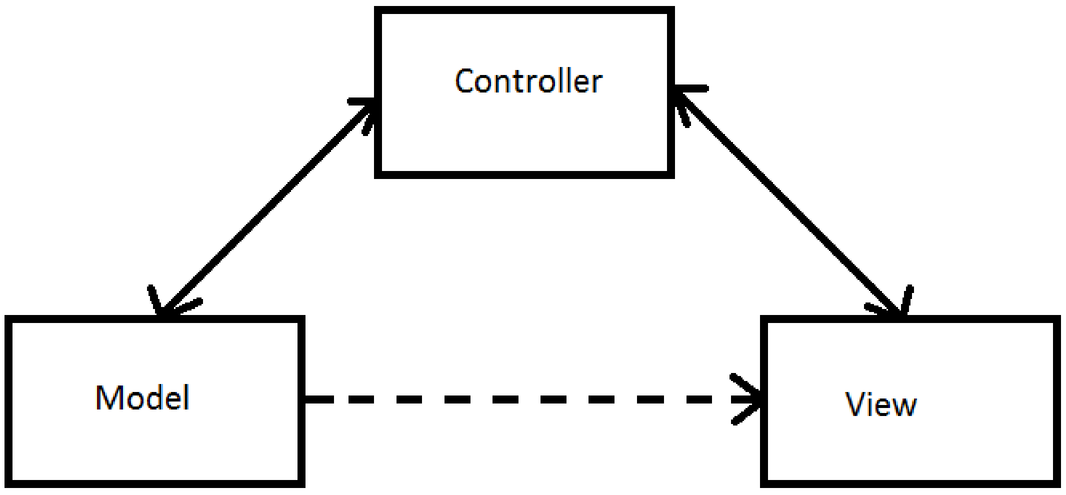
Unit 1c: Applications & Architecture

1. Architecture

Architecture refers to the components of a software solution and how they interact with each other. A wide range of architectural patterns are available and we are going to work with a simplified version of the Model-View-Controller (MVC) pattern. This pattern is commonly supported in frameworks that support web app development such as Struts, Ruby on Rails, ASP.NET MVC etc.; though the pattern actually predates the development of the web and is a common way of separating concerns into components. Typical concerns of a software solution are: managing business system objects (and their persistence); handling user requests; and, presenting data/functionality to users i.e. the user interface.



While there are variations on the **MVC** pattern, and its framework implementations, particularly with regard to the ways in which the components interact; each component has a specific focus.

Model: represents the data within the application domain i.e. essentially implements the business system objects within the application memory – may also implement business rules closely associated with the domain objects e.g. data integrity. The model may be persisted in an external form such as a database.

View: represents the user interface i.e. presents a user with (formatted) data from the model and the ability to make requests for CRUD (Create-Retrieve-Update-Delete) activity on the model.

