ECS510

Algorithms and Data Structures in an Object Oriented Framework "ADSOOF"

# Implementing Objects

Return to `DrinksMachine`

and

Building our own ArrayLists

# Using Objects

- Objects in Java are defined by classes
- But we have used objects without knowing their classes
- We do need to know their public methods and constructors
- plus a specification we know the code underneath will work to
- It is important to think in these terms for large-scale programming

# Large-scale programming

- Realistic programs too large for one person to know every part of their code
- Better to think in terms of components
- Components may use objects defined by others
- Components may define objects for others to use
- Components may be re-used

# Application and Implementation

- Application - the code which uses objects of some class
- Implementation - the code inside the class which makes the objects work
- Specification - the link between the two

# Specification

- A precise definition of how objects of a class will work in the application code
- The writer of the application code will be confident they work this way
- The writer of the implementation code has to make sure they work this way
- Otherwise, they do not need to know about each other's code

# Specifying Lisp List (axioms)

- If `ls` is the result of calling `ls1.cons(h)`
  - `ls.head()` is equal to `h`
  - `ls.tail()` is equal to `ls1`
  - `ls.isEmpty()` is `false`
- If `ls` is the result of calling `LispList.empty()`
  - `ls.head()` will throw an exception
  - `ls.tail()` will throw an exception
  - `ls.isEmpty()` is `true`
- As Lisp lists are immutable, we do not need to take account of them changing

# Some specification for ArrayList

- The call `a.set(i,t)` throws an exception if `i<0` or `i≥a.size()`
- After `a.set(i,t)` but before any other mutating operation on `a`
  - `a.get(i)` is equal to `t`
  - `a.get(j)` where `j≠i` is equal to previous `a.get(j)`
- After `a.add(t)` but before any other mutating operation on `a`
  - `a.size()` is one greater than previous `a.size()`
  - `a.get(a.size()-1)` is equal to `t`
  - `a.get(i)` is equal to previous `a.get(i)` when `i` is not `a.size()-1`

- The call `a.add(p,t)` throws an exception if `p<0` or `p>a.size()`

- After `a.add(p,t)` but before any other mutating operation on `a`
  - `a.size()` is one greater than previous `a.size()`
  - `a.get(p)` is equal to `t`
  - `a.get(i)` is equal to previous `a.get(i)` when `i<p`, and is equal to previous `a.get(i-1)` when `i>p`

# Specification Language

- Formal specification could use logic-like language, this aids automatic proof
- Formal specification in terms of pre-conditions and post-conditions is good, but may get complex
- In practice specification is more likely to be in formal English
- Specification is a major issue which could be covered in much more detail in a more formal module
- But getting used to precise explanations in English is an important part of this module

# Static and instance methods

- Static methods are self-contained, work in their own environments
- Non-static methods (instance methods) are called “on an object” and work in an environment which contains the method’s variables plus the object’s variables
- As with static methods, the method’s variables are new for each call
- The object’s variables remain in existence for as long as the object, and instance method calls alter them permanently

# Self-reference

- In the code for an instance method the keyword `this` refers to the object the method call is made on
- If an instance method is called not attached to any object, it is assumed to be attached to `this`
- A constructor is like an instance method, but `this` refers to the new object it is constructing
- The call `this(...)` as the first statement in a constructor is a separate thing (means use another constructor in the same class)

# Object Variables

- Object variables are declared inside a class but outside methods in the class, and are not `static`
- Each object created by the class's constructor has its own object variables of the name given by the object variable declaration
- When an instance method uses an object variable name, it means the variable of that name of the object the method is called on
- But it can use the object variables of another object of the same class by attaching the name to reference to the other object

# Example - `DrinksMachine`

```
class DrinksMachine
{
 private ArrayList<Can> cokes, fantas;
 private int price,balance,cash;
```

means every object of type `DrinksMachine` has its own variables called `cokes`, `fantas`, `price`, `balance` and `cash`

- The `private` means they cannot be used in methods in other classes, only indirectly through calls to `DrinkMachine`'s methods
- Object variables are usually declared as `private` so that objects have control over their own state

# Example method

```
public void insert(int n)
{
 balance=balance+n;
}
```

- In a method call `m1.insert(sum)`, `n` is a local variable, assigned as if `n=sum`
- and `balance` is an object variable, inside the object referred to by `m1`
- In `m2.insert(sum)`, `balance` is an object variable inside the object referred to by `m2`
- So the two `balance`s are separate variables unless `m1` and `m2` are aliases

# Other objects' private variables

- We could write a method inside class `DrinksMachine`:

```
boolean cheaperThan(DrinkMachine m)
{
 return price<m.price;
}
```

- Then we can call e.g `m1.cheaperThan(m2)`
- Then `price` is the `price` variable in `m1`, and `m.price` is the `price` variable in `m2`

# How objects work

- So the variables `cokes`, `fantas`, `price`, `balance` and `cash` represent the internal mechanism of a drinks machine
- The changes in the values of these variables and the objects they refer to when methods are called represent the changes to the machine when it is used
- The user of the machine does not know e.g. that there are two ArrayLists inside

# Users

- The user of a program is typically a human being who interacts with its state through a Graphical User Interface
- The user of a class is code in another class which interacts with it by calling its methods
- Do not confuse these two, a human user doesn't know about the programming code which makes it work
- This module is about components, so does not cover issues of human-computer interaction

# Throwing exceptions

- A method may throw an exception
- If it's a checked exception, it has to be given in the signature, uses keyword `throws`
- An exception is an object, it has to be created through a constructor
- The statement `throw` followed by a reference to an exception causes the method call to halt and throw an exception
- Once a method call is halted through `throw`, as through `return`, it is never returned to

# Implementing ArrayList

- This is an exercise, in practice there's usually no point in not using the class Java gives us
- What we need to do
  - Consider an internal representation, and how that relates to what we think of as an "ArrayList"
  - Consider code for the methods which use this internal representation to respond according to the method specification

# ArrayList represented by Array

- An ArrayList is a numerically indexed collection of items of the same type
- An array is a numerically indexed collection of items of the same type
- So represent ArrayList by an internal array?

