ECS510
Algorithms and Data Structures in an Object Oriented Framework

Summary & Conclusion

Using Objects

- Objects are created using a “constructor” (the word `new` followed by the class name followed by some arguments)
- An object represents an entity in a situation we are modelling
- Variables refer to an object of their type or a subtype
- Variable assignment leads to aliasing

Calling methods on objects

- A method which is not static has to be called “on” an object
- A method call must come from the type or a supertype of the object reference it is called on
- The types of the arguments to a method call must match the types of parameters in the method header

Effect of methods

- A method call may return a value
- A method call may change the state of the object it is called on
- A method call may change the state of objects passed to it as arguments
- A method call may have an external (input/output) effect
- If the state of an object is changed, a method call may not have the same effect as a previous identical method call

Immutable and mutable objects

- An immutable object is one which has no methods which can change its state
- Aliasing means a state change to a mutable object made through a call on one variable is observed through any other variable which aliases it
- A parameter variable in a method aliases an argument variable in a method call, this is why methods can change arguments

Static methods and environments

- A `static` method is not called attached to any object
- It executes in its own environment
- That means it has its own set of variables, they are not the same as variables in other environments even if they have the same name
- But they may alias objects referred to by variables in other environments
- When a method call finishes, execution returns to the previous environment with the return value of the method call replacing the call

Exceptions

- Instead of returning normally, a method call may throw an exception
- An exception can be caught in a `try...catch` statement
- If an exception is not caught, the method where the call was made which threw the exception throws it also
- Some exceptions are “checked”, meaning they either have to be caught or the method which throws them on has to indicate that using `throws`

Defining and throwing your own exceptions

- Exceptions are objects
- You can, and should, define your own exception types, that is how to handle method arguments or other situations which cannot be handled normally
- To define an exception type, extend the class `Exception`
- To throw an exception use `throw` followed by the exception thrown

Arrays

- Arrays are a data structure directly reflecting computer store
- An array type is made by joining `[]` to a type
- An array is an object created by `new` followed by a type name followed by a value of type `int` within `[` and `]`
- Array objects are fixed in length once created
- Individual cells of an array can be accessed through an array variable name followed by a value of type `int` within `[` and `]`

Array notation

- If `arr` refers to an array and `exp` is an expression of type `int`, `arr[exp]` can be treated just like a variable, but which variable it is changes as the value of `exp` changes
- No other structure in Java has special symbols for accessing parts of it
- Arrays are mutable objects and can be aliased, so if `arr1` and `arr2` are two aliases to the same array, `arr1[exp]` and `arr2[exp]` are the same variable

Constructive and destructive operations

- An operation on an array could involve changing the values of an array (although its size can't be changed), this is “destructive” because it destroys the old value of the array
- An alternative approach is to make the operation “constructive” meaning instead of changing the array, a new array with the desired changes is constructed
- A destructive method works by changing the array passed to it as an argument, it does not return it
- A constructive method returns the new array, it should not change its argument array

Aliases and multiply-named variables

- Assigning a variable which refers to one object to refer to a newly created one is not the same as destructively changing the original object because any aliases still refer to the original object
- So if an array or other object is passed as an argument to a method and then the variable in the method header is assigned to something else, the change does not pass to the variable in the method call
- A variable inside an object which is aliased has separate names through the different aliases, but the aliases all share if that variable gets changed
- So if a variable inside an object which is passed to a method gets changed, the change does pass to the object as viewed by the argument to the method

ArrayLists

- Java provides types for collections, `ArrayList` is an example
- Java collections types are generic, `ArrayList` has to be accompanied by a "base type" to make the full type, `ArrayList<Integer>` is an example, but the base type can be any non-primitive type (including types which are themselves generic)
- An `ArrayList` object is manipulated by method calls, not by special notation, the methods `get` and `set` are directly equivalent to array access

Flexible size of ArrayLists

- Unlike arrays, ArrayLists have method calls which increase or decrease their size
- The methods `add` and `remove` work by inserting and deleting items, moving following items up or down one position
- Arrays are created of a fixed size from the start, ArrayLists are created of initial size 0 and then expanded as items are added

Primitive Types

- The numerical types `int`, `double`, `char`, `boolean` and a few others are “primitive types”
- Methods cannot be called on primitive type values. So == has to be used for equality testing
- For all other types, variables store references to objects, so == means “alias testing”
- So for non-primitive types, equality testing should be done using `equals`

