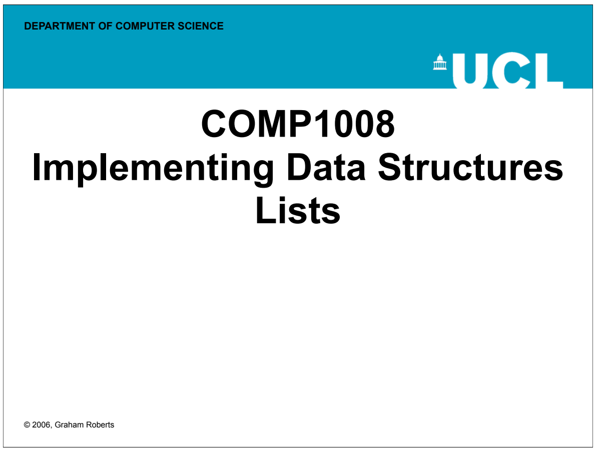
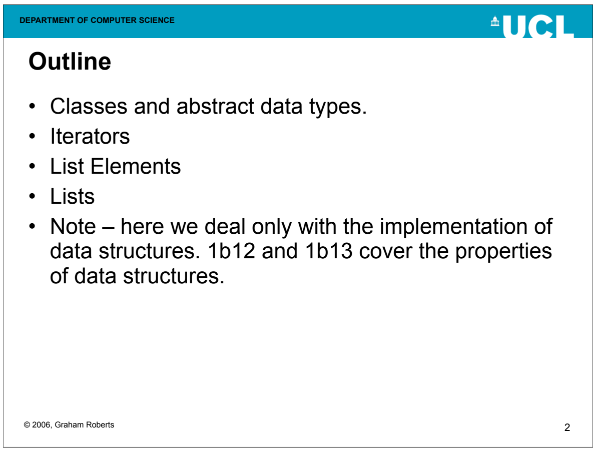
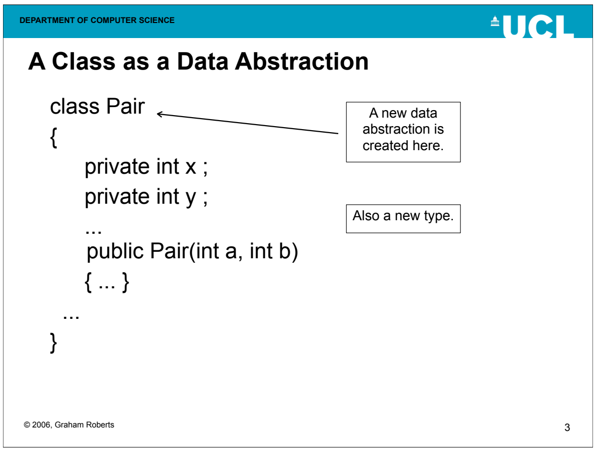


COMP1008 Implementing Data Structures Lists



Outline

- Classes and abstract data types.
- Iterators
- List Elements
- Lists
- Note – here we deal only with the implementation of data structures. 1b12 and 1b13 cover the properties of data structures.

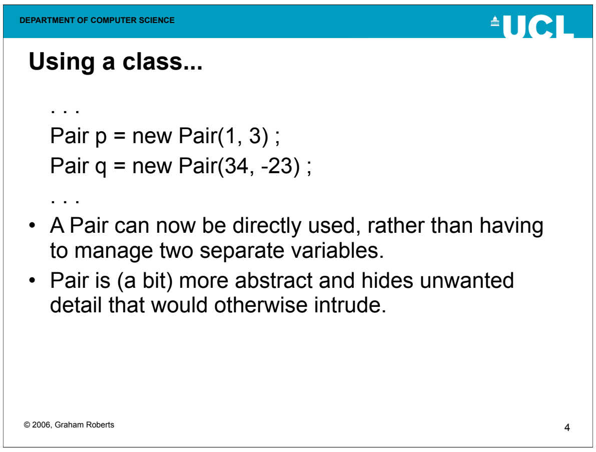


A Class as a Data Abstraction

```
class Pair
{
     private int x ;
     private int y ;
     ...
     public Pair(int a, int b)
     { ... }
   ...
}
```

A new data abstraction is created here.

