Grammars, Sets, and Maps
Outline

1 Languages and Grammars

2 Sets

3 Foreach Loops

4 Maps
**Definition (Formal Language)**

A **Formal Language** is a set of words or symbols.

For example:

\{1, 2, 3, 4, 5\} is a language, and \{hello, goodbye\} is a language.

**Definition (Grammar)**

A **Grammar** is a set of rules that *generates* a particular language.

Grammars are used to:

- **generate** strings, and to
- **check** if strings are in the language
Definition (Backus-Naur Form (BNF))

**BNF** is a syntax for describing language grammars in terms of transformation rules, of the form:

\[
\langle symbol \rangle ::= \langle expression \rangle | \langle expression \rangle | ... | \langle expression \rangle
\]

BNF is made up of two types of symbols:

- **Terminals**: Literals (symbols that are interpreted literally)
- **Non-terminals**: A symbol describing how to generate other symbols based on the rules of the grammar
An Example Grammar

Example Grammar

\[ \text{object} ::= \text{article} \ \text{thing} \]
\[ \text{article} ::= \text{The} \ | \ \text{A} \ | \ \text{That} \ | \ \text{This} \]
\[ \text{thing} ::= \text{ball} \ | \ \text{index card} \ | \ \text{word} \ | \ \text{balloon} \]

To generate \text{object}s from this grammar, we do the following steps:

1. Start at \text{object} and look at what to transform to:
   \[ \text{article} \ \text{thing} \]

2. For each non-terminal, look at its rule and choose an option.

Some \text{object}s in this grammar:

- The ball
- That index card
- The balloon
Count the Number of **Distinct** Words in a Text

Write a program that counts the number of unique words in a large text file (say, “Alice in Wonderland”). The program should:

- Store the words in a collection and report the number of unique words in the text file.
- Allow the user to search it to see whether various words appear in the text file.

What collection is appropriate for this problem?

**We could use an ArrayList...**

We’d really like a data structure that takes care of duplicates for us.
A **set** is an *unordered* collection of *unique* values. You can do the following with a set:

- Add **element** to the set
- Remove **element** from the set
- Is **element** in the set?

**How To Think About Sets**

Think of a set as a bag with objects in it. You’re allowed to pull things out of the bag, but someone might shake the bag and re-order the items.

**Example Set**

```
"such strings"
"much wow"
"goodbye"
"very hello"
```

Is “goodbye” in the set? **true**
Is “doge” in the set? **false**
Set is an **interface** in `java.util`; implementations of that interface are:

**HashSet**
- $\mathcal{O}(1)$ for all operations.
- **Does not** maintain a useful ordering

**TreeSet**
- $\mathcal{O}(\log(n))$ for all operations
- **Does** maintain the elements in **sorted order**
## Set Reference

### Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new HashSet&lt;E&gt;()</code></td>
<td>Creates a new <code>HashSet</code> of type <code>E</code> that initially has no elements</td>
</tr>
<tr>
<td><code>new HashSet&lt;E&gt;(collection)</code></td>
<td>Creates a new <code>HashSet</code> of type <code>E</code> that initially has all the elements in <code>collection</code></td>
</tr>
<tr>
<td><code>new TreeSet&lt;E&gt;()</code></td>
<td>Creates a new <code>TreeSet</code> of type <code>E</code> that initially has no elements</td>
</tr>
<tr>
<td><code>new TreeSet&lt;E&gt;(collection)</code></td>
<td>Creates a new <code>TreeSet</code> of type <code>E</code> that initially has all the elements in <code>collection</code></td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(val)</code></td>
<td>Adds <code>val</code> to the set</td>
</tr>
<tr>
<td><code>contains(val)</code></td>
<td>Returns true if <code>val</code> is a member of the set</td>
</tr>
<tr>
<td><code>remove(val)</code></td>
<td>Removes <code>val</code> from the set</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all elements from the set</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the number of elements in the set</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns true whenever the set contains no elements</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representation of the set such as <code>[3, 42, -7, 15]</code></td>
</tr>
</tbody>
</table>
How can we list all the elements of a set?

- We can’t do a normal for loop, because **there are no indexes**
- We also don’t know what is actually in the set...

**Solution**

The solution is a new type of loop called the *foreach* loop.

```
Set<Integer> set = new HashSet<Integer>();
set.add(5);
set.add(5);
set.add(5);
set.add(5);
set.add(10);
set.add(10);
set.add(12);
for (int i : set) {
    System.out.println(i);
}
// The set remains unchanged.
```

**OUTPUT**

```
10
5
12
```
In general, foreach loops look like the following:

```java
for (type var : collection) {
    // do something with var
}
```

You can use them for many other collections like Lists. You are **not allowed** to use them for Stacks or Queues.

### Another Example of foreach Loops

```java
List<String> list = new ArrayList<String>();
list.add("a");
list.add("a");
list.add("b");
list.add("d");
String everything = "";
for (String s : list) {
    everything += s;
}
System.out.println(everything);
```

**OUTPUT**

```
>> aabd
```
The following is the performance of various data structures at removing duplicates from a large dictionary of words.
Note that despite it looking like HashSet and TreeSet have the same runtime on the previous slide, they do not.
Count the Number of **Occurrences** of Each Word in a Text

Write a program that counts the number of unique words in a large text file (say, “Alice in Wonderland”). The program should:

- 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?

We could use something sort of like `LetterInventory`, but we don’t know what the words are in advance...

We’d really like a data structure that relates tallies with words.
What is a Map?

Definition (Map)

A **map** is a data structure that relates **keys** and **values**. You can do the following with a map:

- Ask what **value** a particular **key** maps to.
- Change what **value** a particular **key** maps to.
- Remove whatever the relation is for a given **key**.