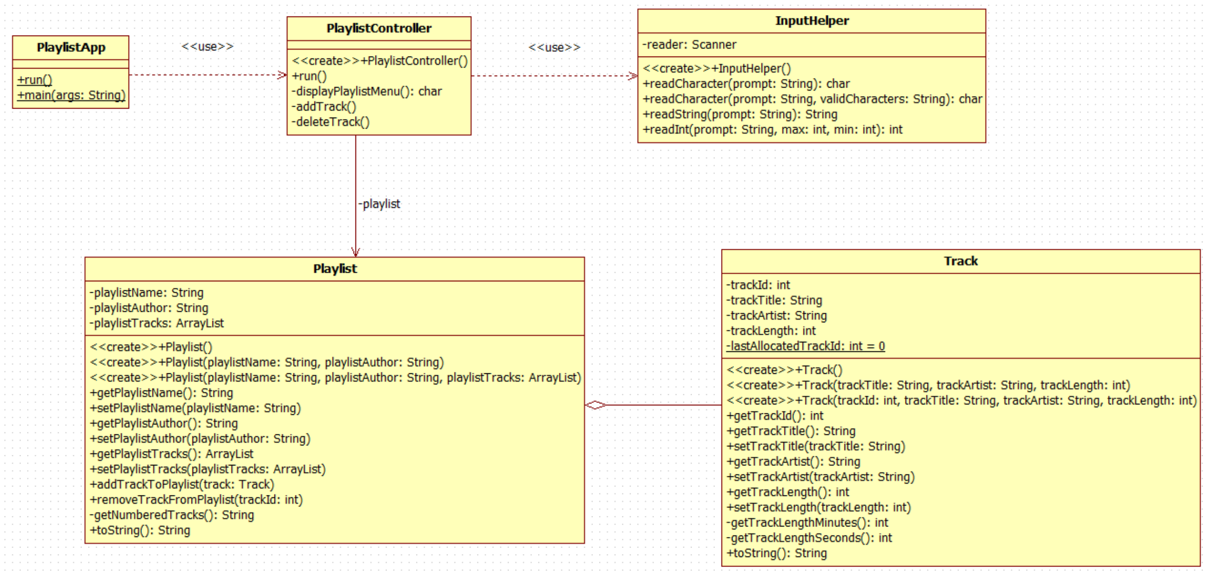
Controller: responds to user requests by querying/updating the model. Commonly it will be responsible for rendering a view based on data provided by the model – in some frameworks, views are closely linked to model objects and updated automatically when the model state changes. A controller basically implements a use case - often there will be separate controllers for each main use case.

As we are going to implement code from scratch rather than employ a framework there is insufficient time available to cover the whole architectural pattern. Therefore we are going to focus on building model classes and limit our apps to a single controller – the scenarios are simple enough to encapsulate required functionality as a single use case.

2. Playlist Application Example

Each unit will focus on a separate application which you will implement in the lab exercises. The first application we will construct will manage the construction of a playlist and its tracks.

a. Playlist Application Class Diagram


PlaylistApp
+run()
+main(args: String)
<<use>>
PlaylistController
<<create>>+PlaylistController()
+run()
-displayPlaylistMenu(): char
-addTrack()
-deleteTrack()
<<use>>
InputHelper
-reader: Scanner
<<create>>+InputHelper()
+readCharacter(prompt: String): char
+readCharacter(prompt: String, validCharacters: String): char
+readString(prompt: String): String
+readInt(prompt: String, max: int, min: int): int
-playlist
Playlist
-playlistName: String
-playlistAuthor: String
-playlistTracks: ArrayList
<<create>>+Playlist()
<<create>>+Playlist(playlistName: String, playlistAuthor: String)
<<create>>+Playlist(playlistName: String, playlistAuthor: String, playlistTracks: ArrayList)
+getPlaylistName(): String
+setPlaylistName(playlistName: String)
+getPlaylistAuthor(): String
+setPlaylistAuthor(playlistAuthor: String)
+getPlaylistTracks(): ArrayList
+setPlaylistTracks(playlistTracks: ArrayList)
+addTrackToPlaylist(track: Track)
+removeTrackFromPlaylist(trackId: int)
-getNumberedTracks(): String
+toString(): String
Track
-trackId: int
-trackTitle: String
-trackArtist: String
-trackLength: int
-lastAllocatedTrackId: int = 0
<<create>>+Track()
<<create>>+Track(trackTitle: String, trackArtist: String, trackLength: int)
<<create>>+Track(trackId: int, trackTitle: String, trackArtist: String, trackLength: int)
+getTrackId(): int
+getTrackTitle(): String
+setTrackTitle(trackTitle: String)
+getTrackArtist(): String
+setTrackArtist(trackArtist: String)
+getTrackLength(): int
+setTrackLength(trackLength: int)
-getTrackLengthMinutes(): int
-getTrackLengthSeconds(): int
+toString(): String

This example is representative of the approach we take through the lab exercises and into the coursework. We have an application class which kicks off the application; a controller class which implements the required app functionality utilising a helper class which provides common user interaction functionality such as requesting integer or string values i.e. reusable functionality. Finally, we have the model classes: in this case **Playlist** and **Track** classes. Note our view functionality will simply involve the controller object directing output to the console window as user interface and receiving keyboard input.