Strings

- The type `String` is a class in Java which represents lists of characters
- `String` is not a primitive type
- There are many useful methods in class `String`
- `String` objects are immutable, there are no methods which change them destructively
- There is a special literal notation for `String` objects, for example `"hello world"`
- The `+` operator joins `String` objects

Some other built-in classes in Java

- The class `Scanner` provides input from the command line window, and from files, and can break text into words and numbers
- Objects of class `Scanner` must be created, linked to the command line window or to files, the read methods are called on them
- The “wrapper classes” provide useful static methods for primitive values, and are needed as type arguments to generic types
- The wrapper classes provide object equivalents of primitive values

Lisp Lists

- Lisp lists are not a standard part of Java
- The type `LispList<E>` given to you shows we can use non-standard types provided by someone else
- Objects of type `LispList<E>` are immutable
- The type `LispList<E>` was provided to give a simple introduction to recursion, as many operations on them are best implemented with recursion
- Lisp list objects can also be processed iteratively using a “stack and reverse” approach

Sorting Lisp lists

- Lisp lists enabled some sorting algorithms to be introduced in a simple way
- Insertion sort involves repeatedly inserting items from an unsorted list into the correct position in a sorted list
- Quicksort involves dividing a list into two parts, sorting each part, and joining the sorted parts together
- With quicksort, the division into two parts involves comparison, the joining does not

Sorting indexed collections

- Sorting algorithms with data structures like arrays involve use of indexes
- Sorting a mutable collection could be “in place”, meaning destructive, done by swapping items around, or constructive
- Selection sort is a simple sort in place algorithm, best explained iteratively
- Bubble sort is another iterative destructive sort algorithm

Recursive sort algorithms

- Quicksort can be applied to indexed collections, and can be constructive or destructive
- Quicksort can described iteratively, but is best described recursively
- A recursive description may involve a shared array with arguments giving the portion of the array being considered
- Merge sort is an algorithm similar to quicksort, but the comparison is done when the sorted parts of the initial list are joined

Algorithm efficiency

- The algorithm is the main factor in how fast an operation like sorting a collection is solved, particularly for large amounts of data
- We can analyse the number of “main steps” in using an algorithm, for example the number of comparisons in sorting
- The “big O” notation is an estimate of the number of steps, as a function of the size of the data, considering only the most significant term and ignoring constant multiplying factors
- For some algorithms there is a best case and/or worst case

Implementing Objects

- A class is a “blueprint” for an object
- A class gives the name of variables which every object of that class has
- A class gives constructors which say how object variables are set when new objects are created
- A class gives the code for methods which is executed when a call is made on an object
- Good practice is for the variables in an object to be `private`, so it is seen by other code only in terms of its `public` methods

Instance methods

- A method which is not declared as `static` is an instance method and has to be called on an object
- An instance method works like a static method, but its environment also includes the variables of the object it is called on
- In the code for an instance method, the word `this` is used to mean “the object this method call was made on”

Specification and Implementation

- Code which uses an object views it in terms of how its public methods work
- The public methods of a class should be well defined, with the definitions only given as the effects as a logical relationship with the arguments and what we think the object represents
- The variables and code inside a class should be whatever makes the public methods work correctly according to their definitions
- So there is a clear distinction between application (use of objects of a class) and implementation (writing what goes inside a class)
- The definition or “specification” is the link between application and implementation

Inheritance

- Inheritance enables you to specify one class as a specialised version (a subclass) of another (the superclass)
- A subclass may add extra methods and variables to a superclass
- A subclass may change (override) the behaviour of a method from the superclass
- Code in the subclass is only those aspects added or changed from the superclass

Dynamic binding

- A variable declared of the type of one class may be set to refer to an object constructed from the constructor of one of its subclasses
- In this case, the “apparent type” is the type of the variable, and the “actual type” is the class of the constructor
- The only methods that can be called on a variable are those from its apparent type (including those it inherits from a superclass of that type)
- But if a method is called on a variable which refers to an object of a subclass and the subclass overrides that method, the code used is from the subclass
- This is called “dynamic binding”

The most general type

- Any class which is not declared as extends another class always implicitly extends the class called `Object`
- This means all classes are subclasses, directly or indirectly, of `Object`
- So `Object` is called the "most general type"
- Java provides the class `Object` with some methods in it, for example `equals` and `toString`
- When writing our own classes, we may decide to override the implementations of the methods inherited from class `Object`

