CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn

Resizable Array

Linked List

Hash Table

We have seen last time resizable arrays (ArrayLists) and Linked Lists, which globally behave the same but that implementation makes more efficient than the other in some cases.

The next implementation is an interesting one. Instead of adding items to the collection as they come, the idea is to compute a number (called a hash code) for each object, and to derive the storage location from this number. All Java classes extend the Object class, that has a method called hashcode() returning an int.

```
public static void main(String[] args) {
     ArrayList<City> cities = new ArrayList<City>();

     int cityCount = load(cities);
     System.out.println(Integer.toString(cityCount)
                       + " cities loaded");
     Collections.sort(cities);
     for (int i = 0; i < cities.size(); i++) {
       System.out.println(cities.get(i) + "\thash = "
         + Integer.toString(cities.get(i).hashCode()));
     }
}
```

We can write a small program to display hash codes for elements in a list of cities.

```
$ java HashCode
161 cities loaded
Abidjan(ci) - 4765000   hash = 2018699554
Addis Ababa(et) - 3103673   hash = 1311053135
Adelaide(au) - 1316779  hash = 118352462
Ahmedabad(in) - 5570585 hash = 1550089733
Alexandria(eg) - 4616625   hash = 865113938
Ankara(tr) - 5271000 hash = 1442407170
Auckland(nz) - 1495000  hash = 1028566121
Baghdad(iq) - 7180889   hash = 1118140819
Bandung(id) - 2575478   hash = 1975012498
Bangkok(th) - 8280925   hash = 1808253012
Barcelona(es) - 1604555 hash = 589431969
Beijing(cn) - 21516000  hash = 1252169911
Bengaluru(in) - 8425970 hash = 2101973421
Berlin(de) - 3517424 hash = 685325104
Bogotá(co) - 7878783 hash = 460141958
...
```

Randomly distributed

What you can see is that these numbers look random

% numslots

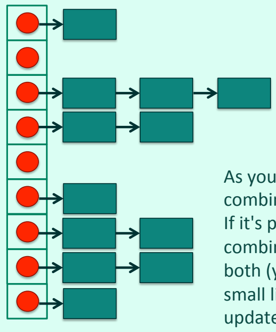
We can take a modulo to store each object in one particular position of an array.

PROBLEM

Different items can hash into the same value.

Conflict

Make each array slot a linked list of items hashing to the same value.



As you can see, a hash table is a combination of array and list. If it's properly designed, it combines the advantages of both (you find quickly each small list, that can be easily updated)

HashMap

(Key, Value) pairs

A HashMap uses this kind of structure to store pairs of objects. The hashcode() value for the key is used for finding the value.

```
import java.util.HashMap;

HashMap<K,V> hm = new HashMap<K,V>();
```

Main methods:

```
hm.put(k, v);
v = hm.get(k);
n = hm.size();
hm.keySet()
hm.remove(k);    hm.clear();
```

Number of (key, value) pairs

Map Iterator

Hashmaps aren't really designed for accessing the whole collection – more to access objects one by one. However, as keys are unique, all keys together fit the Set requirement and can be returned as a Set. Then you can get an Iterator on the set, fetch keys one by one and retrieve associated values.

```
import java.util.Iterator;
import java.util.Set;
import java.util.Map;

Set set = hm.entrySet();
Iterator it = set.iterator();
while (it.hasNext()) {
   Map.Entry en = (Map.Entry)it.next();
   System.out.println("key: "
          + en.getKey() + ", value: "
          + en.getValue());
}
```

Good example of relationship between collections.

~~Map Iterator~~

```
import java.util.Set;
import java.util.Map;

Set set = hm.entrySet();
for (Map.Entry en: set) {
   System.out.println("key: "
          + en.getKey() + ", value: "
          + en.getValue());
}
```

All collections these days also allow accessing their elements with a for loop that uses an implicit iterator.

HashMaps

Very good when data is dynamic

Very efficient search

No order, no chronology

means time information ("time study" in Greek)

The weak spot of hashmaps is that, as location depends only on value, you have no idea (unless you store time in objects) of when you added objects. Contrast with lists, where recent additions are usually at one end.