Later units will add other classes representing more complex models and providing greater separation of concerns in managing model objects and their persistence.

3. Application Classes

In Java an application begins execution by running a `main()` method in an application class. We are adopting the approach of minimising the application object's responsibility – there are a wide range of views on this.

Our `main()` method will simply create an instance of the application class and then invoke a `run()` method on that object to provide the application functionality.

```
public static void main(String[] args) {
   PlaylistApp playlistApp = new PlaylistApp();
   playlistApp.run();
}
```

In fact, we are going to go further: the `run()` method of the application object creates a controller object to control the operation of the application.

```
public static void run() {
   PlaylistController playlistController =
                                 new PlaylistController();
   playlistController.run();
}
```

This is all our application classes are going to do; a larger application may require to create separate controller objects for each implemented use case and may require to interact with users to determine which controller objects to create and run.

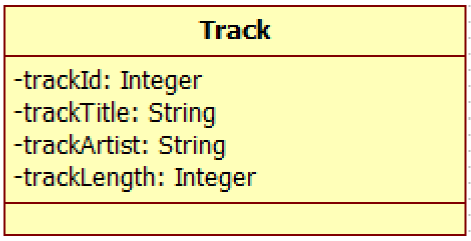
4. Model Classes

The responsibilities of model classes are to represent the actual data associated with the objects of the business system i.e. in-memory objects that the controller object can query and modify.

Recall a model class will specify the attributes of the business system object and their data types. The behaviour of a model class object is specified by a common set of methods: constructors; getters, setters, and, other useful methods such as `toString()`.

a. Example (Track Class)

Attributes

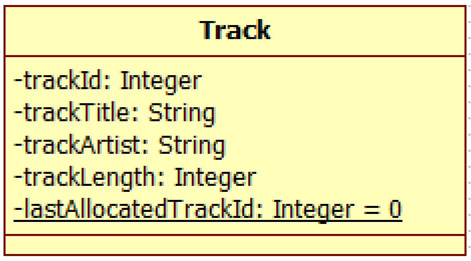


A **Track** object has four attributes:

- *trackId*: an integer which acts as a unique identifier to distinguish one track from all other tracks
- *trackTitle*: a String holding the title of the track
- *trackArtist*: a String holding the artist of the track
- *trackLength*: an integer holding the length of the track in seconds

All of these attributes will have a **private** modifier so they cannot be directly accessed from outside the object i.e. **public** methods will be required to access & modify attribute values.

We have indicated that we require the *trackId* to be used to uniquely identify tracks, so when a new **Track** object is created a new value should be generated. We'll make this the responsibility of the class by using a class variable *lastAllocatedTrackId*; when a new **Track** object is created this can be incremented and used to set the *trackId* property.



Constructors

Now we should consider the required methods. We'll start with constructors; recall a constructor allows the instantiation of an object, since we may know differing values for the attributes at the time of creation; we normally require more than one constructor - they will differ in their signature i.e. their argument (parameter) list.

Java creates a default constructor but it is better practice to define your own - a parameter-less constructor.

Constructor 1: no values supplied at instance of creation - require to set *trackId* value.

```
public Track() {
   this.trackId = ++lastAllocatedTrackId;
}
```

Constructor 2: title, artist and length supplied - require to generate new *trackId* value and set the values of the object attributes *trackTitle*, *trackArtist* and *trackLength*.

```
public Track(String trackTitle, String trackArtist,
             int trackLength) {
   this.trackId = ++lastAllocatedTrackId;
   this.trackTitle = trackTitle;
   this.trackArtist = trackArtist;
   this.trackLength = trackLength;
}
```

Constructor 3: a track object is required where previously a trackId had been allocated e.g. where the track information is being loaded from a persistence store - require to set the attribute values trackId, trackTitle, trackArtist and trackLength.

```
public Track(int trackId, String trackTitle,
             String trackArtist, int trackLength) {
   this.trackId = trackId;
   this.trackTitle = trackTitle;
   this.trackArtist = trackArtist;
   this.trackLength = trackLength;
   if (trackId > lastAllocatedTrackId)
       lastAllocatedTrackId = trackId;
}
```

Accessors (getters)

Accessors (getters) are public methods which allow other objects, such as a controller object, to query attribute values, typically we will have one get method for each attribute.

