CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn

Recursion is very important with complex collections of objects

We have seen it with the quick-sort: inside an array you can find smaller sub-arrays. Inside a set you find subsets. And when you subdivide everything enough, you find empty subsets, or subsets that contain a single element. It's a fantastic ground to apply recursion.

Java Generics

Book Chapter 21

But before we talk about collections of objects, we have to talk about "Generics" (an idea very similar to "Templates" in C++, although implementation is quite different), which are very important for reusing code in Java. You may have already had some exposure to Generics through ArrayLists.

Most languages that have the ambition of being used in critical applications insist on "strong typing". It means that the compiler will NOT let you mix variables of different types, unless they are known to be compatible (such as int and float). It protects against unwanted effects (note that many scripting languages take the opposite approach and guess the type of variables from how you are using them).

Java = strong typing
SAFE
but ...

```
String[] months = {"Jan", "Feb", "Mar", "Apr",
                   "May", "Jun", "Jul", "Aug",
                   "Sep", "Oct", "Nov", "Dec"};
int[]    sales = {120, 98, 75, 110, 150,
                  180, 170, 174};
```

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
120 98  75  110 150 180 170 174
```

The problem with strong typing is that for instance if you want to write a methods that displays on a line elements from an array, the same method cannot handle an array of Strings and an array of ints. It's one or the other, exclusively.

```
public static void displayArr(String[] arr) {
       int n = arr.length;

       for (int i = 0; i < n; i++) {
          if (i > 0) {
            System.out.print("\t");
          }
          System.out.print(arr[i]);
       }
       System.out.println("");
   }
```

If this is the function that displays an array of Strings ...

```
public static void displayArr(int[] arr) {
       int n = arr.length;

       for (int i = 0; i < n; i++) {
          if (i > 0) {
            System.out.print("\t");
          }
          System.out.print(arr[i]);
       }
       System.out.println("");
   }
```

Overloading

... you must overload it to display an array of integers. Note that, apart from the parameter type, the code is strictly identical.

Overloading allows you to give the same name to several functions. By looking at parameters, Java will know exactly which one to call.

Overloading

Different versions of a function

- Same name
- Same return type
- Different parameters

Flickr: Tim McCune

It's the linker of the class loader that will match your code to methods with the same name that are available.

What defines the "signature" of a function?

Number and types of parameters

```
subString(String s, int start)
subString(String s, int start, int len)

isGreater(double, double)
isGreater(String, String)
isGreater(Date, Date)
```

```
public class Overloading {
   static int function(int n) {
       System.out.println("This is function(int)");
       return 0;
   }
   static int function(double x) {
       System.out.println("This is function(double)");
       return 0;
   }
   public static void main(String[] args) {
       int    n = 1;
       double val = 0;
       float  f = 0;
       n = function(n);
       n = function(val);
       n = function(f);
   }
}
```

In this example, when we call the function with a float, it's automatically "upgraded" to double and the function for doubles is called (the same function would be called for the int if there were no special function for integers)

If overloading helps keep the code safe, it's a waste of time to write identical code several times (even if we copy and paste). Worse, if we want to change the code, for instance to separate array elements by semicolons (;) instead of tabs when we print them, we must modify every method.

Waste of time.
Change: must be repeated in all methods.

Code that works with Strings and integers?

BEWARE: in Java, generics ONLY WORK with objects (references). Base variables, such as int, float, char, boolean variables ARE NOT objects. However, all base types have a "shadow" corresponding class (Integer, Float, Character, Boolean ...). For using generics, we must use these classes, which convert automatically to and from base data types (operations known as "boxing" and "unboxing")

IMPORTANT

What we'll see only works with OBJECTS

String

int ⟹ Integer

```
String[] months = {"Jan", "Feb", "Mar", "Apr",
                   "May", "Jun", "Jul", "Aug",
                   "Sep", "Oct", "Nov", "Dec"};
Integer[] sales = {120, 98, 75, 110, 150,
                   180, 170, 174};
```

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
120 98  75  110 150 180 170 174
```

If we have an array of Strings (which are objects) and an array of Integers (that are objects too), then we can create a single method that works for both, without any explicit overloading.

"Generics" comes from a Latin word that means "family" or "kind". It can be seen as automatic overloading. Once again (it's worth repeating) it only works with references in Java.

