

CSE 143

Lecture 14

Maps/Sets; Grammars

reading: 11.2 - 11.3

slides adapted from Marty Stepp and Hélène Martin

<http://www.cs.washington.edu/143/>

Exercise

- Write a method `crawl` accepts a `File` parameter and prints information about that file.
 - If the `File` object represents a normal file, just print its name.
 - If the `File` object represents a directory, print its name and information about every file/directory inside it, indented.

```
cse143
  handouts
    syllabus.doc
    lecture_schedule.xls
  homework
    1-sortedintlist
      ArrayIntList.java
      SortedIntList.java
      index.html
      style.css
```

- **recursive data:** A directory can contain other directories.

File objects

- A `File` object (from the `java.io` package) represents a file or directory on the disk.

Constructor/method	Description
<code>File(String)</code>	creates <code>File</code> object representing file with given name
<code>canRead()</code>	returns whether file is able to be read
<code>delete()</code>	removes file from disk
<code>exists()</code>	whether this file exists on disk
<code>getName()</code>	returns file's name
<code>isDirectory()</code>	returns whether this object represents a directory
<code>length()</code>	returns number of bytes in file
<code>listFiles()</code>	returns a <code>File[]</code> representing files in this directory
<code>renameTo(File)</code>	changes name of file

Public/private pairs

- We cannot vary the indentation without an extra parameter:

```
public static void crawl(File f, String indent) {
```

- Often the parameters we need for our recursion do not match those the client will want to pass.

In these cases, we instead write a pair of methods:

- 1) a public, non-recursive one with the parameters the client wants
- 2) a private, recursive one with the parameters we really need

Exercise solution 2

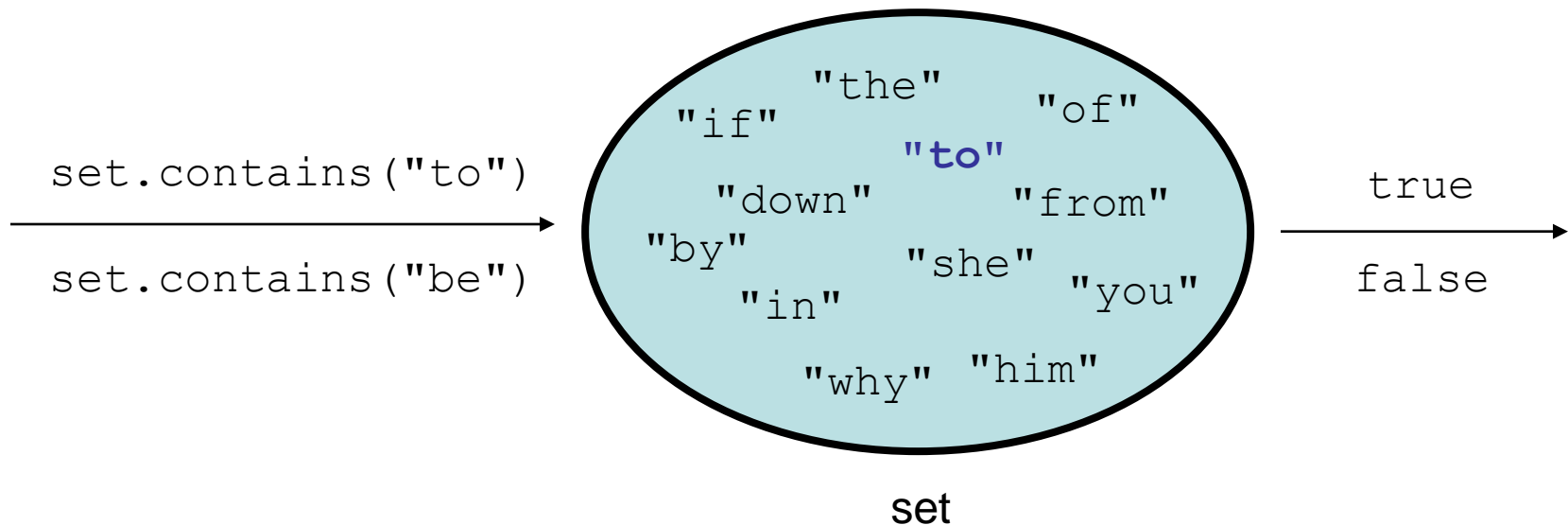
```
// Prints information about this file,  
// and (if it is a directory) any files inside it.  
public static void crawl(File f) {  
    crawl(f, "");    // call private recursive helper  
}  
  
// Recursive helper to implement crawl/indent behavior.  
private static void crawl(File f, String indent) {  
    System.out.println(indent + f.getName());  
    if (f.isDirectory()) {  
        // recursive case; print contained files/dirs  
        for (File subFile : f.listFiles()) {  
            crawl(subFile, indent + "    ");  
        }  
    }  
}
```

