Prof. Hamzeh Roumani
Dept of Comp Sc & Eng
York University

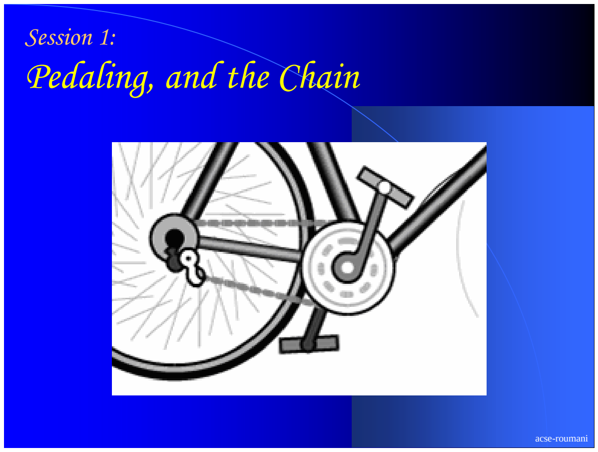
We Don't Need Arrays!

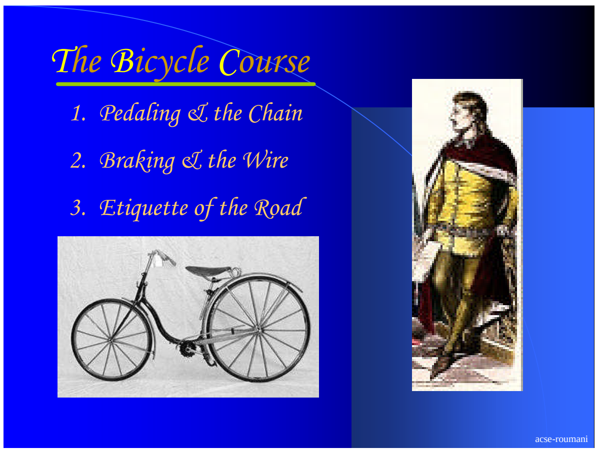
A call for a component-based software architecture

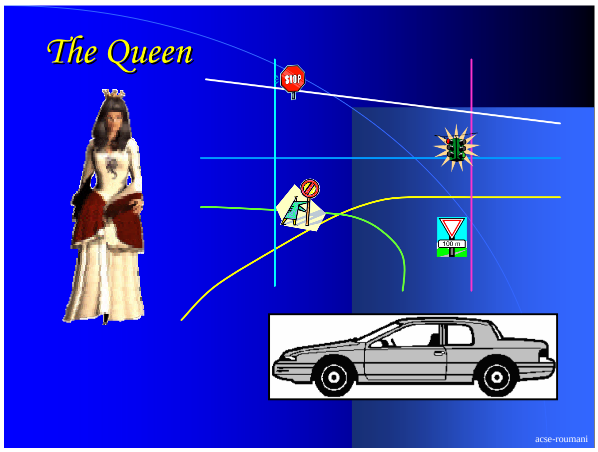
A Story

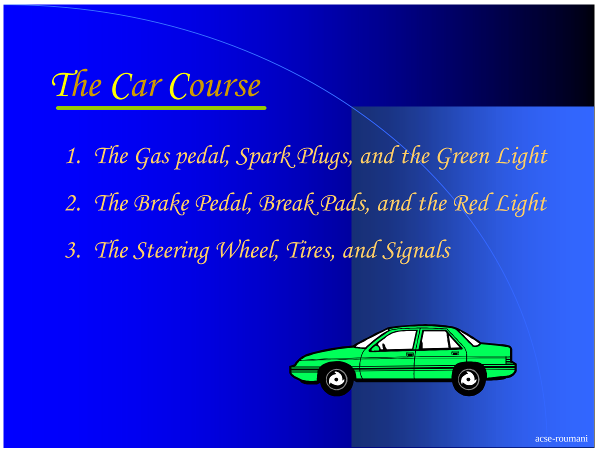
- A Story
- Reflections
- The Collection Framework
- More Delegations