GENERICS

Computer-aided Overloading

Only works with references

```
public static <T> void displayArray(T[] arr) {
       int n = arr.length;

       for (int i = 0; i < n; i++) {
          if (i > 0) {
            System.out.print("\t");
          }
          System.out.print(arr[i]);
       }
       System.out.println("");
   }

displayArray(months);
displayArray(sales);
```

Before the return type of the method, you specify one (or several) symbols between angular brackets that represent Classes.

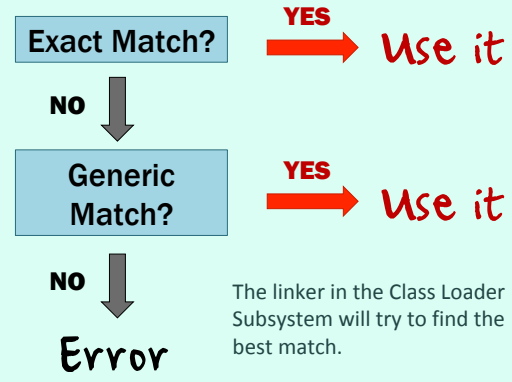
You need to specify the symbol before the return type because the return type of the generic method could perfectly be defined as T. If you need several generic classes (for instance one for the return type and one for the parameter, or because you want to pass parameters from two different classes), you separate them with commas.

<T,U>

Generic methods can be overloaded

```
public static <T> void displayArray(T[] arr)

public static <T> void displayArray(T[] arr,
                                    String sep)

public static void displayArray(Date[] arr,
                                 String format)
```

Like any method, generic methods can be overloaded, either by another generic method, or for instance to handle a very special case.



The linker in the Class Loader Subsystem will try to find the best match.

Functions can be generic.

Classes can also be generic.

Important for collections!

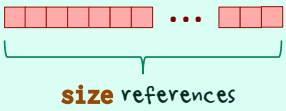
This idea of turning a class into a kind of parameter can also be applied to classes; this is important for collections, which are "container classes". For instance, an ArrayList is a class that can hold (contain) objects of any class.

Collections

Let's move to collections by reviewing the different ways we have to group objects together.

Arrays of Objects

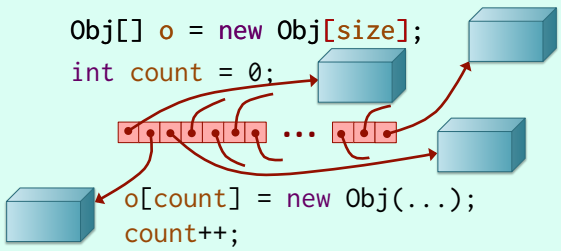
```
Obj[] o = new Obj[size];
int count = 0;
```



When you create an array of objects, each element isn't an object (contrary to arrays of base types). It's just a place to store a reference.

Arrays of Objects

```
Obj[] o = new Obj[size];
int count = 0;
```



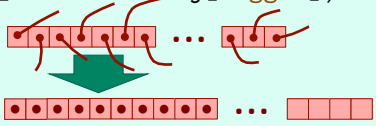
```
o[count] = new Obj(...);
count++;
```

Each object is created one by one.

Arrays of Objects

And then?

```
Obj[] o2 = new Obj[bigger];
```



```
o = o2;
o2 = null;
```

If you exceed the size of the array, you can create a bigger one and copy the references there.

I have just mentioned boxind and, I'd like to come back to the topic. Because of the close link between base type and shadow class, it's often performed automatically.

Boxing

`int` → `Integer`

Performed automatically
(autoboxing)

... in one direction as in the other one.

Unboxing

int  Integer

Performed automatically
(auto-unboxing)

problem with arrays

If one needs not to worry much about simple variables, it's unfortunately a different story with arrays, as the two following examples prove.

```
public class Boxing1 {

   public static void displayVal(Integer val) {
       System.out.println(val);
   }

   public static void main(String[] args) {
       int n = 10;

       displayVal(n);
   }
}
```

If I'm calling a method that takes an Integer parameter with an int parameter, everything works fine.