Exercise

- Write a program that counts the number of unique words in a large text file (say, *Moby Dick* or the King James Bible).
 - Store the words in a collection and report the # of unique words.
 - Once you've created this collection, allow the user to search it to see whether various words appear in the text file.
- What collection is appropriate for this problem?

Sets (11.2)

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
 - add, remove, search (contains)
 - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



Set implementation

- in Java, sets are represented by `Set` type in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
 - `HashSet`: implemented using a "hash table" array;
very fast: **$O(1)$** for all operations
elements are stored in unpredictable order
 - `TreeSet`: implemented using a "binary search tree";
pretty fast: **$O(\log N)$** for all operations
elements are stored in sorted order
 - `LinkedHashSet`: **$O(1)$** but stores in order of insertion;
slightly slower than `HashSet` because of extra info stored

Set methods

```
List<String> list = new ArrayList<String>();  
...  
Set<Integer> set = new TreeSet<Integer>(); // empty  
Set<String> set2 = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

<code>add (value)</code>	adds the given value to the set
<code>contains (value)</code>	returns <code>true</code> if the given value is found in this set
<code>remove (value)</code>	removes the given value from the set
<code>clear ()</code>	removes all elements of the set
<code>size ()</code>	returns the number of elements in list
<code>isEmpty ()</code>	returns <code>true</code> if the set's size is 0
<code>toString ()</code>	returns a string such as "[3, 42, -7, 15]"

The "for each" loop (7.1)

```
for (type name : collection) {  
    statements;  
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, `array`, or other collection

```
Set<Double> grades = new HashSet<Double>();  
...
```

```
for (double grade : grades) {  
    System.out.println("Student's grade: " + grade);  
}
```

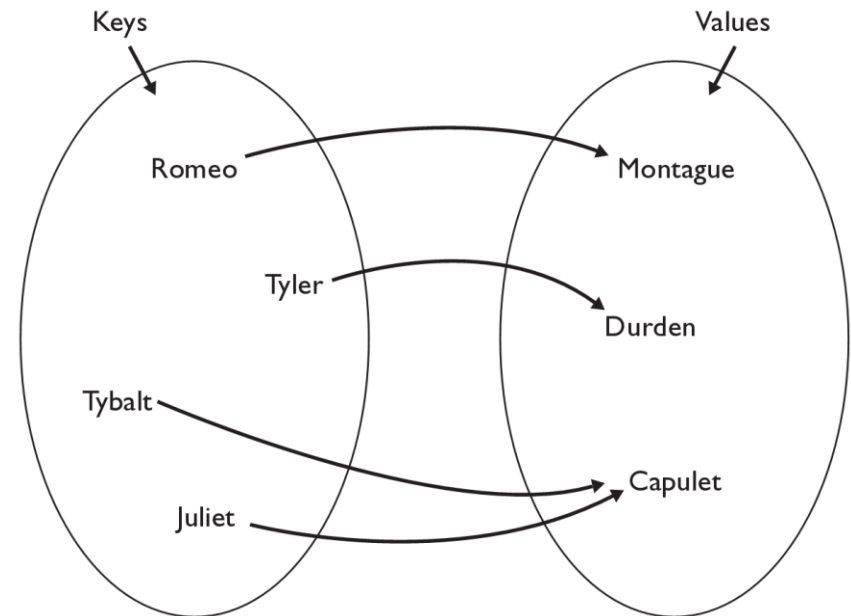
– needed because sets have no indexes; can't get element `i`

Exercise

- Write a program to count the number of occurrences of each unique word in a large text file (e.g. *Moby Dick*).
 - Allow the user to type a word and report how many times that word appeared in the book.
 - Report all words that appeared in the book at least 500 times, in alphabetical order.
- What collection is appropriate for this problem?

