CS2103 Notes

Week 1

Overload Constructors

```
public Time() {
    this(0, 0, 0);
}
public Time(int hour, int minute, int second) {
    this.hour = hour;
    this.minute = minute;
    this.seconds = second;
}
```

Convert double to int using `int`

```
x = (int)2.25 // x=2
```

Math functions

```
Math.PI // Get value of Pi
Math.pow(x, y) // Raise x to the power of y
Math.max(x, y)
```

Remember getters and setters

Basic Java program template ***

```
public class Main {
    public static void main(String[] args) {
        // ...

        System.out.println(...);
    }
}

class Circle {
    // Attributes
    private int hour;
    private int minute;
    private int second;

    // Class-level attributes
    private static int numOfCircle = 0;

    // Constructors
    public Circle() {
        this.hour = 0;
        // ...
    }
    // Getters
    public int getHour() {
        return hour;
    }
    // Setters
    public void setHour(int hour) {
        this.hour = hour;
    }

}

// Inheritance and child class
class Oval extends Circle {
    public Oval() {
        // Must be in the first line of subclass constructor
        super(); //***
    }
}
```

Enumerations

- Fixed set of values that can be considered as a data type
- Similar to setting levels in a factor in R

- Prevents assignment of invalid values/levels

```
public enum Day {
    // Constants, so uppercase
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

Day today = Day.MONDAY;
Day[] holidays = new Day[]{Day.SATURDAY, Day.SUNDAY};

switch (today) {
    case SATURDAY:
    case SUNDAY:

        System.out.println("It's the weekend");
        break;
    default:
        System.out.println("It's a week day");
}
```

Type Signature

- Type sequence of the parameters of a method

Ways to create an array

1.

```
Animal[] animals = new Animal[]{
    new Cat("Mittens"),
    new Dog("Spot")
};
```

2.

```
Shape[] shapes = new Shape[100];
```

Abstract Classes

- Class cannot be instantiated, but can be subclasses
- E.g. Animal as generalisation of subclasses Cat, Dog, Horse, Tiger
  - Does not make sense to instantiate Animal object
  - move method of Animal class likely abstract because not impossible to implement a move method at Animal class level to fit all subclasses (all

animals move in a different way)

- If a class contains abstract methods, class itself must be abstract

```
public abstract class Animal {
    protected String name;
    public Animal(String name) {
        this.name = name;
    }
    // Note method signature ends with a semicolon, and no method body
    // ***

    public abstract String speak();
}
```

- NOTE: Cannot instantiate abstract classes!
- **WRONG**

```
a = new Animal(); // Compile error
```

- **OKAY**

```
Animal a; // All right to use a type
```

Interface

- Behaviour specification, collection of method specifications
- Is a reference type
- Class implementing an interface is an is-a relationship, like class inheritance

```
public interface DrivableVehicle {
    // Can contain constants and static methods
    int MAX_SPEED = 150;

    static boolean isSpeedAllowed(int speed){
        return speed <= MAX_SPEED;
    }

    // Method signatures have no curly braces, terminated with a semicolon
    void turn(Direction direction);
    void changeLanes(Direction direction);
    void signalTurn(Direction direction, boolean signalOn);
}

// Cannot be instantiated, only implemented
public class CarModelX implements DrivableVehicle {
```

```
    // ...
}
// Can be used a type
DriveableVehicle dv = new CarModelX();
// Interface can inherit other (multiple) interfaces
public interface SelfDrivableVehicle extends DrivableVehicle {
    void goToAutoPilotMode();
}
```

Substitutability

- Every instance of a subclass is an instance of the superclass, but not vice-versa
- An instance of a subclass can be declared as type of superclass

Dynamic Binding

- Mechanism where method calls in code are resolved at runtime, rather than at compile time
- E.g. overriden methods

Static Binding

- Early binding
- When a method call is resolved at compile time
- E.g. overloaded methods, overloaded constructors

Polymorphism

- i. Substitutability
  - Able to treat objects of different types as one type
- ii. Overriding
  - Objects of different subclasses can display different behaviours in response to same method call
- iii. Dynamic Binding
  - Polymorphic code can call method of parent class and yet execute implementation of child class

Collection

- Or "container"
- Object that groups multiple elements into a single unit

- Can store, retrieve, manipulate and communicate aggregate data
- Unified architecture for representing and manipulating collections
- Contains:
    i. Interfaces
        - Abstract data types that represent collections
        - Allow collections to be manipulated independently of the details of their representation
        - e.g. List interface for ArrayList, LinkedList
    ii. Implementations
        - Concerete implementations of collection interfaces
        - Reusable data structures
        - e.g. ArrayList implements List interface
        - e.g. HashMap<K, V> implements Map<K, V> interface
    iii. Algorithms
        - Methods that perform useful computations on objects that implement collection interfaces
        - Polymorphic, same method can be used on many different implementations of the appropriate collection interface
        - e.g. sort(List) can sort a collection that implements List interface

Core Collection interfaces

1. Collection: root of collection hierarchy
2. Set: collection that cannot contain duplicate elements
3. List: ordered collection/sequence
4. Queue: collection used to hold multiple elements prior to processing
5. Map: object that maps keys to values