HashSet，使用HashMap實現的Set

HashSets

= Set implemented with HashMaps

As keys occur only once, a Hashmap can also be ued to implement a Set.

linkedhashmap返回的對象與插入的順序相同，這點與HashMap不同

There is a LinkedHashMap class which returns objects in the order of insertion when you iterate. It's not the case with a HashMap.

You can also find some special classes for special needs. Check the docs.

Resizable Array

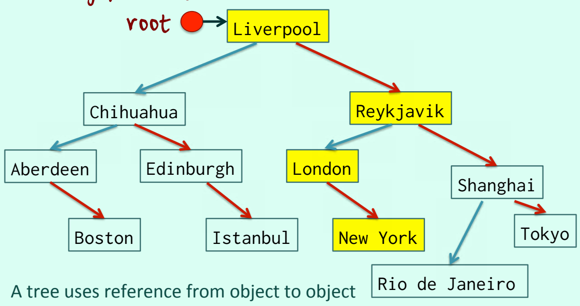
Linked List

Hash Table (+ Linked List)

Tree

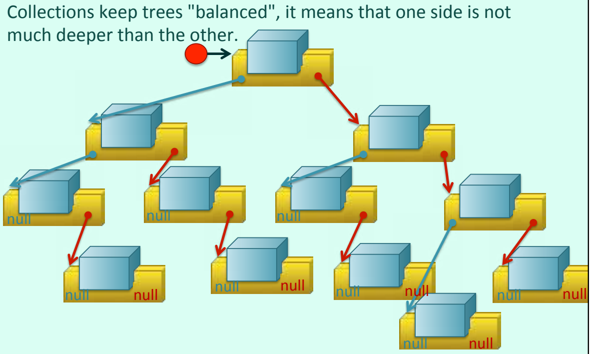
Finally, the last main implementation is trees.

Looking for "New York"



A tree uses reference from object to object as a list, but each item (usually called a node or a leaf) is linked to more than one other item. You navigate one branch or the other depending on how the value you are looking for compares to the one in the node. As efficient as a binary search.

Collections keep trees "balanced", it means that one side is not much deeper than the other.

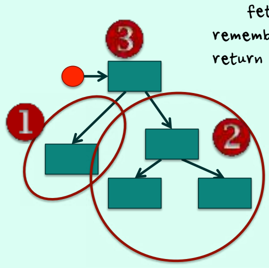


In reality there are more than two descendants, but it's the idea

Strongly ordered (Keys implement **`Comparable`**)

Iterators work recursively behind the scene

If there is a left-hand tree
    fetch next from left-hand tree
remember right-hand reference for next time
return current



Trees combine the efficient searches of hash tables with the strong ordering of linked lists.

TreeMaps

You also have TreeSets (unique values)

Very good when data is dynamic

Efficient search

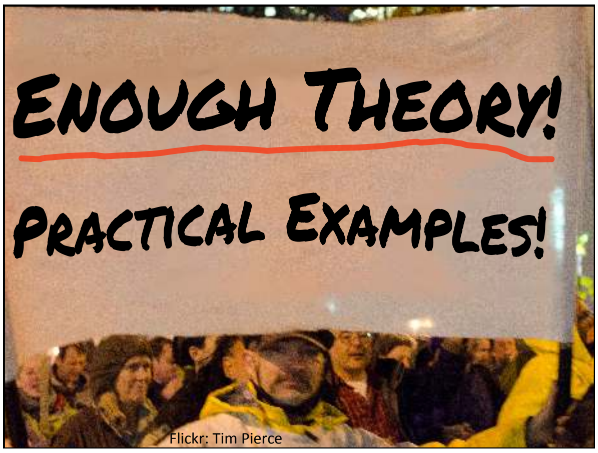
Ordered by construct

No chronology

Like hash tables, trees don't keep track of time of insertion.