Maps (11.3)

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
 - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
 - **put**(*key*, *value*): Adds a mapping from a key to a value.
 - **get**(*key*): Retrieves the value mapped to the key.
 - **remove**(*key*): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns "Capulet"

Map implementation

- in Java, maps are represented by `Map` type in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
 - `HashMap`: implemented using an array called a "hash table"; extremely fast: **$O(1)$** ; keys are stored in unpredictable order
 - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **$O(\log N)$** ; keys are stored in sorted order
 - `LinkedHashMap`: $O(1)$; keys are stored in order of insertion
- A map requires 2 type params: one for keys, one for values.

```
// maps from String keys to Integer values
```

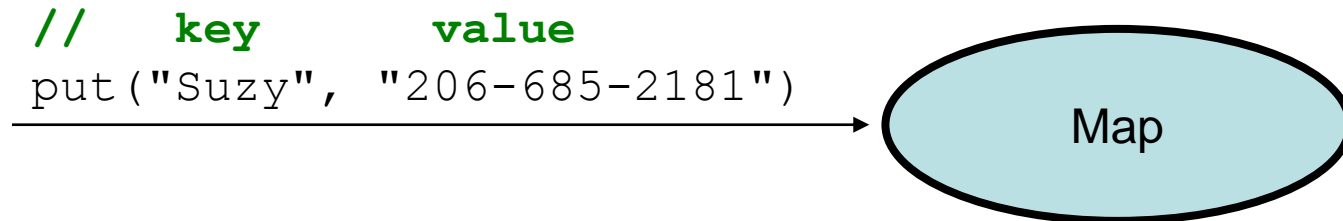
```
Map<String, Integer> votes = new HashMap<String, Integer>();
```

Map methods

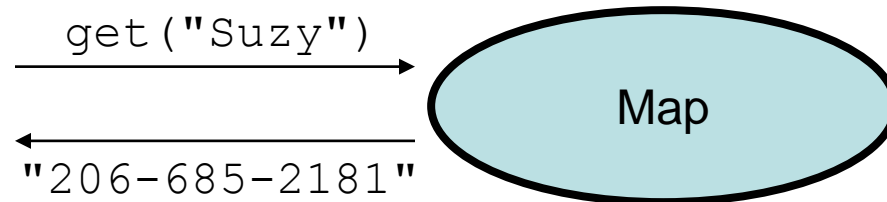
<code>put(key, value)</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>get(key)</code>	returns the value mapped to the given key (<code>null</code> if not found)
<code>containsKey(key)</code>	returns <code>true</code> if the map contains a mapping for the given key
<code>remove(key)</code>	removes any existing mapping for the given key
<code>clear()</code>	removes all key/value pairs from the map
<code>size()</code>	returns the number of key/value pairs in the map
<code>isEmpty()</code>	returns <code>true</code> if the map's size is 0
<code>toString()</code>	returns a string such as " <code>{a=90, d=60, c=70}</code> "
<code>keySet()</code>	returns a set of all keys in the map
<code>values()</code>	returns a collection of all values in the map
<code>putAll(map)</code>	adds all key/value pairs from the given map to this map
<code>equals(map)</code>	returns <code>true</code> if given map has the same mappings as this one

Using maps

- A map allows you to get from one half of a pair to the other.
 - Remembers one piece of information about every index (key).