Works OK

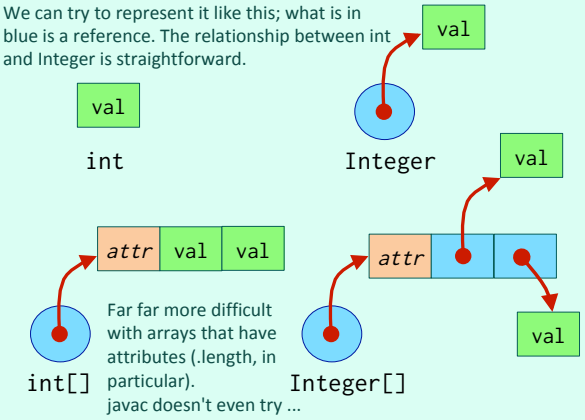
```
public class Boxing2 {

   public static void displayVal(Integer[] valarr) {
       for (int i = 0; i < valarr.length; i++) {
         System.out.println(valarr[i]);
       }
   }

   public static void main(String[] args) {
       int[] n = new int[3];

       for (int i = 0; i < 3; i++) {
          n[i] = i;
       }
       displayVal(n);
   }

}
```

What happens with an array of ints passed where an array of Integers is expected?

```
$ javac Boxing2.java
Boxing2.java:15: error: incompatible types: int[]
cannot be converted to Integer[]
       displayVal(n);
                  ^
Note: Some messages have been simplified; recompile
with -Xdiags:verbose to get full output
1 error
$
```

It fails. The reason of the problem is that boxing/unboxing know how to convert between base type and object, but not between object and object ... and an array is an object (it's created by calling new). We have "lost" the base type.

We can try to represent it like this; what is in blue is a reference. The relationship between int and Integer is straightforward.



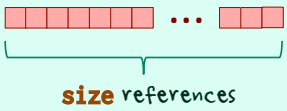
Far far more difficult with arrays that have attributes (.length, in particular). javac doesn't even try ...

Collections

With this in mind, we can return to Collections that, once again, are, in what is called the "Java Collection Framework", objects, and not base types.

Arrays of Objects

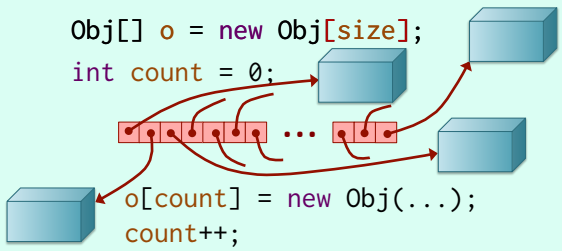
```
Obj[] o = new Obj[size];
int count = 0;
```



The simplest group of objects we can think of is an array of objects. Note that initially it's just an array of empty references.

Arrays of Objects

```
Obj[] o = new Obj[size];
int count = 0;
```



```
o[count] = new Obj(...);
count++;
```

Each object added to the array must be separately created ("instantiated"), which is very different from arrays of base types.

Searching arrays

Arrays, even arrays of objects, are relatively easy to search.

```
public class City {
    private String  name;
    private String  country;
    private int     population;

    public City(String n, String c, int p) {
       name = n;
       country = c;
       population = p;
    }

    public String toString() {
       return name + "(" + country + ") - "
                   + Integer.toString(population);
    }
}
```

Suppose that we have an array of City objects.

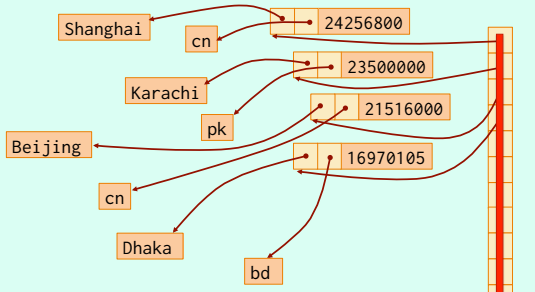
```
#Name,Country,Population
Shanghai,cn,24256800
Karachi,pk,23500000
Beijing,cn,21516000
Dhaka,bd,16970105
Delhi,in,16787941
Lagos,ng,16060303
Istanbul,tr,14025000
Tokyo,jp,13513734
Guangzhou,cn,13080500
...
```

Create an array

```
City[] cities = new City[200];
```

Open the file and read into the array

It's very common to load data in memory from external datafiles or from databases.



When we load everything in memory, every object (including Strings) becomes a reference to the actual values. To search the array, we'll just start at index 0 in the array and increase the index.

```
public class City {
      private String  name;
      private String  country;
      private int     population;
      ...
      public boolean isNamed(String name) {
          return this.name.equals(name);
      }
}
int i = 0;
while (!cities[i].isNamed("...")) {
  i++;
}
```

ONLY if we know that each name occurs only once (unique)

Time will be proportional to the number of cities.