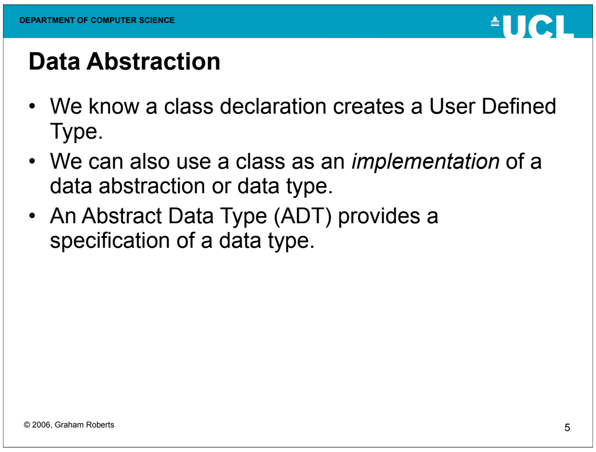
Also a new type.



Using a class...

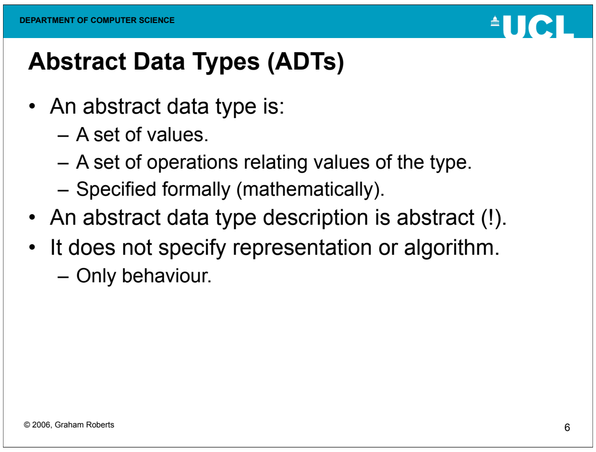
```
. . .
Pair p = new Pair(1, 3) ;
Pair q = new Pair(34, -23) ;
. . .
```

- A Pair can now be directly used, rather than having to manage two separate variables.
- Pair is (a bit) more abstract and hides unwanted detail that would otherwise intrude.



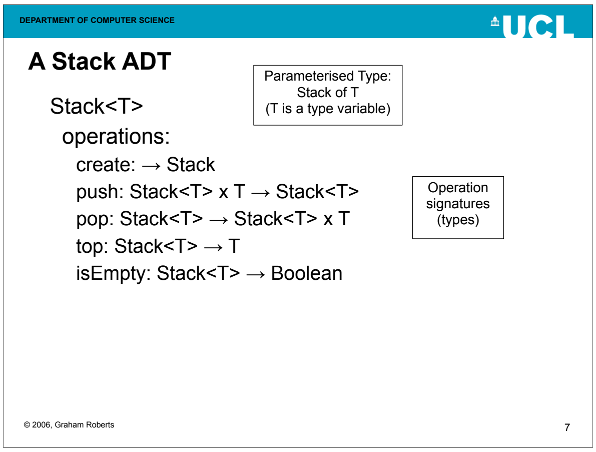
Data Abstraction

- We know a class declaration creates a User Defined Type.
- We can also use a class as an *implementation* of a data abstraction or data type.
- An Abstract Data Type (ADT) provides a specification of a data type.



Abstract Data Types (ADTs)

- An abstract data type is:
  - A set of values.
  - A set of operations relating values of the type.
  - Specified formally (mathematically).
- An abstract data type description is abstract (!).
- It does not specify representation or algorithm.
  - Only behaviour.