Interface Types

- An interface is like a class, but it contains only method headers
- The name of the interface provides a type name
- A class can only extend one superclass, but it can implement any number of interfaces
- When a class implements an interface it must provide code for its methods
- An interface type is the type of all objects which have the methods in that interface and which implement it, it can be used for generalised programming
- Dynamic binding means the code used when a method is called on a variable of an interface type depends on the actual class of the object it refers to

Inheritance in method arguments

- The rule that a variable of one type can refer to an object of a subtype applies to parameters in method headers
- So if a parameter is of an interface type, the matching argument on a method call can be of any class which implements that type
- If a parameter is of a class type, the matching arguments can be of any subclass of that class
- This is one way of writing generalised methods

Casting and `instanceof`

- The rule that methods called on a variable must come from the type of the variable means when writing generalised methods, if a parameter is of a superclass we cannot call methods on it which are only in subclasses of it
- In some cases, we may wish to test if an argument is actually of a subclass, and in that case call a method on it which comes only from that subclass
- In that case, we can use `instanceof` to test if it comes from the subclass, and casting to assign a variable of the subclass to refer to it
- Then we can call the method on that variable
- If a method is in the superclass, no testing or casting is needed, dynamic binding ensures the right code is used

Casting and `instanceof`: syntax

- The test `var instanceof Type` returns `true` if and only if variable `var` refers to an object whose actual type is `Type` or a subclass of `Type`
- The expression `(Type) expr` can be used as a value of type `Type` when `expr` is a value of a supertype of `Type`
- So `a` is a variable of type `Alpha`, and `Beta` is a subtype of `Alpha`, and `bmeth` is a method in type `Beta` but not type `Alpha` then

```
if(a instanceof Beta)
    ((Beta) a).bmeth(…)
```

  calls `bmeth` on `a` if `a` refers to an object of type `Beta`
- A method call has the type of the method's return type, so casting may be used to view the return value of a subtype if it is actually of that subtype

Generic Types

- We can define our own generic types by writing classes and interfaces which have type variables (declared in the class or interface header)
- A type variable may be used as a type in the code, but the only method we can call on an object whose type is a type variable is one from `Object`
- A class may implement a generic interface by providing types for its type variables
- A class may implement a generic interface and use the same type variable, so it is itself generic
- When an object of a generic class is constructed, values for the type variables must be provided

Generic Methods

- A generic method is one which takes an argument of a generic type where it does not need the type argument of the generic type to be set to a particular value
- A generic method has its own type variables declared in its header before its return type
- A generic method can be used with an argument of a generic type with its type variable set to anything, so it is a way of writing generalised code
- The type variable may be used as a type in the method header and code, it is set when the method is called depending on the method's arguments

Bounded type variables

- If we have `Beta` as a subtype of `Alpha`, and `Gen<E>` a generic type, then though an `Alpha` variable may refer to a `Beta` object, a `Gen<Alpha>` variable cannot refer to a `Gen<Beta>` object
- If we want to write generalised code which could take a `Gen<Alpha>` or `Gen<Beta>` argument, we have to give the parameter as `Gen<T>` with `T` a bounded type variable
- A bounded type variable declared as `<T extends Alpha>` could be set to `Alpha` or any subtype of `Alpha`
- Then if a variable is of type `T`, it is possible to call methods from type `Alpha` on it

Generic argument wildcards

- The type variable declaration `<T extends Alpha>` occurs before the return type of the method header it is in. After that `T` is used in the method like any type, the name of the type is not `<T>`, the angled brackets are used when declaring the type variable and when passing it as a type argument, as in `ArrayList<T>`.
- As an alternative, instead of declaring a bounded type variable, we can use the "wildcard" notation, which would mean the type written as `ArrayList<T>` is written `ArrayList<? extends Alpha>`.
- Now there is no name for the base type of the ArrayList object, but elements from it can be referred to by variables of type `Alpha` as it must be `Alpha` or a subtype of `Alpha`.

Inheritance and type variables for generalised programming

- Inheritance using interface types enables us to write and use generalised methods which can take a variety of different types of object when the method deals with individual objects
- Type variables with bounds enable us to write and use generalised methods which can take a variety of different types of object when the method deals with collections of objects
- Inheritance using `extends` on classes enables us to write new classes which specialise existing classes, and methods which take those existing classes still work with objects of the specialised classes.