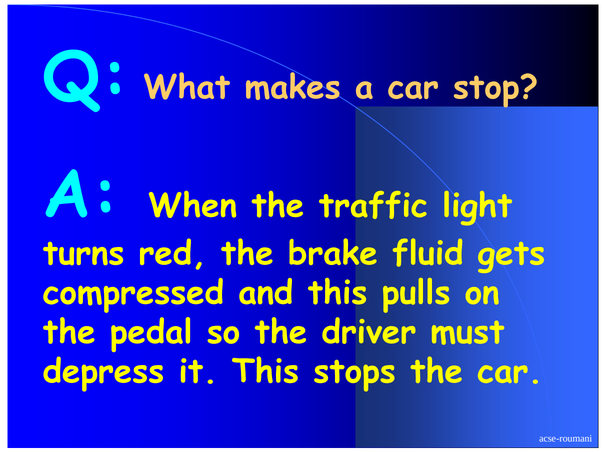

The King
The Minister
acse-roumani


Session 1:
Pedaling, and the Chain
acse-roumani


The Bicycle Course
1. Pedaling & the Chain
2. Braking & the Wire
3. Etiquette of the Road
acse-roumani


The Queen
acse-roumani

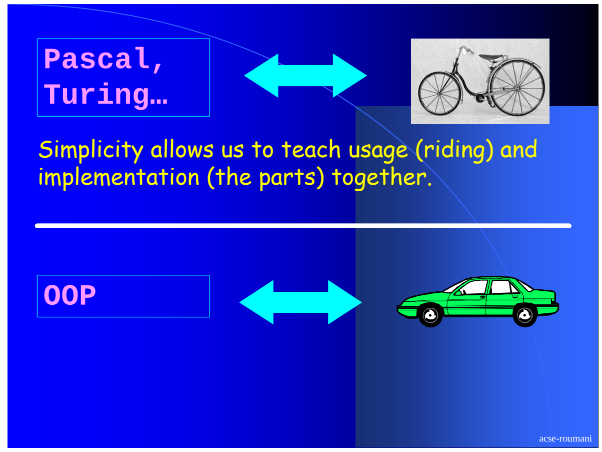

The Car Course
1. The Gas pedal, Spark Plugs, and the Green Light
2. The Brake Pedal, Break Pads, and the Red Light
3. The Steering Wheel, Tires, and Signals
acse-roumani

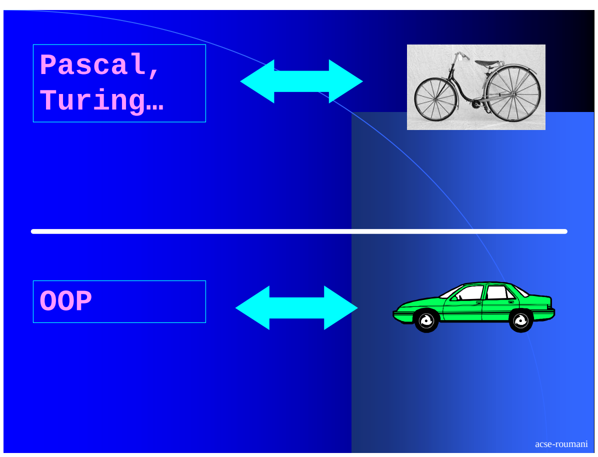

Q: What makes a car stop?
A: When the traffic light turns red, the brake fluid gets compressed and this pulls on the pedal so the driver must depress it. This stops the car.
acse-roumani

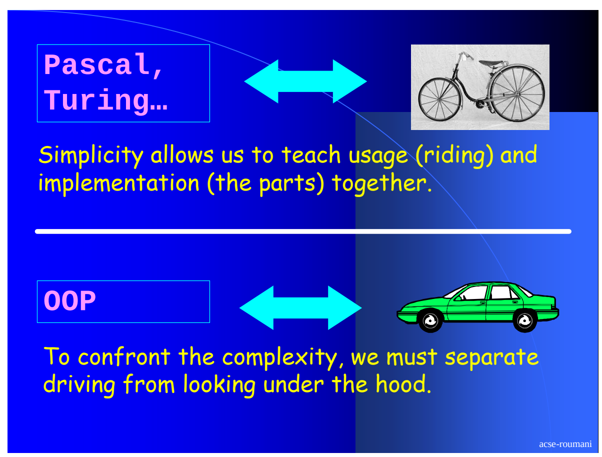

Session 2:
Stopping the Car
acse-roumani


acse-roumani


Reflections
acse-roumani


Pascal,
Turing…
Simplicity allows us to teach usage (riding) and implementation (the parts) together.
OOP
acse-roumani


Pascal,
Turing…
OOP
acse-roumani


Pascal,
Turing…
Simplicity allows us to teach usage (riding) and implementation (the parts) together.
OOP
To confront the complexity, we must separate driving from looking under the hood.
acse-roumani