Resizable Array	**List**
Linked List	**Queue/Deque**
Hash Table	**Set**
Tree	**Map**

Because of the requirements of interfaces, and the limits of implementations, some combinations work very well, and others not at all. Because you lack the time information in a tree or hash table, it makes no sense to implement a Queue/Deque with them. However, arrays and lists are very good for that. When it comes to maps and sets, it's search performance that matters and there it's the opposite – arrays and lists don't really work, hash tables and trees are excellent. It's your requirements and the methods you'll need that dictate your choice of a collection (there are often several possibilities).



One interesting (and very useful) example of hash tables is the Properties class, which associates two strings.

`java.util.Properties`

```
Hashtable<T,T>
      |
Properties
```

```
Key:   String
Value: String
```

```
# Location of data files
data_dir = C:\Users\Public\Data

# Remote server
server = 192.168.1.214

# Theme name
theme = Funky
```

preferences.cnf

Properties

When you install a piece of software on your computer, you are usually asked a lot of questions, such as where you want to install the program, the location of other resources, possibly a theme. All this information is stored in a .ini, .conf or whatever file. Each parameter is a name associated with a value. Properties objects deal with these files, can read them, write them, and ignore for instance lines starting with #.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

public class PropertiesExample {

    public static void main(String Args[]) {
        Properties defprop = new Properties();
        defprop.put("data_dir", ".");
        defprop.put("theme", "classic");
        Properties prop = new Properties(defprop);
        try (BufferedReader conf
             = new BufferedReader(new FileReader("preferences.cnf"))) {
          prop.load(conf);
        } catch (IOException e) { // Ignore
           System.err.println("Warning: using default preferences");
        }
        // Display the preferences
        System.out.println(prop.getProperty("data_dir"));
        System.out.println(prop.getProperty("theme"));
    }
}
```

"try with resources" (remember?)

You can define default values. Calling prop.get() would return null for a value not read from the file, prop.getProperty() returns the default if there is one.

Practical Example #2

The second example uses a Tokenizer class that returns words from a text file one by one, ignoring space and punctuation. Goal: finding the 10 most used words in a speech. We need to associate with each word a counter (so, we need a Map<String,Integer>). If we don't know the word, we store it wih "1". Otherwise, we retrieve the counter, increase its value by one, and store it back. When we have counted words, we need to find the 10 most used – we need a map (associating a number of occurrences to a list of words, as there may be ties) but we also need some ordering. We need to go through our hash map and store its objects in, for instance, a TreeMap<Integer,TreeSet<String>>. Then we can iterate on the tree map and retrieve the most common words.

Practical Example #2

The result is usually very disappointing, because in an English speech the most common words are likely to be "the", "is", "a", and so forth. Those words are not very significant words and are usually called "stop words" (search engines on the Internet ignore them).
What we need to do is have a list of stop words, read it into an easily searchable structure such as a tree, and start counting words only when we cannot find them in this list of not important words. It gives a completely different vision of a speech.

A sample program (and a few speeches) has been uploaded to Sakai (under `Resources/Sample Programs`).

Java Goodies

What I'll present now are very interesting features from Java, which are mostly absent from the textbook, for mostly two reasons:
- or they appeared less than 10 years ago, when the book was written
- or they have taken an importance not suspected 10 years ago.

Annotations

The first feature is annotations. You may have noticed some annotations already; it's common when you use inheritance and you redefine in the child class a method defined in the parent class to precede the child class definition with
`@Override`
(which means *replace* the existing method).
This is an annotation, which is completely optional but warns javac of your intent. If you mistype the name, it will enable javac to detect an error if there is no such method in the parent class.

Annotations = Tags

Completely optional

Change nothing to what the program does

Help Javac – or the program

Much used by code-generating tools

As annotations can be accessed by programs, many tools that generate code – for tests, for instance – use annotations to collect information they cannot get otherwise.

Marker **`@Override`**

Single parameter

Multiple parameters

Annotations can take different forms, from the simple marker to some kinds of function calls that are outside the program itself.

METADATA

= DATA about the CODE

Metadata is a big concern in real life. Companies consider programs as assets, on which several generations of developers can work, which must be written in an easy-to-comprehend, standard way, and well documented. Metadata allows, among many other things, to industrialize code production and to standardize everything.

3 standard annotations

Java provides three standard annotations, which are all a way to give hints to javac.

@Override

@Deprecated = Obsolete

@SuppressWarnings (*warnings to suppress*)

For instance
`@SuppressWarnings({"deprecation", "unchecked"})`
`javac -X` gives the list of warnings, associated to `-Xlint`

2 annotations added in Java 7 and 8

@SafeVarargs

@FunctionalInterface

"Varargs" stands for "Variable [number of] Arguments"
We'll talk soon about what is a functional interface ...

You can create your own annotations!

Declared as interfaces