Design Patterns using interface types

- A “design pattern” is a common way of putting code together, having named patterns helps programmers describe their code to each other, and gives ideas for possible solutions to programming problems
- Many design patterns make use of interface types
- The design pattern “Decorator” has a class implements an interface type and has within it an object of a class which implements the same interface, the idea is that a method call on the Decorator object is implemented by a call to the same method on the object onside it, with some extra work done “decorating” that method call
- The design pattern “Adapter” adapts an object to fit in with an interface

Factory method

- A “factory method” is the design pattern name for a method which works like a constructor, creating new objects
- Unlike a constructor, the return type for a factory method can be an interface type
- The factory method will determine the actual class of the object it returns
- The code which calls the factory method may give it arguments to tell it which actual class object to create, or the factory method could use an actual class the calling code does not know of
- Simple factory methods are static, but we can also have “factory objects” which provide factory methods

Method header

Consist of (in this order):

- Modifiers (`public`/`private`, `static` etc) *optional*
- Type variable declaration *optional*
- Return type
- Method name `(`
- Parameter list `)`
- Exceptions declaration `throws` *optional*

The parameter list consists of type/name pairs

The type variables are enclosed within < and >, separated by `,` if there is more than one type variable

Checked exceptions must be declared if the method can throw them

Class/Interface header

- Can be `class` or `interface` (or `enum`)
- An interface only has instance method headers
- A class has field declarations, methods with code, static methods, nested classes
- A class can extend one other class
- A class can implement any number of interfaces
- An interface can extend any number of interfaces
- Generic classes and interfaces are given by declaring type variables after the class or interface name
- A class can implement a generic interface with the interface's type variables given values

Using types

- The return type of a method tells you where a call to the method can be used
- The class or interface the method header is in tells you on what type of object references it can be called
- The types of the parameters to the method tells you what the types of the arguments to the call can be
- A method call must allow a type variable which occurs more than once in the header to be set consistently
- The bounds on a type variable when it is declared tell you the range of types it can be used for
- Think of types as connectors which help you put code together correctly

Abstract Data Types

- Now we can see that an interface type describes a type just in terms of the methods that can be called on objects which implement that type
- This is known as an “Abstract Data Type”
- A class type gives code which implements the abstract data type, variables which hold a “data structure” and methods which manipulate that structure
- We saw the array data structure and then array-and-count used to implement `ArrayList`
- The type `LispList` can also be implemented with arrays
- But a linked list structure is better for implementing `LispList`

Linked lists

- If a class has a variable in it of its own type, it forms a linked structure, one object of that class links to another and that links to another and so on
- The links don't go on forever because the variable could be set to `null`
- Another possibility is a variable refers back to another object creating a circle of links
- A linked list has objects with just one recursive link

Linked list implementations

- Linked lists can become complex with shared access to objects in the structure
- They are best used as a data structure inside another class to implement well-defined operations
- A linked list closely resembles a Lisp list, so it is a good way of implementing the class `LispList<E>`

Difference between linked list and Lisp list

- Lisp lists can be manipulated only by the methods that can be called on them, and these methods make them immutable
- Linked lists are manipulated by direct access to the variables in the objects linked together
- So linked lists are mutable, and objects in them may be shared
- A Lisp list could be implemented in some other way (such as using an array), we can view it only in terms of the methods that can be called on it
- So Lisp list is an Abstract Data Type, linked list is a data structure

Java's Collection Framework

- Java contains a range of classes and interfaces together called the "Collections Framework"
- The class `ArrayList<E>` is just one aspect of this Collections Framework
- The class `ArrayList<E>` implements an interface `List<E>`
- The class `LinkedList<E>` also implements the interface `List<E>`
- In different circumstances, one or the other implementation is the more efficient

Coding to the interface

- Writing code which uses the interface type `List<E>` means it can be used for `ArrayList<E>` or `LinkedList<E>` objects
- So only use the implementation types when creating new objects
- The type `Collection<E>` is a supertype of `List<E>`
- The type `Set<E>` is another subtype of `Collection<E>`
- The type `List<E>` adds operations to deal with elements being at indexed positions, the type `Set<E>` defines `add` and `remove` operations to give mathematical set behaviour

Iterators

- An object of type `Iterator<E>` can be obtained from any object of a subtype of `Collection<E>`
- It can be used to go through the collection by repeated calls of `next()`
- It goes through `Set<E>` collections even though they don't have indexing
- The class `HashSet<E>` implements `Set<E>` but also `TreeSet<E>` implements `Set<E>`
- With `TreeSet<E>` the `Iterator<E>` object returns the items in sorted order
- With `HashSet<E>` the `Iterator<E>` object returns the items in random order, but access to items is more efficient