Check boundaries!

O(n)

The bigger, the slower.

We can do better with arrays

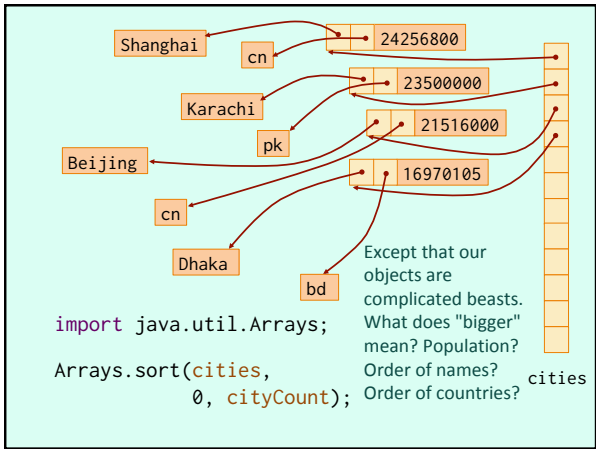
BINARY SEARCH

We can do far better with an array by running a binary search, which is the kind of search you run when looking for a word in a dictionary: you open in the middle, check a word there, and search either the first half or the second half (I hope you recognized recursion).

There is just one prerequisite: you wouldn't be able to search a dictionary (otherwise than reading every page) if words were not ordered into it. A binary search can only work if the array is sorted. Class Arrays implements static methods that do that efficiently.

First, the array must be sorted.

```
import java.util.Arrays;

Arrays.sort(cities,
            0, cityCount);
```




```
import java.util.Arrays;

Arrays.sort(cities,
            0, cityCount);
```

Except that our objects are complicated beasts. What does "bigger" mean? Population? Order of names? Order of countries?

Arrays.sort() needs to know. In fact, it will require the existence of a method called compareTo() that it will call for organizing objects in the correct order.

You cannot sort if you cannot

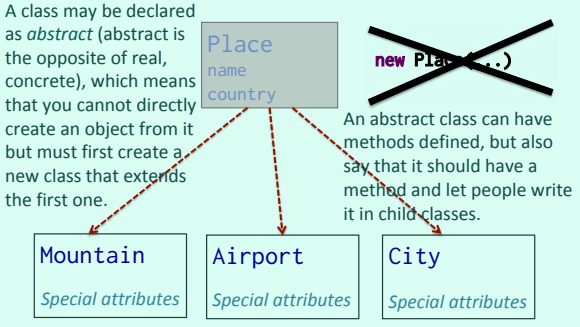
COMPARE

Reminder: interface

When a specially named function has to exist in a class, it's said that the class must *implement* an interface. Let's review what an interface is.

A class can be declared abstract

A class may be declared as *abstract* (abstract is the opposite of real, concrete), which means that you cannot directly create an object from it but must first create a new class that extends the first one.



An abstract class can have methods defined, but also say that it should have a method and let people write it in child classes.

Interfaces – lightweight classes

abstract *(implicit)*

no variable attribute

define methods that classes MUST implement to conform

constants OK

Interfaces are special lightweight classes that mostly define a behavior, through methods. If a class says that it implements an interface, then it must have the methods defined in the interface. Note that some interfaces require no method (saying that a method implements these interfaces is just an indication for javac that will generate some special code)

Arrays.sort() only works if the class implements the Comparable interface, which requires a compareTo() method. Here we say that comparing cities means comparing their names.

Need for the **Comparable** interface.

```
public class City implements Comparable {
      private String  name;
      private String  country;
      private int     population;
      ...
      public int compareTo(Object o) {
          City other = (City)o;
          return this.name.compareTo(other.name);
      }
      public int compareTo(String name) {
          return this.name.compareTo(name);
      }
}
```

Required by Arrays.sort()

I'll need this

Find New York

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverpool
6	London
7	New York
8	Reykjavik
9	Rio de Janeiro
10	Shanghai
11	Tokyo

12 elements

Middle = index 5

Search 6 to 11

New middle = index 8

Search 6 to 7

New middle = index 6

Search 7 to 7

A binary search works by dividing by two at each step the size of the part of the array that is searched.

Number of comparisons

Size of the array is N

Sequential search: N / 2

If we double the number of items , we DOUBLE the number of comparisons.