```
import java.lang.annotation.*;

public @interface MyAnnotation {
}
```

Annotation-based tools use their own set of annotations, which you just need to import before using.

They can have methods but:

Methods should not have any parameters.

Methods declarations should not have any **throws** clauses.

As annotations are a bit special (it's a kind of program in the program) they are constrained by a number of rules.

They can have methods but:

Methods should not have any parameters.

Methods declarations should not have any **throws** clauses.

or array of these types

Return type must be one of:

primitive type String enum Class

boolean int char float double ...

May provide structured documentation

```
class SomeClass {
      //  Created by S Faroult
      //  Creation date: 21/03/2017
      //  Revision history:
      //   24/05/2017 – Constructor
      //        with String parameter
      //   26/02/2018 – toString() rewritten

}
```

What can you use annotations for in practice? Any Software Development Manager dreams of seeing comments like this. But every developer will not write them, and those who do may use a different format.

May provide structured documentation

```
import java.lang.annotation.*;

public @interface ClassDoc {
      String   author();
      String   created();
      String[] revisions();
  }
```

methods

ClassDoc.java

Annotations may help turning readable but unparsable comments into data usable by a program.

May provide structured documentation

```
@ClassDoc(author="S Faroult",
         created="21/03/2017",
         revisions={"24/05/2017 – Constructor
 with String parameter",
         "26/02/2018 – toString() rewritten"})
class SomeClass {

}
```

Because the annotation is defined and checked by javac, you can ensure a standard way of documenting code. This information can then be retrieved (we'll see how soon) to document programs.

Meta Annotations

5 other annotations about annotations

@Retention Says whether the annotation is available to javac, or available at runtime.

@Documented Make it appear in docs generated by the javadoc tool

@Target What it applies to: Constructor, Method, Parameter ...

@Inherited Passed to child classes (false by default)

@Repeatable Can be applied more than once

JUNIT generates tests for checking your programs. Frameworks are software tools that try to generate automatically the boring bits of a program (which are often a lot of copy-and-paste).

Much used by tools

JUNIT

Frameworks

We'll see them later.

Reflection

I have said that annotations can be accessed by program, "reflection" is how to do it if your annotation was prefixed by `@Retention(RetentionPolicy.RUNTIME)`

Generally speaking, "reflection" is your program asking the JVM what it knows about it – and the JVM knows a lot of things.

As all this happens of course while the program is running, it allows for a lot of on-the-fly operations that would be impossible with a compiled program written in C, for instance. Reflection is considered rather advanced programming, but some of its features are frequently used, for instance with JDBC which is the standard Java way to access a database and which will see in some detail in a few weeks.

Reflection

examine or modify the runtime behavior

Reflection

Works because of the JVM

Once again, it only works because of the JVM. The loading subsystem needs to read a lot of information to make the program runnable, and this information is stored and made available when the program runs.

stores in memory the description of classes when it loads them

Reflection

The JVM stores objects (of class `Class`) that describe every class used in the application.

Works because of the JVM

class called **Class**

objects represent classes in the running application

no constructor – built by the JVM

Reflection

There are two ways two retrieve class information from the JVM.

```
ClassName obj = new ClassName();
```

```
obj.getClass()
```
method inherited from Object

1. The getClass() method of an object.

