INFO1113

Object-Oriented Programming

Notes from 2018 Semester 2

Contents

Java syntax fundamentals

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

`public`	`class`	`Anatomy`
Access modifier	Used to define a class	Class name

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

`public`	`static`	`void`	`main`	`(String[] args)`
	Allows this method to be accessed **without instantiating** an object			Command line arguments of type `String`. `[]` defines an array.

Standard output

`System.out.println`	`("Hello World!")`	`;`
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	`&&`
or	`||`
not	`!`

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);
```

WEEK 5 —

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()
  ```