A Stack ADT

Stack<T>

operations:

create: → Stack

push: Stack<T> x T → Stack<T>

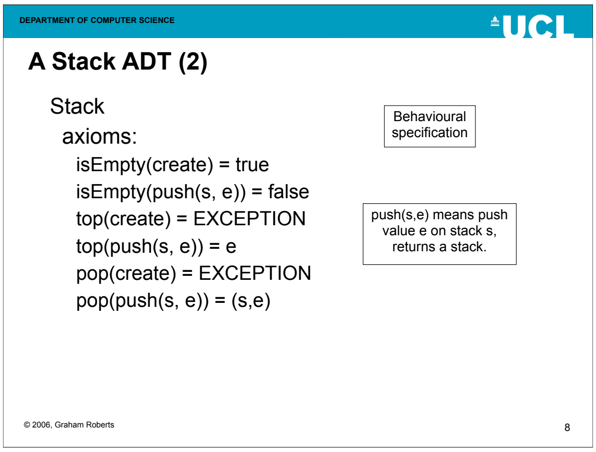
pop: Stack<T> → Stack<T> x T

top: Stack<T> → T

isEmpty: Stack<T> → Boolean

Parameterised Type: Stack of T (T is a type variable)

Operation signatures (types)



A Stack ADT (2)

Stack

axioms:

isEmpty(create) = true

isEmpty(push(s, e)) = false

top(create) = EXCEPTION

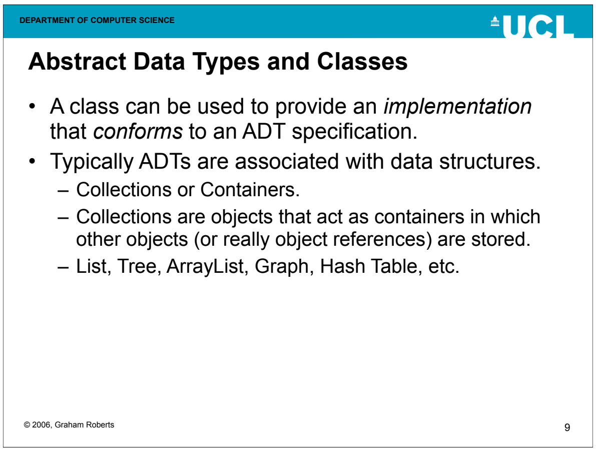
top(push(s, e)) = e

pop(create) = EXCEPTION

pop(push(s, e)) = (s,e)

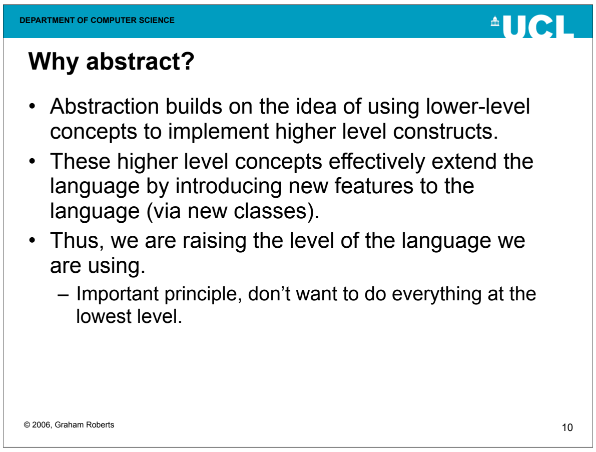
Behavioural specification

push(s,e) means push value e on stack s, returns a stack.