Binary search: 2*log_2(N - 1)

If we double the number of items , we add ONE more comparison.

O(logn)

It's a very efficient algorithm.

```
static City binarySearch(City[] arr,
                         int    elements,
                         String lookedFor) {
    // Assume that the array is sorted
    int     start = 0;
    int     end = elements - 1;
    int     mid = 0;
    int     comp;
    boolean found = false;

    while (start <= end) {
      mid = (start + end) / 2;
      comp =  arr[mid].compareTo(lookedFor);
      if (comp < 0) {
         // Array element smaller
         start = mid + 1;
      } else if (comp > 0) {
         // Array element bigger
         end = mid - 1;
      } else {
         // Found
         found = true;
         start = end + 1;  // To stop the loop
      }
    }
    if (found) {
      return arr[mid];
    } else {
      return null;
    }
}
```

This is how it can be written in Java.

Or we can do it recursively

Trivial case? 0 or 1

You can also try to write it recursively, although this is a case where recursion doesn't make the code much simpler.

Class Arrays contains several (static) binarySearch() methods. You don't need to write it ...

In practice these classic algorithms are part of the Java standard methods.

What can we say about arrays of objects?

Arrays of object are very convenient for some operations, and much less for others.

Arrays of Objects

Not too good when data is very dynamic

Search efficient only when sorted

Keeping order is hard if inserted randomly

When you keep adding/removing items anywhere in the array, they are hard to manage (how do you manage "holes" in the array?). If they can be searched very efficently, it only works if they are sorted. If you want to insert values randomly and keep the order, you must move bytes around quite a lot.

Arrays of Objects

The index isn't always a natural way to access data

Additionally, the way you refer to an element in an array is its index (its position in the array). When you refer to a city, it's more natural to give the name of the city than to say "city at position n". In a program, that means that you would probably have the program user enter a name, search the array to find the position, then use the position afterwards.

Java Collections

Classes and interfaces in **`java.util`**

No predefined size

Suitable for special usages

You need to know what is important for your application

The Java Collections define interfaces and classes that allow you to group objects otherwise than in simple arrays. There are different ways to group objects, all with different features, and you need to choose the one that is right for what you want to do.

ArrayLists of Objects

grows automatically

... but it remains costly

(byte shifting still occurs!)

The ArrayList that you have probably already used (if not, you may have used a Vector that is very much like an ArrayList) is such a collection. It grows automatically (which means that it automatically creates a bigger array when needed and copy the elements there – you need not do it yourself)

ArrayList

You don't use exactly the same syntax with an ArrayList as with an array, but you can basically do the same operations. Some methods apply to one element, others to all of them.

Array that resizes automatically

```
import java.util.ArrayList;

ArrayList al = new ArrayList();
```

al[i] doesn't work!

Main methods:

```
al.add(e);
e = al.get(i);
n = al.size();
al.remove(i);    al.removeAll();
```

ArrayList

Behaviour

"List" in the name refers to the methods.

Array that resizes automatically

```
import java.util.ArrayList;

ArrayList al = new ArrayList();
```

Interface

Main methods:

```
al.add(e);
e = al.get(i);
n = al.size();
al.remove(i);    al.removeAll();
```

ArrayList

Implementation

"Array" refers to how data is physically stored. You can often offer the same operations, but store data in very different ways.

Array that resizes automatically

```
import java.util.ArrayList;

ArrayList al = new ArrayList();
```

Main methods:

```
al.add(e);
e = al.get(i);
n = al.size();
al.remove(i);    al.removeAll();
```

```java
import java.util.ArrayList;
import java.util.Random;

public class ArrList {

    public static void main(String[] args) {
       ArrayList al = new ArrayList();
       Random    r = new Random();

       al.add(42);
       al.add("A character string");
       al.add("Война и мир");
       al.add(3.141592);
       al.add('试');

       int target = r.nextInt(al.size());
       al.remove(target);
       for (int i = 0; i < al.size(); i++) {
         System.out.println("Data @" + i + " = " + al.get(i));
       }
    }
}
```

An ArrayList, like any collection, can store objects of any type, even of different types as here.