- **"What" versus "How"**
  Reinventing the Wheel? Inferiority?
- **Encapsulation** (a.k.a Need-to-Know)
  Reusability… Accountability… Sbstitutability
- **Specification**
  Shift the emphasis to communication, specs, APIs

Separation of Concerns

Component World

Programming:

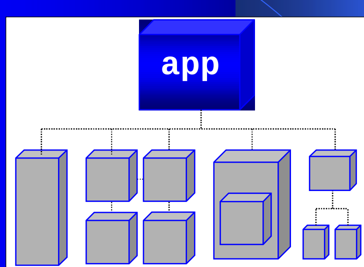
- Variables and Types
- If statements and Loops
- Components

Each Component:

- Belongs to a package
- Utility (all static) or non-utility (must instantiate)
- Concrete or not (look for a concrete that extends or implements it)

The Software of the Future

Component-Based Architecture



Our Challenge

- Launch an Editor;
- launch the API;
- and write applications that have only a `main` method.

Do not implement classes; use only the existing ones.

The Collection Framework

acse-roumani

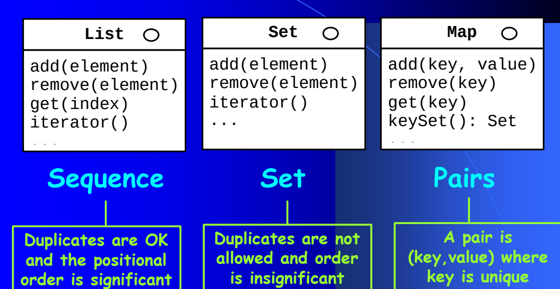
Overview

acse-roumani

- Overview of the Collection Framework
  - The Main Interfaces
  - The Implementing Classes
  - Generics
  - No More Arrays
- The Framework's API
  - Highlights
  - The Iterator
  - Searching and Sorting
  - Summary
- Applications

acse-roumani

The Interfaces



Reference: Java By Abstraction, Roumani, Pearson Addison-Wesley, Toronto (2006)

acse-roumani

The Implementing Classes

List	Set	Map
add(element) remove(element) get(index) iterator() ...	add(element) remove(element) iterator() ...	add(key, value) remove(key) get(key) keySet(): Set ...
ArrayList LinkedList	HashSet TreeSet	HashMap TreeMap

The two classes that implement each interface are equivalent in the client's view. The only visible diff is performance (running time).

Generics

All classes in the framework support generics. By specifying the type (between < and >) the client ensures:

- No rogue element can be inserted
- No casting is needed upon retrieval

Example:

```
List<Date> bag = new ArrayList<Date>();
// bag.add("Hello"); will not compile!
bag.add(new Date());
Date d = bag.get(0); // no cast!
```


The Classes, cont.

ArrayList LinkedList	HashSet TreeSet	HashMap TreeMap

- Declare using the interface, not the class
- Use LinkedList only if your app tends to add or remove elements at index 0
- Use TreeSet/Map only if you want to keep the elements sorted
- Specify the type of the elements that you intend to store in the collection

Example: A list of strings

```
List<String> bag = new ArrayList<String>();
```


API

Highlights

- Use add to add elements to lists and sets:

```
List<Date> list = new ArrayList<Date>();
Set<String> set = new HashSet<String>();
list.add(new Date());
set.add("Hello");
```

- Use put to add an element to a map

```
Map<Integer, String> map;
map = new HashMap<Integer, String>();
map.put(55, "Clock Rate");
```


Highlights

The elements of lists are indexed (starting from 0). Hence, but only for lists, we can also add and delete based on the position index:

- To insert x at position 5:

```
list.add(5, x);
```

This will work only if the list has at least 5 elements, and it will adjust the indices of all elements after position 5, if any.

- To delete the element at position 5:

```
list.remove(5);
```

This will work only if the list has at least 6 elements.

Highlights

- Use remove to delete from lists and sets:

```
boolean done = set.remove("Adam");
```

Note that remove returns `false` if the specified element was not found and returns `true` otherwise.

- To delete a map element given its key:

```
String gone = map.remove(55);
```

Note that remove in maps returns the value of the element that was removed or null if the specified key was not found.

Highlights

The elements of lists and maps (but not sets) can be retrieved using get:

- The element at position 3 in a list:

```
Date d = list.get(3);
```

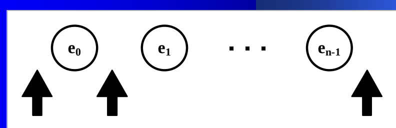
- The value of the element with key 55 in a map:

```
String s = map.get(55);
```

Note:
All interfaces come with size(), equals(), toString(), and contains (containsKey in maps).