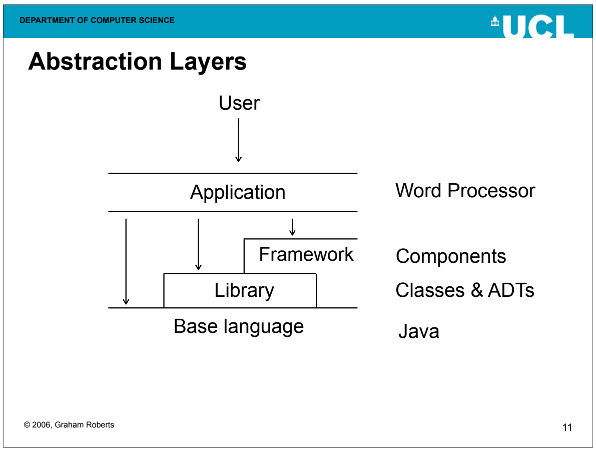
Abstract Data Types and Classes

- A class can be used to provide an *implementation* that *conforms* to an ADT specification.
- Typically ADTs are associated with data structures.
  - Collections or Containers.
  - Collections are objects that act as containers in which other objects (or really object references) are stored.
  - List, Tree, ArrayList, Graph, Hash Table, etc.



Why abstract?

- Abstraction builds on the idea of using lower-level concepts to implement higher level constructs.
- These higher level concepts effectively extend the language by introducing new features to the language (via new classes).
- Thus, we are raising the level of the language we are using.
  - Important principle, don’t want to do everything at the lowest level.



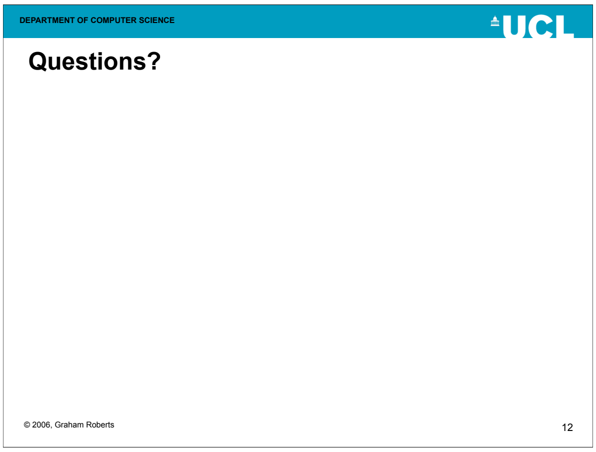
Abstraction Layers

Word Processor

Components

Classes & ADTs

Java



Questions?

Implementing a container

- Obviously use a class…
- Need a data structure to store contained *object references*:
  - one or more instance variables (private of course).
- Need algorithms to implement access operations as methods.

Implementation properties

- Need to consider:
  - Memory use.
  - Speed of operation.
- Typically trading off one property against another.
- Need to select implementations that match the needs of your program.
  - Typically have several implementations, conforming to the same interface for same abstraction.
  - List -> ArrayList, LinkedList.

Iterators

- Every container class has to provide a mechanism for accessing each element in sequence.
- Such a mechanism is called an *iterator*.
- Algorithms such as linear searching, comparison, function application depend on use of iterators.
- An iterator aims to decouple element access from container implementation.

Iterator Protocol

- Ideally we want a common iterator protocol across all of our container classes.
  - Make Iteration look the same for all container classes.
- Java provides a Collections Framework that includes various container classes and provides Iterator as the iterator protocol.

Familiar container - the Array!

- An array is a container but it is primitive and there is no class Array* (although arrays are actually objects).
- An array is a collection of items of the same type.
- The number of items is fixed.
- Efficient but low-level abstraction.
- *OK, there is a class Array but it is a collection of static utility methods.

Array Iteration

```
int[] array = new int[42] ;
for (int j = 0 ; j < array.length ; j++)
{
   doSomething(array[j]) ;
}
```

Array indexing. Depends on integer index mapping to element.

```
for (int n : array.length)
{
   doSomething(n) ;
}
```

Enhanced for loop. More generic and will work for other containers that cannot be indexed by integers.

Iterator Objects

- General abstraction of iteration.

```
ArrayList<String> a = new ArrayList<String> () ;
...
for (Iterator<String> i = a.iterator() ; i.hasNext() ; )
{
   doSomething(i.next()) ;
}
```

Ask ArrayList object for an iterator.