ArrayList

- Resizable array implementation

```
import java.util.ArrayList;

ArrayList<String> items = new ArrayList<>();

items.add("Apple"); // ["Apple"]
items.contains("Box");
items.get(2);
```

```
items.size();
items.clear();
```

HashMap

- Collection of key-value pairs

```
import java.awt.Point;
import java.util.HashMap;
import java.util.Map;

HashMap<String, Point> points = new HashMap<>();

points.put("x1", new Point(0, 0));
pointAsString(points.get("x1")); //[0,0]
points.containsKey("x1");
points.containsValue(new Point(0, 0));

for (Map.Entry<String, Point> entry : points.entrySet()) {
    print(entry.getKey() + " = " + pointAsString(entry.getValue()));
}
```

where pointAsString is a method defined

Exception Handling

- HANDLE AND RECOVER FROM PROBLEMS (not PREVENT them)
- When error occurs, application may:
    i. request user intervention
    ii. recover on its own
    iii. log user off/shut down system (extreme cases)
- Exceptions are used to deal with 'unusual' but not entirely unexpected situations encountered during run time
- 3 basic categories of exceptions in Java
    i. **Checked exceptions**
        - Application anticipates and recovers from
        - Catch exception and notify user of mistake
    - **Unchecked exceptions**
    ii. Errors
        - Exceptions external to application
        - Application usually cannot anticipate or recover from

        - Indicated by 'Error' and its subclasses
        - e.g. `java.io.IOError`
        - Program might print stack trace and exit
    iii. Runtime exceptions
        - Exceptions internal to application
        - Application usually cannot anticipate or recover from
        - Indicated by ' `RuntimeException` ' and its subclasses
        - e.g. programming bugs, logic errors, improper use of an API
        - e.g. `NullPointerException`

- How exceptions are typically handled
    - When error occurs some point in execution, code being executed creates an exception object
        - Contains info about error (type, state of program)
    - Hands it off to runtime system, i.e. throwing an exception
    - Runtime system tries to find something to handle in call stack
        - Search for method/code that can handle exception, i.e. exception handler
        - Search begines with method in which error occurred and proceeds through call stack in reverse order in which the methods are called
    - When appropriate handler found, runtime system passes the exception to handler
        - Appropriate handler if type of exception object thrown matches the type that can be handled by the handler
        - i.e. catch the exception

- Advantages of exception handling
    - Ability to propogate error info through the call stack
    - Separation of code that deals with 'unusual' situations from code that does the 'usual' work

- `try` - `catch` - `finally` blocks
    - `try` : identifies block of code in which an exception can occur
    - `catch` : identifies block of code (i.e. exception handler) that can handle a particular type of exception
    - `finally` : specify code that is guaranteed to execute with or without the exception

```
public void writeList() {
    print("starting method");
    try {
        print("starting process");
```

```
        print("starting process");
        process();
        print("finishing process"); // will not be printed if exception occurs
in process() - does not execute all of try-block
    // Needs at least one catch/finally-block for try-block
    } catch (IndexOutOfBoundsException e) {
        print("caught IOOBE");

    } catch (IOException e) {
        print("caught IOE");

    } finally {
        // clean up
        print("cleaning up"); // place to close files, recover resources,
clean up after code in try-block
    }
    print("finishing method");
}
```

- Class of exception object indicates type of exception thrown
- Exception object can contain further info about error, including error message
- `Throw` -statement

```
if (size == 0) {
    throw new EmptyStackException();
}
```

- In Java, Checked exceptions have a "Catch or Specify Requirement"
  - Code that might throw checked exceptions must be enclosed in a try-statement (with handler) or method that specifies that it can throw the exception (with throws-clause that lists exception)

```
public void writeList() throws IOException, IndexOutOfBoundsException {
    print("starting method");
    process();
    print("finishing method");
}
```

- Examples *

```
public class IllegalShapeException extends Exception {
//no other code needed
}
```

  - `NumberFormatException`
  - `IndexOutOfBoundsException`

Other notes:

- In Java, a class that does not have any abstract methods can be declared as an abstract class
- Subclass should provide implementations for all abstract methods in superclass or else must be also declared abstract

```
shapesCount++; // Increase count in a method
Integer newValue = Integer.valueOf(roster.get(day).intValue() + 1);
```

Week 1 (Lecture - 16/8)

Main 2 task scenarios in an internship:

1. Greenfield
    - New product
    - Nothing in the field
    - Likely to be a prototype
2. Brownfield
    - Something that is existing
    - Production

For this module, need:

1. Retention: remember things you learn
2. Fluency: use the concepts in a way a native speaker uses the language

Git Fork

1. Fork
2. Clone fork
3. Commit changes
4. Push to fork
5. Can pull directly from remote

Regressions

- Unintended side-effects of fixing a bug
  - e.g. breaking other code
- Automated regression testing of CLI apps
  - Input and expected output files (compare using command ' `FC file1 file2` ')
  - To detect unintended changes