**Programming by Contract**

**Programming by Contract** is a concept which creates a contract between the software developer and the software user.

Every method starts with a **Pre-condition** that must be satisfied by the user of the feature.

* They don’t have to be handled by the developer
* If they hold, error checking is redundant
* They can be specified in Javadoc along with associated parameters @param @pre

Every method ends with a **Post-condition** which the developer guarantees to be true (IFF pre-conditions were met when the method is called).

* The developer guarantees expected behaviour for the end user

**/\*  
 \* Method to deposit money into a bank account  
 \* @param amount is the amount to be deposited  
 \* @pre amount > 0  
 \* @post balance = balance + amount  
 \*/  
Public void deposit(float amount) { code inside method }**

Each class has an **Invariant** which is a logical condition that must be satisfied after any changes are made to the object represented by the class. It guarantees that the object is in a valid state.

* The condition is true after the constructor has completed execution (guarantees no invalid objects are created)
* The condition is true before and after a method call, but it can be temporarily violated during execution of a method.

**/\*   
 \* Bank Account handles the deposit and withdrawal of funds  
 \* @invariant balance >= 0  
 \*/  
Public class BankAccount() { variables and methods inside class }**

Other notes

* Getters don’t really need @pre @post as no state is being changed in the object. The user is simply viewing the object.
* **Calendar.getInstance()** returns a calendar object whose fields have been initialised with the current date and time.
* **currDate.get(Calendar.DATE)** returns the DATE field from the Calendar object.

**Generic Types / Polymorphism**

A **Generic Type** is a class or interface that can be called with different arguments or types.

Use **angle brackets <> to specify the type parameter**.

Difference between standard Array and ArrayList

* Standard Arrays are of fixed length.
  + After initialisation, they cannot expand or shrink.
  + They support primitive types (int, float, double etc.)
* ArrayLists are created with an initial size, however are dynamic.
  + When size exceeds the current max size, the ArrayList size will expand.
  + When an object is removed from the ArrayList, the ArrayList size will shrink.
  + They do NOT support primitive types. Must use (Integer, Float, Double, String etc.) which are actual classes

**List<String> list = new ArrayList<>();**

**list.add(“Apple”);**

**for (String val; list) {**

**String s = val;**

**}**