# MyArrayList of String

```
class MyArrayList
{
 private String[] arr;
 …
 public void set(int i,String str)
 {
  arr[i]=str;
 }

 public String get(int i)
 {
  return arr[i];
 }
```

# Generic MyArrayList

```
class MyArrayList <T>
{
 private T[] arr;
 …
 public void set(int i,T item)
 {
  arr[i]=str;
 }

 public T get(int i)
 {
  return arr[i];
 }
```

# Constructor

```
public MyArrayList(int n)
{
 arr = (T[]) new Object[n];
}
```

- Java doesn't let us do `new T[n]`, this is how to get round it (will cause a compiler warning)
- But this doesn't fit in with the way ArrayLists work - they are not fixed size like arrays
- Also this does not correspond with Java's `ArrayList<T>` constructor

# Implementing size change

```
public MyArrayList()
{
 arr = (T[]) new Object[0];
}
```

…

```
public void add(T item)
{
 T[] arr1 = (T[]) new Object[arr.length+1];
 for(int i=0; i<arr.length; i++)
    arr1[i]=arr[i];
 arr1[arr.length]=item;
 arr=arr1;
}
```

# Array and Count Representation

- Replacing the array each time the ArrayList size changes is inefficient
- Array and count representation:
  - Have array of maximum size needed, and count giving current portion of array in use
  - Throw exception if attempting to use index beyond current count value
  - For methods which change ArrayList size, change count and move items in array as necessary

```
class MyArrayList <T>
{
 private T[] arr;
 private int count;
 private static int MAX_SIZE=100;

 public MyArrayList()
 {
  count=0;
  arr = (T[]) new Object[MAX_SIZE];
 }

 public int size()
 {
  return count;
 }

 …
```

```
…

public T get(int i)
{
 if(i>=count)
    throw new IndexOutOfBoundsException();
 return arr[i];
}

public void set(int i,T item)
{
 if(i>=count)
    throw new IndexOutOfBoundsException();
 arr[i]=item;
}

public void add(T item)
{
 arr[count++]=item;
}

…
```

```
…
 public void add(int pos,T item)
 {
  if(pos>count) throw new IndexOutOfBoundsException();
  for(int i=count; i>pos; i--)
     arr[i]=arr[i-1];
  arr[pos]=item;
  count++;
 }

 public T remove(int pos)
 {
  if(pos>=count) throw new IndexOutOfBoundsException();
  T removed = array[pos];
  for(int i=pos+1; i<count; i++)
     arr[i-1]=arr[i];
  count--;
  return removed;
 }
…
```

# Abstract Data Type

- An Abstract Data Type is considered only in terms of the operations we can do on it and their specifications, so a class seen in terms of its public methods can be considered an ADT
- A data structure is a collection of values which have a particular pattern. The variables inside an object of class, and the rules which keep them to particular values, may be considered a data structure

# Information Hiding

- Keep variables in one part of a program so they can't be accessed directly from another part
- Only interaction between program parts is calling public methods
- If program parts can only interact in a few well-defined ways, there is less chance of errors occurring
- We can change the private parts (to a more efficient implementation?) so long as the public parts interact in the same way, without that causing problems to any other code

# Implementation of ADT

- So an ArrayList is an Abstract Data Type, and we have seen so far two data structures which may implement it:
  - Array
  - Array and count

# Writing your own classes

- Programming in an object-oriented language like Java is mainly about writing your own classes
- Classes define objects
- Code in other classes manipulates objects by calling their public methods
- As far as possible we should write the code for a class in a self-contained way, which means it should only need to know the public methods of other classes, and other classes should only need to know its public methods

# Object Oriented Design

- Object oriented design is about designing a system in terms of what objects it has and how they interact with each other
- It is covered in more detail in modules like Software Engineering and Systems Analysis
- In ADSOOF we are concerned with implementing the classes that define objects by writing code for them

# Top down and bottom up programming

- A “top-down” software engineering approach will consider first those objects which relate directly to the real world situation the system is working with
- Component programming is about building classes which describe objects whose purpose is general, so they will be re-used in many different systems
- Algorithms and data structures is primarily about component programming which is a “bottom up” approach

# Component Programming

- Components must be general so they can be taken and used in a variety of situations
- A general storage component like Java's `ArrayList<E>` is a good example
- A method which performs an algorithm like sorting, but coded in a general way so we can use the code whenever we need to sort a collection is another
- Commonly used components like these are available as the APIs of your programming language
- You may have to program your own more specialist components

# Java's API

- As exercise work and to help understand the principles of algorithms and data structures, in ADSOOF we often write code which implements components already in the Java API
- ADSOOF is not a module on "further Java" so we do not cover any of Java's API except what is necessary to cover the principles
- That is why exercise solutions involving use of Java API code which has not been covered in the module are missing the point
- In "real life" programming, however, you should use what the API provides you where possible

# Algorithms and Data Structures in an Object Oriented Framework

- We have covered how to define your own classes - this should have been revision material
- We have emphasised the importance of having a good specification and keeping to it
- Program components which can be relied on to interact only through a well-defined specification can be constructed and used independently
- This division of programs into components is a key aspects in developing realistic scale software
- The algorithms and data structures aspect of ADSOOF is about code which is used inside larger systems
- Understanding the distinction between “abstract data type” and “data structure” is a key part of developing the way of thinking that is needed at this level of programming