I'm removing a random element.

```
$ javac ArrList.java
Note: ArrList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$
```

Note that mixing very different objects is a poor programming practice, and the javac compiler isn't too happy with it. You can neverthe less run the program.

```
$ javac ArrList.java
Note: ArrList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java ArrList
Data @0 = 42
Data @1 = A character string
Data @2 = 3.141592
Data @3 = 试
$
```

Indices were renumbered

Russian text is gone

My random deletion removed the Russian text. You see here that the array was rearranged and that everything is nicely displayed. But once again, a collection of different types of objects (you may have noticed autoboxing in action) is a bad idea. In a collection, the type of all items should be the same.

What if we need to deal with lists of different data types with specific methods?

What makes javac unhappy, though, isn't so much that there are different types of objects, but that in fact it doesn't know what type of object is used; javac may have the same issue with ArrayLists in which all the elements are the same type.

Option 1

```
import java.util.ArrayList;

public class MyStringList {

   private ArrayList _list;

   public void setElement(String s) {
      _list.add(s);
   }

   public String getElement(int i) {
      return (String)_list.get(i);
   }
}
```

If I wrap the ArrayList methods into methods that ensure that the Object type is the same …

Option 1

javac not happy:

```
$ javac MyStringList.java
Note: MyStringList.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
$
```

… javac is still unhappy. After all, I could overload my methods with methods adding objects of a different type to the same ArrayList.

Option 2

But all objects derive from the Object class.

I could take advantage that every class extends Object, and use only Object parameters (if you remember, I did something like this when implementing the Comparable interface in the example I gave earlier).

Option 2

```
import java.util.ArrayList;

public class MyList {

   private ArrayList _list;

   public void setElement(Object o) {
      _list.add(o);
   }

   public Object getElement(int i) {
      return _list.get(i);
   }
}
```

Just take what you are given and stop complaining.

Option 2

javac not happier:

```
$ javac MyList.java
Note: MyList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$
```

It still doesn't work because javac is at heart a control freak.

Option 3

Of course I could specify ArrayLists of Strings, or ArrayLists of Integers. Then it would make difficult to reuse software (one new version by object type)

Hand-crafting special, typed ArrayLists

Software reuse?

Option 4

Defining templates for javac

GENERICS

Collections are definitely where you want to use generics.

Option 4

```
import java.util.ArrayList;

public class MyGenericList<T> {

   private ArrayList<T> _list;

   public void setElement(T o) {
      _list.add(o);
   }

   public T getElement(int i) {
      return _list.get(i);
   }
}
```

If you need to have your special wrapper around a collection, you use a generic type that you pass down to the collection.

Option 4

```
$ javac MyGenericList.java
$
```

javac is happy!

Then javac can control the full chain of operations, and make sure that you aren't misusing classes.

```
MyGenericList<String>   nameList;
MyGenericList<Float>    distanceList;
```

All collections expect to be typed

```
ArrayList<City> cities = new ArrayList<City>();
```

And as you have seen they may be typed with a generic type.

Usual naming conventions

<E>	Element (collection)
<K> and <V>	Key and Value (maps)
<N>	Number
<T>	Type

<S>, <U>, <V> and so forth 2nd, 3rd, nth type

There are some common conventions in the naming of generic types. "T" is very common, but in a collection you often use "E" instead and if you use an object to retrieve another object (we are going to see this soon), you usually call the first one "K" and the second one "V".

A generic type is a parameter of a sort. You can also sometimes pass a parameter to the parameter ... rather advanced usage.

```
public class City implements Comparable {
      private String  name;
      private String  country;
      private int     population;
      ...
      public int compareTo(Object o) {
          City other = (City)o;
          return this.name.compareTo(other.name);
      }
}
```

Unsafe

My implementation of Comparable was rather poor. I can make it better by specifying what I compare with generics.

```
public class City implements Comparable<City> {
      private String  name;
      private String  country;
      private int     population;
      ...
      public int compareTo(City other) {
          return this.name.compareTo(other.name);
      }
}
```

Now javac can check!

My compareTo() method must match a specialized Comparable interface.

Remember, Generics only work with OBJECTS (references)

~~ArrayList<int> intArray = new ArrayList<int>();~~

```
ArrayList<Integer> intArray = new ArrayList<Integer>();
```

Once again, it only works with Objects, not base types, and collections are an area where autoboxing/unboxing doesn't really work.

How to sort an ArrayList?

~~import java.util.Arrays;~~

~~Arrays.sort(cities,~~
~~0, cityCount);~~

```
import java.util.Collections;

Collections.sort(cities);
```

Even if an ArrayList is an array behind the scene, it's not a plain array and the methods from the Arrays class no longer work. You must instead use methods from the Collections class.

