntax fundamentals

Must have a **class** that is the **same name as the file**

| | | |
|---------------------|------------------------|------------|
| <code>public</code> | <code>class</code> | Anatomy |
| Access modifier | Used to define a class | Class name |

The class has a **main method**, defined with the line

| | | | | |
|--|---------------------|-------------------|--|------------------------------|
| <code>public</code> | <code>static</code> | <code>void</code> | <code>main</code> | <code>(String[] args)</code> |
| Allows this method to be accessed without instantiating an object | | | Command line arguments of type <code>String</code> . [] defines an array. | |

Standard output

| | | |
|---------------------------------|--|------------------------------------|
| <code>System.out.println</code> | <code>("Hello World!")</code> | <code>;</code> |
| Use this method | String literals are defined using double quotes | All lines must end with semicolons |

When declaring and instantiating variables, the **type must be defined**

```
int integerVar = 1;
```

Multiple variables can be declared in one line

```
int integerVar, integerVar2;
```

To read from standard input

See Scanner and input

Booleans, logical operators, and if statements

| | |
|-----|-------------------------|
| and | <code>&&</code> |
| or | <code> </code> |
| not | <code>!</code> |

In Java, if statements must have a condition of **type boolean**.
Java uses the keywords `if`, `else if`, `else`

Unlike Python, there is **no exponent operator**

Use `Math.pow(base, exponent)`

Conditional / ternary operator

A shorter way to assign a variable using a condition.
`var = condition ? condition_true : condition_false;`

`exit()` method

```
System.exit(0);
```

Class inheritance and overloading

Inheritance

Inheritance allows **reusability** and changes to inherited methods between different types in a hierarchy.

- Attribute and method reusability
- Defining sub-type methods
- Overriding inherited methods
- Type information

```
[public] class ClassName extends SuperClassName
```

- ClassName becomes an **extension** of SuperClassName.
- The super class **does not know about its subclasses**.
- You can only inherit from **1 class**.
- You **cannot** use subclass properties through a superclass binding.

Encapsulation

The **protected** access modifier will make something not accessible to **other classes**, but it is **accessible within inherited classes**. It will inherit any **protected** or **public** methods or attributes.

Constructors

A subclass will refer to a **super class's constructor**.

When the subclass's constructor is called, the super class's constructor is **called first**, and then the code of the subclass's constructor is executed.

Constructors with parameters

If the super class's constructor **has parameters**, the subclass **must invoke the super constructor** and specify those arguments:

```
super("", 0, 0, 0);
```

We could also match the constructor of the parent type:

```
public Bottle(String name, double width, double height, double depth) {}
public GlassBottle(String name, double width, double height, double depth)
{}

```

Reference types

All reference types inherit the Object class.

Overloading

In Java, overloading is reusing the **same method name** with a **different method signature**.

This is invalid if the **return type** is different but the argument types are the same, since it is **not part of the method signature**.

```
int[] crossProduct(int[] a, int[] b) {}  
int[] crossProduct(float[] a, float[] b) {}
```

When the compiler is unable to determine exactly what method is being called, it will throw an **error**. e.g. Passing `null` as a parameter.

```
int[] x = crossProduct(null, null);
```

By **casting** the reference to a certain type, the compiler can deduce which method to call.

```
int[] x = crossProduct((int[])null, (int[])null);
```

Overloading constructors

Constructors can also be overloaded.

The `this` keyword allows us to refer to the constructor within the context of a constructor.

```
public Person() {  
    this("Jeff", DEFAULT_AGE);  
}  
  
public Person(String name) {  
    this(name, DEFAULT_AGE);  
}  
  
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

The first 2 constructors call the last constructor.

Calling the super constructor

Suppose `Employee` is a subclass of `Person` from above. The `super` keyword calls the super class's constructor.

```
public Employee(String name, int age, long departmentId, long employeeId) {  
    super(name, age);  
    this.departmentId = departmentId;  
    this.employeeId = employeeId;  
}
```

Iterators

An iterator is an object that allows reading through a collection. It maintains state within the collection and where to go next.

How `for`-each loops use iterators

```
ArrayList<String> list = new ArrayList<String>();
for(String s : list) {
    System.out.println(s);
}
```

The `for`-each loop did not exist prior to Java 5. However, iterators did. There exists a pattern that `for`-each loops translate into, using iterators:

```
Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String s = iterator.next(); // Returns element and moves it
    System.out.println(s);
}
```

Implementing iterators

- Collection** implements `Iterable<T>`
- The collection must **implement** the `Iterable` interface:
`public class MyCollection<T> implements Iterable<T>`
 - The collection must implement the abstract method `iterator()` (from `Iterable`):
`public Iterator<T> iterator()`
 - This method must **return an iterator** (that you create):
`return new LinkedListIterator<T>(head);`

- Iterator** implements `Iterator<T>`
- The iterator must **implement** the `Iterator` interface:
`class LinkedListIterator<T> implements Iterator<T> {`
 - The iterator must implement the abstract methods
`public boolean hasNext()
public T next()`