ECS510
Algorithms and Data Structures in an Object Oriented Framework

"ADSOOF"

`Strings` and `ArrayLists`:
Java's Built-in Classes

`Strings`

- In Java, strings are objects
- Java library class `String` defines them
- If `str` is of type `String`
  - `str.length()` is the length of the string it refers to
  - `str.charAt(`*`expr`*`)` is the character (type `char`) at position *`expr`* where *`expr`* evaluates to an integer
- Strings have a “literal representation” - the characters in them surrounded by double quotes e.g. `"fred"`

`Strings` are immutable

- There is no method in class `String` which changes the `String` object it is called on
- Don't think that e.g.

  `str.charAt(i)='x';`

  would work because `a[i]='x'` would work if `a[i]` is of type `char[]`. Remember, `charAt(i)` is a method call

`String` methods

Java's `String` class provides many useful methods on strings, examples:

- `str.replace(ch1,ch2)` replace all occurrences of `ch1` by `ch2`
- `str.toUpperCase()` change all lower case letters to equivalent upper case
- `str.trim()` remove all leading and trailing blanks

These are all constructive, they return new `Strings`

More `String` methods

For the purposes of this module you don't need to know all the methods in class `String`, but the following are important:

- `str.substring(p1,p2)` returns a string which is those characters from `str` starting at position `p1` up to but not including position `p2`
- `str.substring(p)` returns a string which is those characters from `str` starting at position `p` up to the last character
- `str1.compareTo(str2)` returns an integer which is negative if `str1` is before `str2` alphabetically, positive if `str1` is after `str2` alphabetically, and 0 if they are equal

`String` equality

- With `str1` and `str2` of type `String`, `str1==str2` is an alias test
- Two `String` objects may contain the same characters in the same order, but may not be aliases
- So `str1.equals(str2)` is how to test equality
- This is because the method `equals` is defined in class `String` so that `str1.equals(str2)` returns `true` if `str1` and `str2` contain the same characters in the same order, `false` otherwise

`String` concatenation

- The `+` operator when used with `String`s is a concatenation operator
- For example if `str1` is `"black"` and `str2` is `"berry"` then `str1+str2` is `"blackberry"`
- When `+` is used with a `String` an another object, the reference to the object is automatically replaced by a call to `toString()` on it

The `toString()` method

- Every class has a method with signature

```
public String toString()
```

- It is inherited from the "most general class", `Object`
- Other methods inherited from `Object` include `equals` and `clone`
- The default behaviour of these methods may not be what you want, so when you write your own classes you may have to override these methods with your own code

Recursion with `Strings`

- Recursion is when "a method calls itself", here is an example:

```
public static boolean
   startsWith(String str1,String str2) {
 if(str2.length()==0)
    return true;
 else if(str1.length()==0)
   return false;
 else if(str1.charAt(0)!=str2.charAt(0))
   return false;
 else
    return startsWith(str1.substring(1),
                      str2.substring(1));
}
```

Recursion

- Recursion is when “a method calls itself”, but a better way of putting it is that “a method call makes a new call to the same method”
- This emphasises that a method call has its own variables of the names given by the parameters and any local variables declared in the method, you cannot assign a value to a variable of a particular name in one method call and cause the variable of the same name in another call to the same method to change its value
- So each call to `startsWith` here has its own variables called `str1` and `str2`, whereas a solution using a loop (“iteration”) would just have two variables whose values are changed

Iteration

- Here is an iterative version of the same operation:

```
public static boolean
    startsWith(String str1, String str2)
{
 int i;
 for(i=0; i<str1.length()&&i<str2.length(); i++)
      if(str1.charAt(i)!=str2.charAt(i))
         break;
 return i==str2.length();
}
```

- Strings are usually best processed iteratively, but recursion is another option, it means working by passing a smaller `String` to a recursive call rather than changing index values

Tail Recursion

- Here is another iterative version of the same operation:

```
public static boolean
    startsWith(String str1,String str2) {
 while(str2.length()!=0&&str1.length()!=0&&
       str1.charAt(0)==str2.charAt(0))
  {
   str1=str1.substring(1);
   str2=str2.substring(1);
  }
 return str2.length()==0;
}
```

- This is closer to the recursive version, except that as it is iterative variables `str1` and `str2` have their values changed instead of there being separate variables of the same name with different values in each recursive call

`Scanner`

An object of library type `Scanner` reads text

- `Scanner in = new Scanner(System.in);`

  declares a `Scanner` variable called `in` which refers to an object which reads from the console window
- `Scanner f = new Scanner(new File(name));`

  declares a `Scanner` variable called `f` which refers to an object which reads from the file named by the `String` referred to by `name`.