Collections is a kind of dummy class (like Arrays) that only contains static methods. Other than sorting methods, it provides methods for turning a collection into another type of collection when possible, and so forth.

Collections

Class with static methods for:

- sorting
- converting between collections
- Synchronizing

This refers to multithreading, a topic we'll see later. Ignore it for now.

You can pass to sort() a **Comparator** as second parameter (advanced!)

```
Collections.sort(MyList,
                 new Comparator<T>() {
       public int compare(T o1, T o2){
              // Logic goes here
                 }});
```

You can define how you compare your objects in the call to the sort() method if your objects are really fanciful. It's simpler, and better in my opinion, to implement Comparable in your object class (the fact that you can do something doesn't mean that you should).

ArrayLists

When we compare ArrayLists to regular Arrays, we have seen that it's better when data is dynamic (grows automatically, the ArrayList was rearranged when we removed an element) but otherwise it's very much the same

- Better when data is dynamic
- Search efficient only when sorted
- **Keeping order is hard** if inserted randomly
- Index is **not a natural way** to reference an object.

ArrayLists

Implementation

Interface

What are the possible interfaces?

Time to take a look at other collections, and before anything let's check which interfaces are provided.

List

Keeps track of order

Can contain several times the same value

The characteristic of a lists is that it usually keep track of the order of operations (.add() is the usual way of adding an element, and it's usually appended to the end). Another important feature is that you can add the same value several times to a list.

Closely related to lists, you have queues and deques. Queue is a French word (mispronounced in English, sounds a bit like 피 in French) which means "tail" and also "waiting line". Deque was invented by English-speakers to mean a reverse operation ("de" in Latin means to undo. It sometimes appears as "dis" in English, eg mount/dismount, honour/dishonour).

Queue/Deque

Keeps track of chronology

Can contain several times the same value

FIFO (Queue)
FIFO/LIFO (Deque)

The main difference with plain Lists is that Queues and Deques are primarily designed for FIFO and LIFO operations. What?

FIFO is shorthand for "First In First Out" which is what happens when you wait in line for a service almost anywhere. Inside computers you also have service providers (writing to a disk is a service, connecting to another machine is a service) that apply these rules.

First In First Out

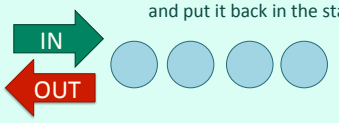
"First come, first served"

LIFO is shorthand for "Last In First Out" and refers to a stack – that's usually how you take plates from a stack of plates. This strategy is much used inside computers too – you have a stack for function calls in memory (remember recursion). If a computer evaluates a numerical expression, such as

((7+3) * 6)/ 5

it uses a stack too. It reads symbols, stacks them, and when it find a closing parenthesis it takes off the stack everything up to and including the opening parenthesis, computes the result, and put it back in the stack if the stack isn't empty.

Last In First Out



Queue/Deque

Dynamic processing

As you can see, Queues and Deques assume that objects are flying in and out (when in a regular List they could simply be stored there). These interfaces are optimized for this type of processing.

Set

Each element appears only once

Ordered or not

Another big category are Sets. The peculiarity of Sets is that you cannot have the same element twice in a Set. Strictly forbidden. In practice, it also means that Sets must be fast to search, because otherwise it would be painfully slow in a big Set to check if the new element that you are trying to add is already there or not.

not a true Collection

Map Index is **not a natural way** to reference an object.

Key	Value

Key is unique

Can be ordered by key

but the purpose is mostly getting what is associated with one key

Finally you have Maps, which aren't collections (strictly speaking) but close relatives. Maps associate two objects. One, the key, must appear only once in the Map. The other can appear several times or only once.

Collection interfaces

List

Queue/Deque

Set

Map

So, as a summary, this is the main interfaces you have. If you need to associate a "Key" (often a String) and a "Value" (for instance a city name and a City object), you'll use a Map. If you want to ensure that you store a City only once and don't want to store the city name separately from the City object, it's more a Set that you want. Queues and Deques are for active systems, and a List is a kind of general-purpose interface.

ArrayLists

We have talked about interfaces, let's talk about implementation.

Implementation

What are the possible implementations?

