Lecture 4 Recap

- Finished introducing main OOP principles and how they are instantiated in Java
  - Modularity
  - Inheritance
  - Abstraction
  - Polymorphism
Lecture 5

- Focus on specific Java features that we feel you should know
  - Parametrised Types & Generics
  - Enumerations
  - Exceptions & Testing
  - Java Collections
  - Cloning

- Last topic before we declare you a Java Expert, and move on to datastructures
Recall our Date class

- field month should really only contain valid months, not arbitrary integers or strings
- how do we limit the type of inputs that can be passed in?

```java
class Date {
    String month;
    int day;
    int year;
}
```
Most OOP languages address this problem with **enums**

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.

**Enums are most often used in switch statements**, where different actions are taken for each value of the enum.

```java
enum Month {
    JANUARY,
    FEBRUARY,
    MARCH, ...
}
Month month = ...;
switch(month):
    case JANUARY:
    case FEBUARY:
```
Generics (very briefly)

- It occasionally makes sense for a class to store groups of fields of different types
  - ex: a 2D vector stores x and y coordinates. x/y could be integers, doubles, floats, etc.
  - But operations on them would remain the same

- Generics allow us to parametrise a class by a specific type. Avoid having to rewrite multiple classes

- More in Assignment 3....
It occasionally makes sense for a class to store groups of fields of different types
- ex: a 2D vector stores x and y coordinates. x/y could be integers, doubles, floats, etc.
- But operations on them would remain the same

Generics allow us to parametrise a class by a specific type. Avoid having to rewrite multiple classes

More in Assignment 3....
So far, we’ve assumed that every input was correct, and that we wrote perfect code (ahem)

- this is rarely true!

Three types of errors

- **syntactic error** -> code doesn’t compile, etc. Usually fairly easy to spot
- **logical errors** (ie: bugs) -> your function doesn’t do what you thought it does
- **external errors** -> external inputs that we can’t control
Minimising Bugs - Unit Testing

- Modular and systematic testing of every method/constructor
  - Yes, it’s boring and may sometimes seem stupid, but necessary
  - OOP encourages you to make each class independent of others, so should also be possible to test independently of others.

- Java provides two main tools to do that
  - assert statement -> program will crash if assertion is false
    - assert(x==5);
  - JUnit, a framework for writing repeatable test
    - Tutorial in the Java Hypertext
Dealing with errors

- **Definition** Errors are expected exceptional behaviour (ex: the file is corrupted or does not exist)

- Three main ways
  - via return codes
  - via deferred error-handling
  - via exceptions

- Different languages prefer different styles
  - Ex: C prefers return codes, Java Exceptions
Error Codes

- Traditional way of handling errors is to return value that indicates outcome of function
- 0 for success, 1 for failure for reason X, 2 for failure for reason Y,, etc.

```java
int setValueArray(int index, int[] array, int value) {
    if (index >= array.length) return -1
    else {
        array[index] = value;
        return 0;
    }
}
```
Problems

- Have to keep checking what the return values are meant to signify
- The actual result can’t actually be returned in the return type
- Can ignore the return value

How would you implement a `getValueAtIndex` function with error codes?
Set some state in the system that needs to be checked explicitly for errors

C has a field “errno” that is used by certain functions to store the error code of the function

Allows the function to return a value

Still requires checking `errno` everytime

```c
int main () {
    FILE * fp;
    fp = fopen ("filedoesnotexist.txt", "rb");
    if (errno == 0) {
        fprintf(fp, …):
    } else {
        fprintf(stderr, "Value of errno: %d\n", errno);
    }
    ...
}
```
Exceptions

- Java privileges **exceptions**

- **Definition**: an exception is an object that can be **thrown** by a method when an error occurs. The exception is **caught** by a handler.
Exceptions

- Java privileges **exceptions**

- **Definition**: an exception is an **object** that can be **thrown** by a method when an error occurs. The exception is **caught** by a handler.

- Java exceptions have thrown with the following syntax:
  - Methods that can throw an exception must be marked as such
    - ```java
    void setDay(int day) throws Exception {
        if (day < 0 || day > 31) throw new Exception("Illegal Day");
        else this.day = day;
        System.out.println("Day set");
    }
    ```

Why not use an assert here?

Throwing an exception is an exit point of the function. Last line not printed if day > 31
Exceptions are **handled** in a **try/catch** block.

Exception is **raised** in a **try** block and **caught** in a catch block.

- Catch block is referred to as a **handler**

Can be multiple **handlers** for a given **try** block, for different exception types.

```java
try {
    setDay(day);
}
catch (IllegalDayException e) {
    // NEVER LEAVE THIS BLANK
}
```

```java
catch (IllegalYearException e) {
    ...
}
```
Exceptions break the control flow of the program as prevent code that follows the exception to be executed.

If need to write cleanup code regardless of whether the exception is thrown, wrap it in a **finally** block.

```java
try {
    FileReader file = new FileReader(fileName);
    file.read();
    file.close();
}
finally {
    if (file!=null) file.close();
}
```
Checked exceptions

- Java supports two types of exceptions: **checked** and **unchecked**

- Checked exceptions must be handled
  - Must be specified in the method signature (```throws```)
  - Code won’t compile unless provide a **handler** for each thrown exception
  - Usually used for “expected” errors (ex: file does not exist, IO bug, etc.)

- Unchecked exceptions do not need to be handled. Arise at runtime. Will crash program. Arise because of programming bugs usually
  - Ex: NullPointerException.
Exceptions are classes too!