This allows us to encapsulate the state of an object, hide it from the outside world and provide access to those values we wish to make publicly available - **private** attributes and **public** get methods.

```
public int getTrackId() {
   return this.trackId;
}

public String getTrackTitle() {
   return this.trackTitle;
}
```

Mutators (setters)

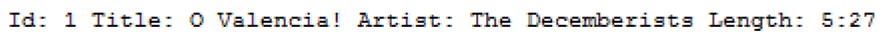
Mutators (setters) are methods which allow other objects to set/change the value of an attribute, typically we would require a set method for each attribute. However, there may be some attributes which we do not want their values to change e.g. the *trackId*.

```
public void setTrackArtist(String trackArtist) {
   this.trackArtist = trackArtist;
}


public void setTrackLength(int trackLength) {
   this.trackLength = trackLength;
}
```

Other methods

In Java, all classes descend from the Object class and so inherit a **toString()** method which provides a `String` representation of an object's attribute values; this should be customised to suit the class you are defining i.e. overriden – note override annotation.

```
@Override
public String toString() {
   return "\nId: " + Integer.toString(this.trackId) +
          " Title: " + this.trackTitle +
          " Artist: " + this.trackArtist +
          " Length: " +
          Integer.toString(getTrackLengthMinutes()) +
          ":" + Integer.toString(getTrackLengthSeconds()) +
          "\n";
}
```



```
Id: 1 Title: O Valencia! Artist: The Decemberists Length: 5:27
```

private (internal) methods

Note we wish to represent the track length as minutes and seconds rather than simply as the stored seconds value, we can use a couple of **private** methods to calculate the minutes and seconds:

```
private int getTrackLengthMinutes() {
   return this.trackLength/60;
}

private int getTrackLengthSeconds() {
   return this.trackLength % 60;
}
```

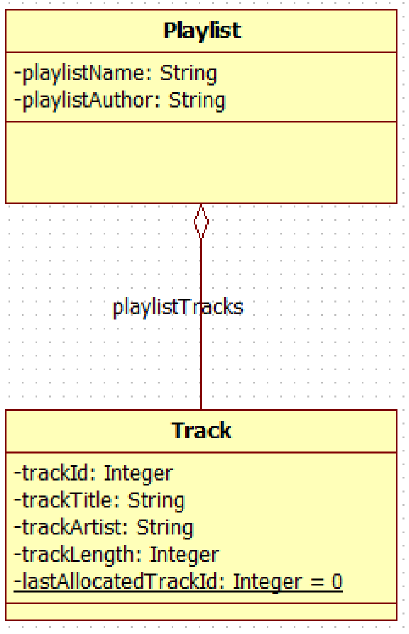
b. Example (Playlist Class)

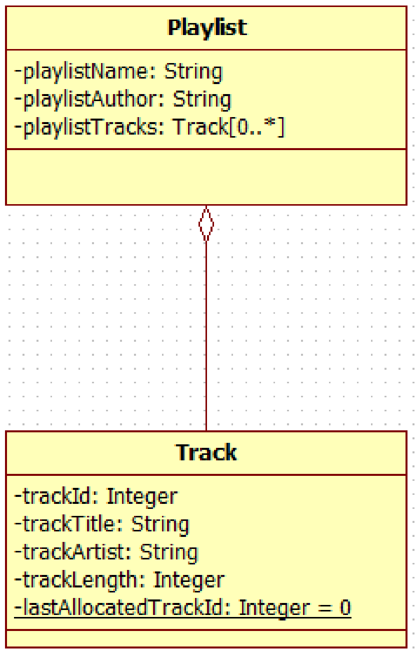
Attributes

Next we consider that a **Playlist** object collects together a number of **Track** objects, so in defining a **Playlist** class we require the attributes: playlist name, playlist author & collection of tracks. The first two are `String` attributes.