Resizable Array

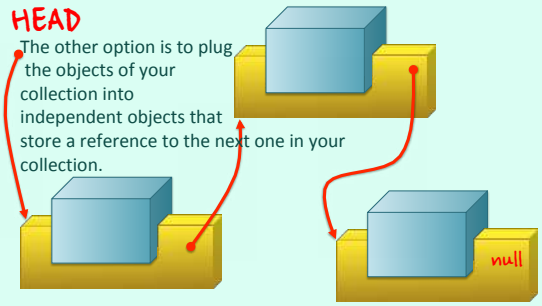
When we talk about an array in a collection context, we really mean a resizable array. The basic characteristic of an array is that its elements (references for an array of objects) are in one block of memory. It has advantages, but also when you resize the array ypu must find a bigger free area and copy all the references there.

contiguous memory (one chunk of memory)

really needed for a list?

HEAD

The other option is to plug the objects of your collection into independent objects that store a reference to the next one in your collection.



Linked List

Several variants exist. When you grow the list, it's little by little.

LinkedList

Methods, of course, come from the interface, and so you'll find basically the same ones with LinkedLists as with ArrayLists.

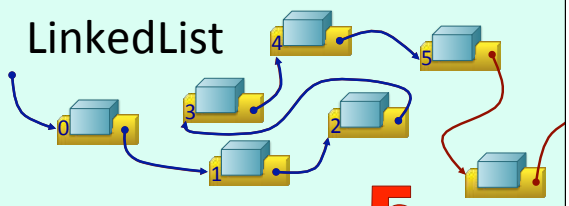
Independent elements

```
import java.util.LinkedList;

LinkedList<T> ll = new LinkedList<T>();
```

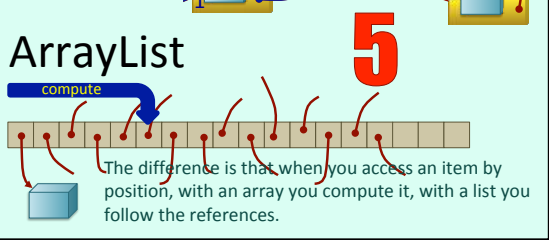
Main methods:

```
al.add(e);
e = al.get(i);
n = al.size();
al.remove(i);    al.removeAll();
```

LinkedList

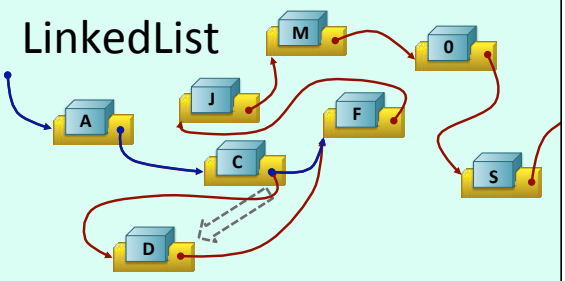


ArrayList



The difference is that when you access an item by position, with an array you compute it, with a list you follow the references.

LinkedList



What is the benefit?

On the other hand, you can add and remove an item by just changing a few references, not reorganizing and shifting everything as you would in an array.

LinkedList

Specific methods:

```
ll.addLast(e);
ll.addFirst(e);
e = ll.pollLast();
e = ll.pollFirst();
e_arr = ll.toArray();
```

Retrieve and remove

LinkedLists also implement a few specific methods, including the possibility of turning the list into an array.

LinkedList

DON'T use an index, use an **Iterator**

Iterum = Again

If you can loop on an index with an array, with lists it should be a very bad idea as finding the n^{th} element means starting from the beginning and counting. Instead, you use a special object called Iterator, which takes you from one element to next. Forget about binary searches.

LinkedList

Iterators are simple to use. Don't confuse `hasNext()`, which just tests if we have reached the end or not, and `next()` that moves forward.

DON'T use an index, use an **Iterator**

```
import java.util.ListIterator;

ListIterator li = ll.listIterator();

while (li.hasNext()) {
   System.out.println(li.next());
}
```

You still see iterators a lot (old style), but you can also write:
`for (<Object> o: <Collection of Objects>) { ... }`

LinkedLists

Good when data is dynamic

Inefficient search

Good for keeping order when inserted randomly

Use iterators rather than indices

This summarizes the strong and weak points of LinkedLists.

NEXT TIME

* More Collections, Collection examples (book Chapter 20) and things not in the text book (mostly because introduced in Java recently)