```
ClassName.class
```
"static" version

2. The .class attribute when there is no object.

no constructor – built by the JVM

Reflection

```
class OuterClass {
   private int dummy;
   OuterClass(){}
}
public class MyClass {
   class InnerClass {
       private int dummy;
       InnerClass(){}
   }
   public static void main(String[] args) {
       OuterClass obj = new OuterClass();
       System.out.println(obj.getClass().getName());
       System.out.println(InnerClass.class.getName());
   }
}
```

```
$ java MyClass
OuterClass
MyClass$InnerClass
$
```

For instance, you can retrieve class names.

Reflection

There are many useful uses for reflection. One common problem is locating files used by your program – the properties file to start with if there is one.

A few useful examples

1 Location of files read by your program

- parameter file
- data file
- multimedia, and so forth

When people click on an icon to launch your program, the idea of "current directory" becomes extremely hazy. If you want to start by reading a properties file, of if you want to display the logo of your company (an image) while initialization is going on, where should you look?
The default directory for installing programs varies from system to system (and don't forget that a Java application can run on Windows as well as on Linux or Mac OSX), and additionally users often have the option of installing software elsewhere than the default location. Your only hope to find out is to get it when the program runs.

Reflection

As the loader knows where it got the .class from, you can just ask the JVM.

Solution Get location at runtime

```
public class Reflection {

   public static void main(String[] args) {
       System.out.println(Reflection
                           .class
                           .getClassLoader()
                           .getResource("Reflection.class")
                           .toString());
   }
}
```

有问题，获取不到，红框中的代码需要删掉

file:/Users/... ... .../Reflection.class

Reflection

And if you know the hierarchy of the files you need, you can easily derive the location of any file you supplied.

Solution Get location at runtime

package1 package2 resources

images

```
URL url = this
.getClass()
.getClassLoader()
.getResource("resources/images/myCat.png");
```

myCat.png

Reflection

I have mentioned that annotations could be read by a program, it's through reflection

A few useful examples

2 Reading annotations

Done by many tools
(we'll see some of them later)

Reflection

There is a condition: the annotation must be available at runtime.

A few useful examples

2 Reading annotations

@Retention(RetentionPolicy.RUNTIME)

By default annotations are **NOT** made available at runtime

Remember that @Retention() is a meta-annotation, an annotation that applies to annotations.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface ClassDoc {
      String   author();
      String   created();
      String[] revisions();
  }
```

ClassDoc.java

If SomeClass is annotated with an annotation available at runtime ...

```
@ClassDoc(author="S Faroult",
         created="21/03/2017",
         revisions={"24/05/2017 – Constructor
 with String parameter",
         "26/02/2018 – toString() rewritten"})
class SomeClass {



}
```

Must be recompiled if ClassDoc is changed

... then getAnnotations() gets it.

```
import java.lang.annotation.Annotation;
public class ReadingAnnotations {

  public static void main(String[] args) {
    Annotation[] annotations = SomeClass
                               .class
                               .getAnnotations();

    for (Annotation annot: annotations) {
       System.out.println(annot.toString());
    }
  }
}
```

```
$ java ReadingAnnotations
@ClassDoc(author=S Faroult, created=21/03/2017,
revisions=[24/05/2018 - Constructor with String
parameter, 26/02/2018 - toString() rewritten])
$
```

Reflection

There is another very important use of reflection.

A few useful examples

3 Dynamically loading a class

Much used for "drivers"

Because of the multiplication of standards, identical functionality is often achieved by different classes, that work with one special piece of hardware or software.

database_system1.jar

database_system2.jar

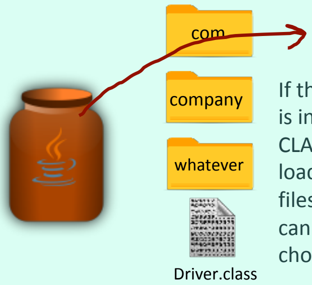
database_system3.jar

This is particularly useful with database access. Although there is a common language for accessing databases, database providers supply (as java archives) classes that implement the required methods to talk to THEIR system.