Java provides a wide range of support for collections through its **Collections Framework** which we will investigate later, for example, we could use an array of track objects quite successfully, however, an **ArrayList** provides better support.

There are two ways of representing the relationship between the **Playlist** class and the **Track** class in class diagrams: we can label the association with the name of the attribute; or, we can explicitly name and type the attribute in the **Playlist** class:





This is an example of an aggregation association where a **Playlist** object uses a collection of **Track** objects.

So the class definition with properties would look like this:

```
public class Playlist {
   private String playlistName;
   private String playlistAuthor;
   private ArrayList<Track> playlistTracks;
```

Constructors

Again we define multiple constructors with different signatures (argument lists); where a collection of tracks is not supplied at the time of creation of the **Playlist** object then the constructor should create the *playlistTracks* object using the new operator applied to the **ArrayList** class, for example:

```
public Playlist (String playlistName, String playlistAuthor) {
   this.playlistName = playlistName;
   this.playlistAuthor = playlistAuthor;
   this.playlistTracks = new ArrayList<>();
}
```

Getters and Setters

Again the class will require getters and setters for each of the attributes we wish to make accessible, for example:

```
public ArrayList<Track> getplaylistTracks() {
   return this.playlistTracks;
}
```

```
    public void setplaylistTracks(ArrayList<Track> playlistTracks)
    {
       this.playlistTracks = playlistTracks;
    }
```

Collection methods

Since we have an attribute which is a collection of other objects then we need to consider how we can add to the collection; using the **Java Collections Framework** of **ArrayList** this is simple:

```
    public void addTrackToPlaylist(Track track) {
       this.playlistTracks.add(track);
    }
```

What about removing a track from a playlist; we have implemented this using some advanced Java functionality which we will explore later in the module:

```
    public void removeTrackFromPlaylist(int trackId) {
       Predicate<Track> trackPredicate =
                                t-> t.getTrackId() == trackId;
       this.playlistTracks.removeIf(trackPredicate);
    }
```

This searches the *playlistTracks* collection for a match with a supplied track id value and then removes any matches from the collection.

5. Controller Classes

Recall that our apps are implementing a single all-encompassing use case and, therefore when we create and run an application object it simply creates a controller object and executes a `run()` method on it.

```
    public static void run() {
       PlaylistController playlistController =
                                 new PlaylistController();
       playlistController.run();
    }
```

a. Example (PlaylistController Class)

Attributes

Next let's consider the attributes and methods that the **PlaylistController** class requires; well, the application handles a single **Playlist** object as described earlier, so the **PlaylistController** class must define a *playlist* attribute:

```
public class PlaylistController {
   private final Playlist playlist;
```

Note the **final** specifier indicates that the playlist object can only be created/assigned once.

Constructor

Currently we only require a single constructor method for the **PlaylistController** class as a controller object is only created in one way.

```
public PlaylistController() {
   InputHelper inputHelper = new InputHelper();
   String playlistName =
                 inputHelper.readString("Enter Playlist Name");
   String playlistAuthor =
                 inputHelper.readString("Enter Playlist Author");
   playlist = new Playlist(playlistName, playlistAuthor);
}
```

Note the constructor creates an **InputHelper** object to deal with the interaction with the user; this object asks for and returns values for playlist name and author. A **Playlist** object is created using the two parameter **Playlist** constructor which will initialize an empty *playlistTracks* collection.

private (internal) methods

The **PlaylistApp** will then execute the **`run()`** method of the **PlaylistController** to provide a menu-driven application which will allow users to add and delete tracks from the playlist.