- Later, we can supply only the key and get back the related value:
Allows us to ask: *What is Suzy's phone number?*



Maps and tallying

- a map can be thought of as generalization of a tallying array
 - the "index" (key) doesn't have to be an `int`

– count digits: 22092310907

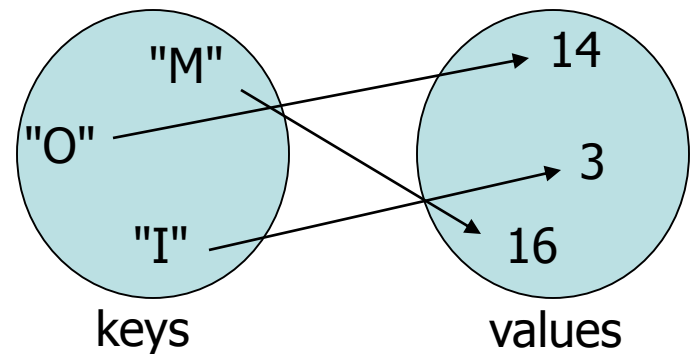
index 0 1 2 3 4 5 6 7 8 9

value	3	1	3	0	0	0	0	1	0	2
-------	---	---	---	---	---	---	---	---	---	---

// (M)cCain, (O)bama, (I)ndependent

– count votes: "MOOOOOOMMMMMMOOOOOOOMOMMI MOMMI MOMMI O"

key	"M"	"O"	"I"
value	16	14	3



keySet and values

- `keySet` method returns a `Set` of all keys in the map
 - can loop over the keys in a `foreach` loop
 - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) { // Geneva -> 2
    int age = ages.get(name); // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
 - can loop over the values in a `foreach` loop
 - no easy way to get from a value to its associated key(s)

Languages and Grammars

Languages and grammars

- (formal) **language**: A set of words or symbols.
- **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
 - describes language *syntax* (rules) but not *semantics* (meaning)
 - can be used to generate strings from a language, or to determine whether a given string belongs to a given language

Backus-Naur (BNF)

- **Backus-Naur Form (BNF):** A syntax for describing language grammars in terms of transformation *rules*, of the form:

<symbol> ::= <expression> | <expression> ... | <expression>

- **terminal:** A fundamental symbol of the language.
- **non-terminal:** A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.
- developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

An example BNF grammar

`<s> ::= <n> <v>`

`<n> ::= Marty | Victoria | Stuart | Jessica`

`<v> ::= cried | slept | belched`

- Some sentences that could be generated from this grammar:

Marty slept

Jessica belched

Stuart cried

BNF grammar version 2

```
<s> ::= <np> <v>  
<np> ::= <pn> | <dp> <n>  
<pn> ::= Marty | Victoria | Stuart | Jessica  
<dp> ::= a | the  
<n> ::= ball | hamster | carrot | computer  
<v> ::= cried | slept | belched
```

- Some sentences that could be generated from this grammar:

```
the carrot cried  
Jessica belched  
a computer slept
```

BNF grammar version 3

```
<s> ::= <np> <v>
<np> ::= <pn> | <dp> <adj> <n>
<pn> ::= Marty | Victoria | Stuart | Jessica
<dp> ::= a | the
<adj> ::= silly | invisible | loud | romantic
<n> ::= ball | hamster | carrot | computer
<v> ::= cried | slept | belched
```

- Some sentences that could be generated from this grammar:

```
the invisible carrot cried
Jessica belched
a computer slept
a romantic ball belched
```

Grammars and recursion

```
<s> ::= <np> <v>
<np> ::= <pn> | <dp> <adjp> <n>
<pn> ::= Marty | Victoria | Stuart | Jessica
<dp> ::= a | the
<adjp> ::= <adj> <adjp> | <adj>
<adj> ::= silly | invisible | loud | romantic
<n> ::= ball | hamster | carrot | computer
<v> ::= cried | slept | belched
```

- Grammar rules can be defined *recursively*, so that the expansion of a symbol can contain that same symbol.
 - There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.

Grammar, final version

```
<s> ::= <np> <vp>
<np> ::= <dp> <adjp> <n> | <pn>
<dp> ::= the | a
<adjp> ::= <adj> | <adj> <adjp>
<adj> ::= big | fat | green | wonderful | faulty | subliminal
<n> ::= dog | cat | man | university | father | mother | child
<pn> ::= John | Jane | Sally | Spot | Fred | Elmo
<vp> ::= <tv> <np> | <iv>
<tv> ::= hit | honored | kissed | helped
<iv> ::= died | collapsed | laughed | wept
```

- Could this grammar generate the following sentences?

Fred honored the green wonderful child

big Jane wept the fat man fat

- Generate a random sentence using this grammar.

Sentence generation