`Scanner` methods

If `in` is of type `Scanner`

- `in.next()` returns `String` giving next word (up to next blank character or new line), and reads past it
- `in.nextLine()` returns `String` of all characters up to but not including the next new line character, and reads past it
- `in.nextInt()` returns `int` giving next word converted to an integer if that is possible (exception thrown if it is not)
- `in.hasNextInt()` return `true` if next word can be interpreted as an integer, `false` otherwise

There are many more, you don't need to know them

Wrapper classes

- Each primitive type in Java has an equivalent object type, `int` has `Integer`, `char` has `Character`, `double` has `Double` and so on.
- Conversion is automatic, an `int` is converted to an `Integer` (boxing) and an `Integer` converted to an `int` (unboxing) when necessary (but not in versions of Java before Java 5)
- The wrapper classes also have useful static methods dealing with their primitive equivalent, for example `Character.isUppercase(ch)` returns `true` if `ch` is an upper case character, `false` otherwise.

`ArrayLists`

- The class `ArrayList` is part of Java's "Collections Framework"
- In early versions of Java, `Vector` was used where `ArrayList` would now be used
- As `ArrayList` is a "generic type", properly it should be written `ArrayList<E>`

Generic types

- A generic type should be combined with another type (the base type) to form a full type e.g `ArrayList<Integer>`, `ArrayList<String>`, `ArrayList<DrinksMachine>`, `ArrayList<ArrayList<Integer>>`
- The base type cannot be a primitive type, so use the equivalent wrapper class

`ArrayLists` as Arrays

- An `ArrayList` object resembles an array in some ways
- It is a collection of items of its base type indexed by integers from 0 to one less than its size
- If `a` is of type `ArrayList<String>` then `a.get(i)` returns the string at position `i` and `a.set(i,str)` changes the string at position `i` to `str`
- But we cannot have `a.get(i)=str` like array `a[i]=str` because `get(i)` is a method call, you cannot assign to a method call
- Do not confuse the `[ ]` of arrays, which contains an index, with the `< >` of ArrayLists which contains a type.

Raw Types

- If `a` is of type `ArrayList<`*`T`*`>` for any *`T`*, then `a.get(i)` can be used where a value of type *`T`* is needed, and `a.set(i,t)` expects `t` to be of type *`T`*.
- Java allows `ArrayList` on its own to be used as a type, this is to maintain compatibility with older versions of Java that did not have generics
- With a "raw type" like this, there is no way of ensuring objects in a collection are of the same type, and type casting has to be used when extracting them: `t = (`*`T`*`) a.get(i);`

`ArrayLists` as flexible sized arrays

- Unlike array objects, an `ArrayList` object can change its size
- If `a` is of type `ArrayList<String>` and `str` is of type `String` and `i` is of type `int`:
  - `a.add(str)` adds `str` to the end of the list, increasing its size by one
  - `a.add(i,str)` adds `str` at position `i`, everything after it is pushed up one place
  - `a.remove(i)` removes the string at position `i`, everything beyond it is moved down one place
  - `a.remove(str)` removes the lowest indexed occurrence of `str` and moves everything following it down one place, leaves it unchanged if `str` does not occur

Starting `ArrayList` objects

- `ArrayList<String> a;`
  declares a variable called `a` of type `ArrayList<String>`
- `a = new ArrayList<String>();`
  creates a new `ArrayList<String>` object and sets `a` to refer to it
- Declaring a variable and creating an object are NOT the same thing!
- `a.size()` returns the current size of the `ArrayList<String>` object referred to by `a`

Building `ArrayList` objects

- The statement

  ```
  a = new ArrayList<String>();
  ```

  sets `a` to an `ArrayList` of size 0, the `ArrayList` can be increased in size by adding things to it
- This is different from array where you have to create an array of the size you want, then set its locations to the things you want to store
- There isn't a constructor equivalent to `new String[n]` with arrays which creates an arrayList with `n` unfilled locations already there
- Note `new ArrayList<String>(n)` is allowed, but doesn't do what you might think

Copying an `ArrayList`

```
ArrayList<String> copy(ArrayList<String> a)
{
 ArrayList<String> b = new ArrayList<String>();
 for(int i=0; i<a.size(); i++)
   b.add(a.get(i));
 return b;
}
```

- But `ArrayList` has a "copy constructor":

```
ArrayList<String> al1;
…
ArrayList<String> al2 = new ArrayList<String>(al1);
```

- This is "shallow copy", new object, contents aliased

Searching an `ArrayList`

```
public static boolean isIn(ArrayList<String> a,
   String w)
{
 int i=0;
 for(; i<a.size(); i++)
    if(a.get(i).equals(w))
       return true;
 return false;
}
```

- Similar to what we saw with arrays, but built-in `a.contains(str)` does the same

Destructive Change

```
public static void
 change(ArrayList<String> a,String w1,String w2)
 {
  for(int i=0; i<a.size(); i++)
    if(a.get(i).equals(w1))
       a.set(i,w2);
 }
```