- Exceptions are defined as regular classes in Java
- Possible to define specialised exceptions
- All exceptions must extend the `Exception` class
  - Q: why is `Exception` not an interface?

```java
class MyNewException extends Exception {
    int illegalValue;
    MyNewException(int value, String msg) {
        super(msg);
        this非法Value = value;
    }
}
```
Java Class Library

- Java isn’t just a language, it’s a platform with thousands of classes/interface with
  - Data Structures
  - Networking/Files
  - GUI/Multimedia/playback (!)
  - Security
  - Image Processing
  - Concurrency

- You should get into the habit of searching the javadoc to find the appropriate package
Collections

- Will take brief look at collections
  - Very useful to use in most programs
  - Demonstrate the use of interfaces

- **Definition**: grouping of objects that can be iterated over

- Collections implement two main interfaces: *Iterable*, and *Collection*
  - (look it up!)
  - All collections therefore support a common set of operations
Collections

- If all support the same operations, why have more than one collection?
  - Collections implement multiple algorithm, that have different performance characteristics (we’ll see later in class)
  - STL containers in C++ is closest equivalent

- 4 main ones:
  - sets/lists/queues/maps

- Lookup the API!
  - https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
Sets (implements Set<E>)

- A collection of elements without duplicates
- Implementing classes:
  - TreeSet
    - Objects are sorted in order
    - Fast to retrieve contiguously sorted items
  - HashSet:
    - Objects are unordered
    - Supports fast addition/retrieval of single elements
An ordered collection of elements (may contain duplicates)

Linked List: linked lists of elements, store pointer to the head and the tail of the list. Can grow dynamically

ArrayList: array of elements. Efficient to access, but costly to grow. Costly to insert elements in the middle of the list.

```java
LinkedList<Integer> l = new LinkedList<Integer>();
l.add(1);
l.get(); // return 1
```

Lists (implements List<E>)

- An ordered collection of elements (may contain duplicates)
- Linked List: linked lists of elements, store pointer to the head and the tail of the list. Can grow dynamically
- ArrayList: array of elements. Efficient to access, but costly to grow. Costly to insert elements in the middle of the list.

```java
LinkedList<Integer> l = new LinkedList<Integer>();
l.add(1);
l.get(); // return 1
```
Queue (implements Queue<E>)

- Ordered collection of elements (may contain duplicates) that supports removal from head only
- LinkedList: yes, it’s both a list and a queue!
- PriorityQueue: sort elements according to priority so that higher priority elements are at head of queue

```java
LinkedList<Integer> l = new LinkedList<Integer>();
l.offer(1);
l.poll(); // return 1
...```
Maps (implements Map<K,V>)

- Maps keys to values
- Keys must be unique but values can be duplicated or null
- Think of dictionaries in Python (for ex) or Matlab
  - TreeMap: keys kept in order
    - Fast to lookup contiguously sorted items
  - HashMap: keys not sorted in order
    - Fast lookup/insertion of single item
Iterating over a collection

- Through a for loop

```java
Set<Integer> mySet = new TreeSet<Integer>();
for (Integer i: mySet) {
    System.out.println(i);
}
```

- Through an iterator
  - Key benefit of an iterator: safe to remove elements to the collection

```java
Set<Integer> mySet = new TreeSet<Integer>();
Iterator<Integer> it = mySet.iterator();
while (it.hasNext()) {
    Integer i = it.next();
}
while (it.hasNext()) {
    it.remove();
}
```
Manipulating objects - Equality

- Collections requires testing whether objects are equal, or sorting objects.

- It is straightforward to compare primitive types:
  - >, <=, ==, !=, <, <=

- Objects require more care:
  - == on objects tests **reference equality**: checks whether point to same object.

- We would like a way to compare objects whose state is identical.
Recall that every class extends the `Object` class

Object class introduces an `equals()` method

default implementation just does reference equality

To test for value equality, need to **override** the `equals` method
Recall that every class extends the `Object` class.

Object class introduces an `equals()` method.
- default implementation just does reference equality.

To test for value equality, need to override the `equals` method.

```java
class Person {
    private String name;
    private Date dob;
    private String netId;

    @Override
    public boolean equals(Object o) {
        if (o instanceof Person) {
            Person p = (Person) o;
            return p.netId.equals(o.netId);
        } else return false;
    }
}
```
Sometimes equality is not enough: many collections require sorting

Objects that are comparable implement the interface Comparable<T>

Pay attention to the use of an interface! Many objects that have nothing to do with each other can all have the sortable functionality. It would not make sense to use an abstract class in this case.

Comparable Interfaces allow you to define greater than/smaller than/equal functionality

Must implement method int compareTo(T obj)

Returns <0 if smaller, then obj, >0 if greater, 0 if equals
Recall how immutable classes? To make a class immutable, need to copy mutable objects before assigning them to a field.

Java provides a mechanism to do that: the clone() method in the Object class.

Other languages provide what is called a copy constructor.
Distinguish between deep and shallow cloning

- **Shallow cloning:** makes a copy of the object, does not change its fields.
- **Deep cloning:** makes a copy of the object, including all objects that it has as fields (recursively)
Clone() method

- Like equals(), must **override** the **clone** method
- Must also implement the **cloneable** interface
  - But cloneable interface is empty!
  - It is a **marker interface**
    - marker interfaces are empty interfaces used to **label** classes for the compiler
- Clone is quite ugly, unfortunately, copy constructors are problematic too (for inheritance)
You are now Java experts!

- This is almost all the Java that we will teach you in this course
- Will see a few last things in the remainder of class
- Now will begin focusing on datastructures
References in JavaHyperText

enumeration
exception
assert
error handling
collections
comparable
cloning
equals