Imperative Programming Parts 1 and 2 Memorandum

1 Part 1

- Some keywords that are useful when formally proving the correctness of a program:
  - a **precondition** is a condition (assertion, restriction) regarding the parameters of the program;
  - a **postcondition** is a condition regarding what the program will return;
  - an **invariant** is a property that is true at the start and end of each iteration of a loop (is useful in proving that the program is correct);
  - a **variant** is an expression of the program variables that, assuming the invariant, is a natural number, and decreases at each iteration of a loop (is useful in proving that the program terminates);
- **Hoare triples** (notation): the mathematical notation

$$\{P\}\text{Prog}\{Q\}$$

  means that if program Prog is executed from a state that satisfies P, it is guaranteed to terminate in a state that satisfies Q.
- Thus, we can write the correctness of a general program as follows:
  - **Init** establishes **I**, asuming **pre**;
  - **Body** maintains **I**;
  - **I && not test** implies **post**;
  - the loop terminates;

```
// pre
Init
// I
while(test){
  // I && test
  Body
  // I
}
// I && not test
// post
```

- We can also describe the correctness mathematically as follows:
  If
  - $\{pre\}Init\{I\}$;
  - $\{I \wedge test\}Body\{I\}$;
  - $I \wedge \neg test \Rightarrow post$;
  - $I \Rightarrow v \in \mathbb{N}$;
  - $\{I \wedge test \wedge v = V\}Body\{v < V\}$;

  then
  $\{pre\}Init; while(test)Body\{post\}$
- **Unit testing** is testing performed on an individual unit of a program, such as a function.
- **Black box unit testing** treats a component such as function or an object as a "black box":
  - Tests are influenced by knowledge of component specification (what it's meant to do) and interface (what the arguments are).

  - Tests have no knowledge about internal organization of a component (the code/data is hidden from the test).

  Advantages:

  - Stricter focus on specification and the user's point of view.
  - Tests can be truly independent of the developers.
  - Tests can be written as soon as the specification is known—before development work starts.

- **White box unit testing** or "transparent box testing" has knowledge of a component's implementation. A test for an object might also have privileged access to its private data:

  - Tests influenced by knowledge of components internals.
  - Tests may check that invariants are not violated (in objects).

  Advantages:

  - Testing is more thorough, with the possibility of covering many different paths through the code.
  - Exposure of data allows us to check preconditions, invariants etc.
  - Testing can start while the component is still being developed(before there is a graphical user interface).

- One testing technique is to divide all possible inputs into **equivalence classes** in which the behaviour of the component ought to be the same. Test one input from each equivalence class: assuming that because two inputs from the same class should behave the same then they actually will.

- Equivalence classes naturally give us **boundaries**. Dogma says that most code fails on inputs which are near to the boundaries. Testing should be concentrated near to the boundary. Equivalence classes and boundary testing sometimes ignores **internal boundaries** where equivalence classes are further divided by the implementation.

2 Part 2

- Some keywords:

  - **Modularisation**: all functionality and data of an object are held in one place;
  - A **module** collects together data and operations upon that data
  - The **interface** is the interfaces of the operations provided by the module, perhaps linked to an abstract view of the state;
  - The **implementation** is the internal data, plus the implementations of the operations;
  - **Abstraction**: functionality is specified in its general form - free from any specific implementation;
  - **Encapsulation**: implementation details are hidden from the caller;

- **Abstract datatypes** give the generic (idealised) description:

  - **state** refers to what is represented;
  - **init** gives the initial configuration of **state**;
  - **preconditions** and **postconditions** on operations;

- A **trait** (in Scala) gives interface of what a module should do. It defines a list of things to be fulfilled by whatever later uses the trait, not an implementation of them, so can't be used in its own right.

- A **class** which extends a **trait** must give a **concrete implementation** to deliver on those promises.

- A **datatype invariant** is a property that is true initially, and is maintained by each operation, and hence is true after each operation.

- The **abstraction function** is the function abs that takes the concrete state c and gives back the corresponding abstract state a; we write $a = abs(c)$.
- In general, to prove that a concrete implementation meets a specification over the abstract state space, we need to proceed as follows. Suppose that c_0 is an initial concrete state that satisfies the DTI, and suppose $a_0 = abs(c_0)$ satisfies the abstract precondition. Then, the concrete operation must terminate (without error) in a concrete state c and return a result res such that:
  - c satisfies the DTI;
  - the abstract postcondition is satisfied by a_0, $a = abs(c)$ and $abs(res)$