Iterator object stores state of iteration and gives access to next object reference.

Iterator v. Enhanced For Loop

- The enhanced for loop actually uses iterator objects.
  - Loop syntax mapped to creating/using iterator.
  - Works properly with nested loops.
- Container class should implement *Iterable* interface to work with enhanced for:

```
interface Iterable<E> {
  Iterator<E> iterator();
}
```

  - Call iterator method to get iterator object.
    - Container class responsible for provide correct iterator that works with its representation.

Iterator

- Type Iterator declared as an interface.

```
public interface Iterator<E>
{
  boolean hasNext() ;
  E next() ;
  void remove() ; // May not be supported by
                  // implementing class
}
```

Interface Reminder

- An interface declares a collection public methods.
  - All methods are abstract - no method bodies.
  - No instance variables.
  - A class implements an interface and must override the methods.
  - (Like an abstract class declaring only abstract methods.)

Iterator Class

```
class MyIterator<E> implements Iterator<E>
{
   // Must override methods
   // declared in the interface.
}
```

- An iterator object allows each value in a collection to be visited in turn (iterated).
- A variable of type Iterator can reference an object of an implementing class.
- Iterator<String> iterator = new MyIterator<String>(...);

Iterating

```
public <E> void print(Iterator<E> iterator)
{
   while (iterator.hasNext())
   {
      System.out.println(iterator.next());
   }
}
```

Programming to an interface.

- Can print contents of any data structure that can provide an Iterator implementation.
- Class of actual iterator object does not need to be known.

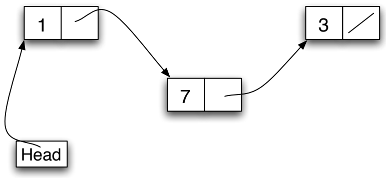
Iterator classes

- Typically declared as a nested class.
  - Inside (member of) a container class.
  - In the container class scope, so has access to private data.
  - Iterator object can access container object to get data.
  - Examples later.

Questions?

Linked Lists

- A linked list is implemented as a chain of linked elements (objects).

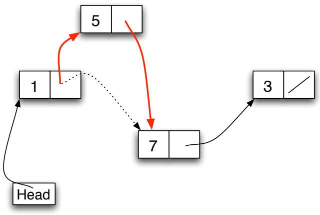


Linked Lists

- Each element or node consists of a stored value and a reference to the next element.
- A reference is maintained to the head of the list.
- An individual element is located by following the chain from the head.
  - Sequential access.
- Elements in a list (or vector, or array) are stored in sequence.
- Accessing elements relies on the sequence.
- A list is a *sequence container*.

Inserting/Removing a Value

- An element is inserted or removed by manipulating links.
- There is no need to shift other elements to add/remove space.



- Head/End are special cases.

List v. LinkedList v. ListElement

- To implement a LinkedList class we will have:
  - interface List<E>, defining public methods that all kinds of lists have.
  - class LinkedList<E>, defining a list implementation using a chain of elements.
  - class ListElement<E>, defining a list element used by LinkedList
    - ListElement will be part of the *private* implementation of LinkedList.
    - Not accessible externally.

List<E> Interface

```
public interface List<E> extends Iterable<E> {
  void insertHead(E val);
  E getHead();
  List<E> getTail();
  boolean isEmpty();
}
```

Extend the Iterable interface, so our Lists will provide a standard iterator.

Plus Iterator<E> iterator() inherited from Iterable.

List Element<E>

Nested in class LinkedList.

```
private static class ListElement<T> {
  public ListElement<T> next;
  public T val;
  public ListElement(ListElement<T> next, T val) {
    this.next = next;
    this.val = val;
  }
  public ListElement<T> copy() {
    return new ListElement<T>(next == null ? null : next.copy(), val);
  }
}
```

This is a *private* infrastructure class so *val* and *next* are public, and can be directly accessed in class LinkedList.