A **private** method of the **PlaylistController** class displays a menu to the user and captures a menu choice; currently the choice is limited to add a track and remove a track so the **`run()`** method needs to invoke **`addTrack()`** or **`deleteTrack()`** dependent on the user choice:

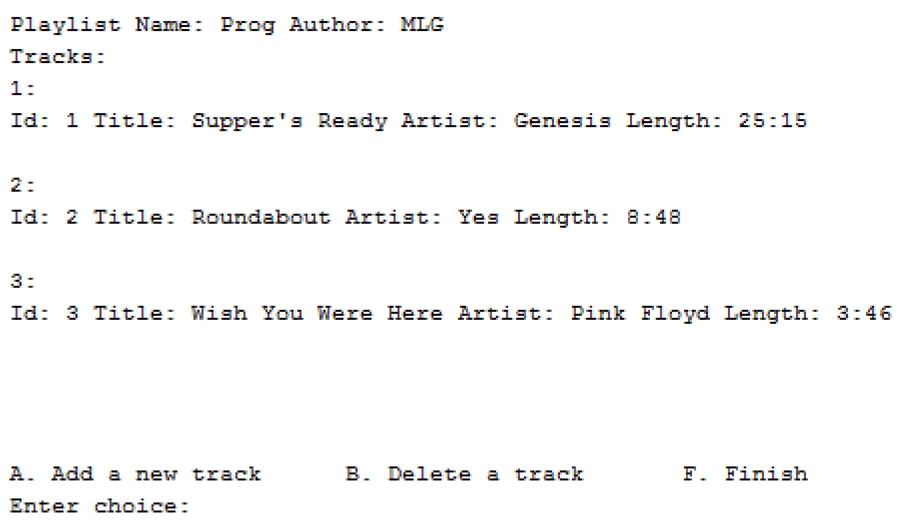
```
public void run() {
   boolean finished = false;
   do {
      System.out.println(playlist);
      char choice = displayPlaylistMenu();
      switch (choice) {
         case 'A':
            addTrack();
            break;
         case 'B':
            deleteTrack();
            break;
         case 'F':
            finished = true;
      }
   } while (!finished);
}
```

Let's consider the functionality required of the **`addTrack()`** method; it should capture user input of track title, artist and track length - again an **InputHelper** object is used to capture the user input. A new **Track** object is created and this **Track** object added to the playlist:

```
      private void addTrack() {
         InputHelper inputHelper = new InputHelper();

         String trackTitle =
                  inputHelper.readString("Enter Track Title");
         String trackArtist =
                  inputHelper.readString("Enter Track Artist");
```

```
       int trackMinutes =
                inputHelper.readInt("Enter Track Minutes", 60, 0);
       int trackSeconds =
                inputHelper.readInt("Enter Track Seconds", 60, 0);

       Track newTrack = new Track(trackTitle, trackArtist,
                                trackMinutes * 60 + trackSeconds);

       playlist.addTrackToPlaylist(newTrack);
    }
```

b. Sample App Run



```
Playlist Name: Prog Author: MLG
Tracks:
1:
Id: 1 Title: Supper's Ready Artist: Genesis Length: 25:15

2:
Id: 2 Title: Roundabout Artist: Yes Length: 8:48

3:
Id: 3 Title: Wish You Were Here Artist: Pink Floyd Length: 3:46



A. Add a new track      B. Delete a track       F. Finish
Enter choice:
```

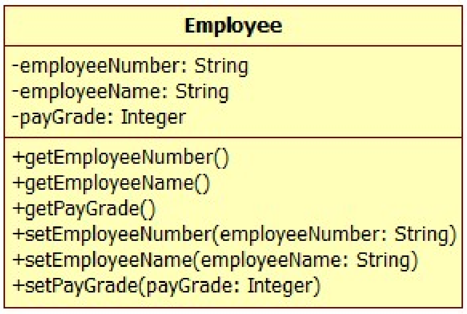
6. Discussion Questions

1. The Playlist Application class diagram details five classes; classify each as either model or functionality.

2. A UML class diagram can utilise a number of specifiers to clarify intent; explain the purpose of each of the ones listed below:

   -   +   underline   <<create>>

3. An Employee class is specified in the class diagram below; note no constructor methods have been specified. Amend the diagram to include constructor methods and any other methods you think are appropriate.



4. Examine the Sample App Run above and the Playlist Application class diagram to identify which objects are created and their classes.