- Similar to what we saw with arrays

Constructive Change

```
public static ArrayList<String>
 constChange(ArrayList<String> a,String w1,String w2)
 {
  ArrayList<String> b = new ArrayList<String>();
  for(int i=0; i<a.size(); i++)
   {
    String next = a.get(i);
    if(next.equals(w1))
       b.add(w2);
    else
       b.add(next);
   }
  return b;
 }
```

- Different from what we saw with arrays, collection grows in size

Destructive change in size

- With `ArrayLists`, unlike arrays, we can have destructive methods which change the size of the collection:

```
public static void
 addAfter(ArrayList<String> a,String w1,String w2)
 {
  for(int i=0; i<a.size(); i++)
    if(a.get(i).equals(w1))
       {
        a.add(i+1,w2);
        i++;
       }
 }
```

- The `i++` is needed to prevent an infinite loop when `w1` and `w2` are the same, always check for subtleties like this

Arrays v. `ArrayLists`

- `Thing[]` is the type "array of `Thing`"
- `a[i]` is a "variable variable" (the memory location it accesses changes as `i` changes)
- Arrays are of fixed size, you create one then fill it in
- Arrays correspond directly to computer memory

- `ArrayList<Thing>` is the type "arrayList of `Thing`"
- `a.get(i)` and `a.set(i,t)` are method calls
- `ArrayLists` can change size, you create one of size 0 and add to it
- `ArrayLists` are implemented by Java code

Need for generic methods

In the previous examples, if we were dealing with `ArrayList`s of integers rather than strings, the code would look almost identical:

```
public static void
 change(ArrayList<Integer> a,Integer n1,Integer n2)
 {
  for(int i=0; i<a.size(); i++)
    if(a.get(i).equals(n1))
       a.set(i,n2);
 }
```

Is there a way round writing lots of very similar methods for collections of different types?

Generic methods

- To make a method generic, declare a type variable before the return type, then use that type variable as a type
- But the only method you can call on an object of the type of the type variable is one inherited from `Object`, such as `equals`
- Java has a way of getting round this, see later

Generic constructive change

```
public static <T> ArrayList<T>
 constChange(ArrayList<T> a, T m, T n)
 {
  ArrayList<T> b = new ArrayList<T>();
  for(int i=0; i<a.size(); i++)
   {
    T next = a.get(i);
    if(next.equals(m))
       b.add(n);
    else
       b.add(next);
   }
  return b;
 }
```

Using generic methods

- When a generic method is called, the type variable is set according to its arguments.
- So with:

  `ar2=constChange(ar1,t1,t2);`

  If `ar1` is of type `ArrayList<Integer>`, `t1` and `t2` must be of type `Integer` (or `int`) and `ar2` must be of type `ArrayList<Integer>`

  If `ar1` is of type `ArrayList<String>`, `t1` and `t2` must be of type `String` and `ar2` must be of type `ArrayList<String>`

Equality testing

Consider

```
public static <T> void destChange1(ArrayList<T> a, T m, T n)
{
  for(int i=0; i<a.size(); i++)
   {
    T next = a.get(i);
    if(next==m)
       a.set(i,n);
}
```

As we saw with `Strings`, the comparison `next==m` does not mean the same as `next.equals(m)`

Using `equals` and `==`

- In general, `obj1.equals(obj2)` is used to test if two objects are equal, it is not the same as `obj1==obj2`
- For example `list.remove(obj)` removes an object at position `i` where `list.get(i).equals(obj)` returns `true`, not only where `list.get(i)==obj` gives `true`
- The method `equals` can be implemented so that `obj1.equals(obj2)` gives `true` when `obj1` and `obj2` have identical content, even if they refer to separate objects whereas `obj1==obj2` evaluates to `true` if and only if `obj1` and `obj2` are aliases
- Code which uses `equals` depends on how it is implemented in the class of the object it is called on (“dynamic binding”)

Java's built-in classes

- Java provides many classes as part of the language, you use them by knowing their public methods
- Classes `String`, `Scanner`, `Character` provide useful methods for text handling
- Class `ArrayList` provides a more flexible way of handling indexed data than arrays
- But you don't need to memorise lots of Java's classes and their methods for this module
- The class `ArrayList` is just one example of a collection class provided as standard in Java, we will look briefly at others later
- Java provides many classes for special purposes, such as database interaction, graphical user interfaces and so on, we will not cover any of that in this module

Other issues

- We have briefly looked at recursion here, we will look at this idea in more detail in the next section on Lisp Lists
- We have also looked at generic types: a type which has one or more type variables
- `ArrayList<E>` is just one example of a generic type
- The type variables in a generic type must be set to particular types to give an actual type of an object, for example we cannot have just an `ArrayList` object, it must be an `ArrayList` of some element type
- A method can be written with type variables, meaning it is generalised so one method can work with arguments of a variety of different types, we will look at this in more detail later
- We will also look at the `equals` method in more detail later