Helper method for copying chain of elements.

Using ListElements

- If a version of ListElement<T> is made a top level class, it could be used to created chains of objects directly.
  - Without a LinkedList class.
- Would need to provide methods to use the chain (add, remove, search, etc.).
- Useful where a full LinkedList class is not needed.

LinkedList<E>

```
public class LinkedList<E> implements List<E> {
  private ListElement<E> head;
  private static class ListElement<T> { // As seen on previous slide}
  public LinkedList() {
    head = null;
  }
  private LinkedList(ListElement<E> e) {
    head = e;
  }
  public void insertHead(E val) {
    head = new ListElement<E>(head, val);
  }
```

Private constructor is useful for LinkedList implementation but not meant to be used publicly.

LinkedList<E> (2)

```
public E getHead() {
  if (head == null) { return null; }
  else { return head.val; }
}
public List<E> getTail() {
  if ((head == null) || (head.next == null)) {
    return new LinkedList<E>();
  }
  return new LinkedList<E>(head.next.copy());
}
public boolean isEmpty() {
  return head == null;
}
```

Note that tail of list is copied.

List Iterator

- To provide an iterator, LinkedList should create and return an Iterator object.
  - Iterator knows how to access elements from the LinkedList implementation.
  - Iterator class will be another nested member class to have access to the LinkedList class scope.
  - Implements the Iterator interface, so will be a standard kind of iterator.

Getting an Iterator

- Ask the LinkedList:
  ```
  public Iterator<E> iterator() {
    return new LinkedListIterator<E>(head);
  }
  ```
- Declared in class LinkedList.
- Can have multiple iterators active at same time.
- But if list changes during iteration, iterator may break.
  - Unless a more sophisticated implementation is used.

LinkedListIterator

```
private class LinkedListIterator<T>
 implements Iterator<E> {
  private ListElement<E> current =
    new ListElement<E>(head,null);
  public LinkedListIterator(ListElement<E> e) {
   current = e;
 }
 public boolean hasNext() {
   return (current != null)
        && (current.next != null);
 }
 public E next() {
   if (current != null) {
     current = current.next;
     return current.val;
   }
   return null;
 }
public void remove()
{
  throw new UnsupportedOperationException();
}
}
```

Another nested member class.

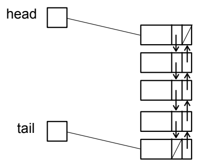
Remove is declared in Iterator interface so must be implemented. But is not supported so throws an exception.

Linked List Properties

- Inserting/removing at beginning.
- Insertion/removal in middle can be fast once the location is found.
- But there is the potential cost of linear access – O (n).
- Good for situations when elements are repeatedly inserted and deleted.
  - And where linear access is required.
  - And where number of elements is unknown or changes frequently.

Double-Link List

- Links in both directions.
- Head and tail references.
- Some algorithms easier to implement but extra storage cost for each element.



Inserting?

- Provided a “Lisp style” list that provides head/tail operations.
  - car & cdr functions
  - Natural for divide & conquer style recursive algorithms.
  - Search the web for more about Lisp.
- But what about inserting elements at any position in the list?

Insert Iterator

```
private class LinkedListInsertIterator<T>
  extends LinkedListIterator<E>
  implements InsertIterator<E> {
  private ListElement<E> last = current;
  public void insert(E value) {
    if (head == null) {
      insertHead(value);
      current = new ListElement<E>(head,null);
      return;
    }
    if (current != null) {
      current.next = new ListElement<E>(current.next,value);
    }
  }
```

Summary

- Lists are a basic data structure build from chains of elements.
  - Exploits properties of references (pointers).
  - Not fixed size, can grow and shrink.
  - Suitable where data structure size changes frequently.
  - But O(n) sequential access.
    - Start from head and search.
    - Not good good for searching/sorting.