The Iterator

- Lists and Sets aggregate an iterator
- Use iterator() to get it
- It starts positioned before the 1st element
- Use next() and hasNext() to control the cursor



The Iterator and Generics

The Iterator class supports generics; i.e. we can obtain a type-aware iterator as follows:

```
Iterator<String> it = set.iterator();
```

To benefit from this, let us rewrite the loop of the previous slide so it prints the elements capitalized:

```
Iterator<String> it = set.iterator();
for (; it.hasNext();)
{
   String tmp = it.next();
   output.println(tmp.toUpperCase());
}
```


The Iterator

The statement: `Iterator it = set.iterator();`

returns an iterator positioned just before the very first element. We use it as follows:

```
Iterator it = set.iterator();
for (; it.hasNext();)
{
   output.println(it.next());
}
```

Note that the iterator methods are not part of the collection; they are in a separate class, `Iterator`. Because of this, we can perform multiple traversals by creating one instance of `Iterator` per traversal.

The Iterator in Maps

The Map interface has no iterator() method but we can obtain a set of the map's keys:

```
public Set<K> keySet()
```

And by iterating over the obtained set, we can, in effect, iterate over the map's elements:

```
Iterator<Integer> it = map.keySet().iterator();
for (; it.hasNext();)
{
   int key = it.next();
   String value = map.get(key);
   output.println(key + " --> " + value);
}
```

Searching and Sorting

Searching

One simple (albeit inflexible) way to search a collection is to use the contains method (containsKey in maps). It determines if an element in the collection is equal to a given value and returns true or false accordingly.

```
output.print("Enter a word to look for: ");
String lookFor = input.nextLine();
output.println(set.contains(lookFor));

output.print("Enter a key to look for: ");
int findMe = input.nextInt();
output.println(map.containsKey(findMe));
```


Sorting Lists

The Collections class has the method:

```
static void sort(List<T> list)
```

It rearranges the elements of the list in a non-descending order. It works if, and only if, the elements are comparable; i.e. one can invoke the compareTo method on any of them passing any element as a parameter.

Recall that compareTo (in String) returns an int whose sign indicates < or > and whose 0 value signals equality.

Searching, cont.

For applications that require more than a simple yes/no, we use traversal-based searches. For example, find out if a given key is present in a map and output its value:

```
output.print("Enter a key to look for: ");
int find = input.nextInt();
Iterator<Integer> it = map.keySet().iterator();
boolean found = false;
Integer key = null;
for (; it.hasNext() && !found;)
{
   key = it.next();
   found = key.equals(find);
}
if (found) output.println(map.get(key));
```


Sorting and Binary Search

The main advantage of sorting is speeding up the search. When the elements are sorted, you don't have to visit all of them to determine if a given value is present in the collection or not.

```
int binarySearch(List list, T value)
```

The method searches for value in list and returns its index if found and a negative number otherwise

Note: Unlike exhaustive search (which is linear), binary search has a complexity of O(lgN).

Sorting Sets and Maps

Simply use TreeSet instead of HashSet.

The same technique applies to maps: use TreeMap instead of HashMap to keep the map's elements sorted on their keys.

Applications

LIST	SET	MAP
Adding Elements		
boolean add(E e) void add(int index, E e)	boolean add(E e)	V put(K key, V value)
Removing Elements		
boolean remove(E e) E remove(int index)	boolean remove(E e)	V remove(K key)
Accessing an Element		
E get(int index)	*none*	V get(K key)
Searching the Elements		
boolean contains(E o)	boolean contains(E o)	boolean containskey(K key)
Traversing the Elements		
Iterator iterator() *invoke on it:* E next() boolean hasNext()	Iterator iterator() *invoke on it:* E next() boolean hasNext()	Iterator keySet().iterator() *invoke on it:* E next() boolean hasNext()
Other methods (available in all three interfaces)		
equals, size, toString		
Algorithms for lists only (static methods in the Collections class)		
binarySearch, copy, fill, reverse, shuffle, sort		

Summary of Features

Fig 10.9, "Java By Abstraction", Roumani, Addison-Wesley

- Template
- FirstList, SortedList, and TraverseList
- FirstSet
- FirstMap
- WordStat
- Cryptography