Defining order

- The "natural order" of a class is as given by its `compareTo` method
- When we write a class, we can write our own `compareTo` method to go in the class
- We can define a different ordering by providing a `Comparator<E>` object
- We can give a `Comparator<E>` argument to a `TreeSet<E>` constructor to make it order in a different way
- Java has built-in sort code, this can sort by natural order or by an order given by a `Comparator<E>` argument

Interfaces for order

- The type `Comparator<E>` is provided as a standard interface type in Java
- For example,

  ```
  class LengthComparer implements Comparator<String>
  ```

  could be used to sort `String`s by length
- The interface `Comparable<E>` is also a standard interface containing the method header for `compareTo`
- For example,

  ```
  class Date implements Comparable<Date>
  ```

  means `Date` has a `compareTo` method enabling `Date` objects to be compared with each other

Java's built-in sorting

- The various interfaces described form part of Java's Collections Framework
- The class `Collections` in Java contains static methods for various operations such as sorting
- We do not have to write our own sort code, so long as our class of objects *`Thing`* implements

  `Comparable<? super` *`Thing`*`>` or we provide for it an object which implements

  `Comparator<? super` *`Thing`*`>` we can use Java's sort code to sort a collection of objects of class *`Thing`*

Maps

- The interface `Map<K,V>` defines a collection of objects accessible through keys
- There are methods to give the set of keys and collection of objects
- It is implemented by `HashMap<K,V>` and `TreeMap<K,V>`
- A `HashMap<K,V>` is more efficient, but it stores the objects in random order
- A `TreeMap<K,V>` is less efficient, but the objects can be obtained from it in order of key

Views

- A view is a separate object which shares the underlying data of another object
- The `sublist` method on a `List<E>` object returns a view, it is part of the original list, but as it is a view changes to the original list change the sublist and vice versa
- Another view that may be obtained on a collection is an “unmodifiable view”, which shares the underlying data, but does not allow any mutating methods.

Conclusion

- Java's Collections Framework means for the most common programming tasks we don't need to write our own algorithms and data structures, we can use what the code library provides for us
- The code library is written so that it is generalised, the same code can be used in a variety of circumstances
- This means it makes extensive use of generics and interface types
- We had to learn a lot about object oriented programming and the idea of generalising in order to make effective use of the Collections Framework

Algorithms and Data Structures

- In some cases it helped to know the underlying algorithms and data structures to pick the best library code for our needs, and to know what goes on underneath
- In more specialist programming tasks we may need to program our own algorithms and data structures
- This module could only give a very brief introduction to this vast topic

Algorithms

- For large amounts of data, the algorithm used makes a huge difference, it is the biggest factor in efficiency
- We saw how a very simple algorithm like “find the biggest” could be generalised, so it works for any type of data, and for any type of collection, and for any definition of “biggest”
- We looked at some algorithms for perhaps the most common task done with computers, sorting
- We saw that reasoned analysis of these algorithms showed why some were more efficient than others

Data Structures

- The crucial issue here was the difference between seeing how an object performs in terms of its public methods being called by code which uses it, and seeing how it perform in terms of the code in the object's class which implements the methods
- It is possible to change what is inside a class to something completely different so long as the public methods interact with outside code to give the same results as before
- Good code design keeps this separation between application and implementation
- What is inside an object to implement its methods is the "data structure", the description of the object in terms of its public methods only is an "Abstract Data Type"

Coding

- The module gave more practical experience in coding
- This is software development from the bottom up, building small components which are executable which in a real environment would just be part of a much larger system
- Other modules you take emphasise the design of a system from top down
- Computer Science isn't all coding, but familiarity with coding at this level is something every Computer Scientists should have
- There is much more to study for those who have a particular interest in good quality coding!

Java

- Although the programming language used was Java, you should be able to see we were moving from seeing coding just in terms of language syntax and semantics to seeing it more in terms of performing useful tasks
- What we have learnt in this module will transfer to other common programming languages
- Expert programming in Java involves knowing a lot of the common APIs, but that wasn't the emphasis in this module
- The Collection Framework is perhaps the most basic and abstract set of APIs
- There is much more to study for those who have a particular interest in practical Java programming!