Usually the driver has a long complicated name to ensure that there is no conflict (two different drivers cannot have the same name).

Fully qualified class name

`com.company.whatever.Driver`



in CLASSPATH!

If the name of the .jar file is included in the CLASSPATH (where the loader looks for .class files), then the program can load the driver of its choice.

```
Class c = Class.forName("com.company.whatever.Driver");
Driver drv = (Driver)c.newInstance();
```

For instance ...

There is a Java graphical tool called Squirrel SQL that uses this to let you query almost any database system, as long as you have the suitable .jar file added to your CLASSPATH. You can switch between very different systems.

Lambda expressions

Our third important topic after annotations and reflection are "Lambda expressions", which were introduced in Java 8 (first released in March 2014). "Lambda expressions" touch on what is called "functional programming", an area which has been recently the object of much interest, even if its roots are more than 100 years old. You'll probably hear about "lambda expressions" and "functional programming" elsewhere than in a Java context.

Nested Classes

```
class OuterClass {

    ...
    class NestedClass {
        ...
    }
}
```

To explain the benefits of lambda expressions, let's take a look back at classes and interface, and start with nested classes, classes defined inside other classes.

```
    class OuterClass {
        private int attr;
 public
private ...
protected class NestedClass {
            ...
        }
    }
```

YES

If a nested class is declared as public, private or protected it can access the private attributes of the outer class.

```
class OuterClass {
    private int attr;
    ...
static class NestedClass {
        ...
    }
}
```

NO

This no longer works if it's defined as static, because the attribute only exists when an OuterClass object is created, but NestedClass is accessible without an object.

Depending on the nested class being static or not, you have two different ways to create a nested class object.

```
OuterClass.NestedClass nestedObject =
                   outerObject.new NestedClass();
```

depends on an existing OuterClass object

```
OuterClass.StaticNestedClass nestedObject =
               new OuterClass.StaticNestedClass();
```

independent from any OuterClass object

WHY NESTING?

Grouping

Encapsulation

You can of course question why classes should be nested. This is mostly done as a way of structuring the code, either by grouping software components or for hiding through encapsulation the inner working.

Local Classes

```
class OuterClass {

    ...
    public void doSomething() {
       class LocalClass {
           ...
       }
    }
}
```

You can also have local classes, that are not only defined inside another class, but inside a method.

In the area of Java software engineering, there is also one component that is very much used: interfaces. Interfaces define the behaviour, and how you can "talk" to an object (remember that object oriented programming is mostly about objects exchanging messages).
If a class can only extend (inheritance) one parent class, it can implement multiple interfaces. Java Collections are a rather good example.

Reminder: Interfaces

- abstract *(implicit)*
- define methods that classes MUST implement to conform
- no variable attribute
- constants OK

```
class SomeClass extends ParentClass {

}
```

methods inherited, unless they are abstract

methods must be rewritten

```
class SomeClass implements Interface {

}
```

The only problem with interfaces is that YOU have to rewrite the methods (fortunately one interface rarely defines many methods)

Anonymous Classes

There are many cases when the only things that we are interested in are interface methods. We can of course define a class implementing the interface ...

```
class NamedClass implements Interface {
   ...
}

Interface anObject = new NamedClass(...);
```

... but as the only thing we really want is an "interface object reference", the named class is a bit useless. One such example is a "Comparator" object. We usually just want the compareTo() method.

Anonymous Classes

Java allows defining an unnamed (ano – nymous = *without a name* in Greek) object that implements all that is required by the interface.

```
Interface anObject = new Interface() {
                        // attribute and method definitions
                    };
```

Very convenient for parameters

Anonymous Classes

```
class NamedClass extends ParentClass {
   ...
}

NamedClass anObject = new NamedClass(...);
```

This works not only with interfaces, but also with inheritance. Children objects can be named ...

Anonymous Classes

```
ParentClass anObject = new ParentClass() {
                    // attribute and method definitions
                   };
```

... or not, if the only thing you are really interested in is a special behaviour of an abstract parent class.