How To Think About Maps

- Maps are a lot like functions you’ve seen in math: \( f(x) = x^2 \) maps 0 to 0, 2 to 4, ... 
- Your **keys** are identifiers for values. Ex: social security numbers (maps SSN \( \rightarrow \) person).
- Safe-deposit boxes are another useful analogy. You get a literal key to access your belongings. If you know what the key is, you can always get whatever you’re keeping safe.

**Example Map**

- How many characters is “much wow”? **8**
- What does “goodbye” map to? **7**
- What is the value for ”such strings”? **12**
Map is an **interface** in `java.util`; implementations of that interface are:

**HashMap**
- $O(1)$ for all operations.
- **Does not** maintain a useful ordering of anything

**TreeMap**
- $O(\log(n))$ for all operations
- **Does** maintain the **keys** in **sorted order**
Creating A Map

To create a map, you must specify **two** types:
- What type are the keys?
- What type are the values?

They *can* be the same, but they aren’t always.

Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new HashMap&lt;K,V&gt;()</code></td>
<td>Creates a new <code>HashMap</code> with keys of type <code>K</code> and values of type <code>V</code> that initially has no elements</td>
</tr>
<tr>
<td><code>new TreeMap&lt;K,V&gt;()</code></td>
<td>Creates a new <code>TreeMap</code> with keys of type <code>K</code> and values of type <code>V</code> that initially has no elements</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong><code>put(key, val)</code></strong></td>
<td>Adds a mapping from <code>key</code> to <code>val</code>; if <code>key</code> already maps to a value, that mapping is replaced with <code>val</code></td>
</tr>
<tr>
<td><strong><code>get(key)</code></strong></td>
<td>Returns the value mapped to by the given <code>key</code> or <code>null</code> if there is no such mapping in the map</td>
</tr>
<tr>
<td><strong><code>containsKey(key)</code></strong></td>
<td>Returns true the map contains a mapping for <code>key</code></td>
</tr>
<tr>
<td><strong><code>remove(key)</code></strong></td>
<td>Removes any existing mapping for <code>key</code> from the map</td>
</tr>
<tr>
<td><strong><code>clear()</code></strong></td>
<td>Removes all key/value pairs from the map</td>
</tr>
<tr>
<td><strong><code>size()</code></strong></td>
<td>Returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><strong><code>isEmpty()</code></strong></td>
<td>Returns <code>true</code> whenever the map contains no mappings</td>
</tr>
<tr>
<td><strong><code>toString()</code></strong></td>
<td>Returns a string repr. of the map such as <code>{d=90, a=60}</code></td>
</tr>
<tr>
<td><strong><code>keySet()</code></strong></td>
<td>Returns a set of all keys in the map</td>
</tr>
<tr>
<td><strong><code>values()</code></strong></td>
<td>Returns a collection of all values in the map</td>
</tr>
<tr>
<td><strong><code>putAll(map)</code></strong></td>
<td>Adds all key/value pairs from the given map to this map</td>
</tr>
<tr>
<td><strong><code>equals(map)</code></strong></td>
<td>Returns true if given <code>map</code> has the same mappings as this</td>
</tr>
</tbody>
</table>
Each map can **answer one type of question**. For example:

If the keys are phone numbers and the values are people

Then, the map can answer questions of the form:

“Who does this phone number belong to?”

```java
Map<String,String> people = new HashMap<String,String>();
people.put("(206) 616–0034", "Adam’s Office");
people.get("(206) 616–0034"); // Returns "Adam’s Office"
```

The people map can **only go in one direction**. If we want the other direction, we need a different map:

If the keys are people and the values are phone numbers

Then, the map can answer questions of the form:

“What is this person’s phone number?”

```java
Map<String,String> phoneNumbers = new HashMap<String,String>();
phoneNumbers.put("Adam’s Office", "(206) 616–0034");
phoneNumbers.get("Adam’s Office"); // Returns "(206) 616–0034"
```
Earlier, we had an example where
- keys were “phrases”
- values were “# of chars in the key”

That map can answer the question:

“How many characters are in this string?”

```java
Map<String, Integer> numChars = new HashMap<String, Integer>();
numChars.put("very hello", 10);
numChars.put("goodbye", 7);
numChars.put("such strings", 12);
numChars.put("much wow", 8);
numChars.get("much wow"); // Returns 8
```
There **is no good way** to go from a **value** to its **key** using a map. But we can go from **each key** to the values:

```java
Map<String, Double> ages = new TreeMap<String, Double>();
// These are all according to the internet... a very reliable source!
ages.put("Bigfoot", 100);
ages.put("Loch Ness Monster", 3.50);
ages.put("Chupacabra", 20); // ages.keySet() returns Set<String>
ages.put("Yeti", 40000);
for (String cryptid : ages.keySet()) {
    double age = ages.get(cryptid);
    System.out.println(cryptids + " -> " + age);
}
```

**OUTPUT**

```
>> Chupacabra -> 20
>> Loch Ness Monster -> 1500
>> Bigfoot -> 100
>> Yeti -> 40000
```
You can get a collection of all the values:

```java
Map<String, Double> ages = new TreeMap<String, Double>();

// These are all according to the internet...a very reliable source!
ages.put("Bigfoot", 100);
ages.put("Loch Ness Monster", 3.50);
ages.put("Chupacabra", 20); // ages.keySet() returns Set<String>
ages.put("Yeti", 40000);

for (int age : ages.values()) {
    System.out.println("One of the cryptids is aged " + age);
}
```

OUTPUT

```
>> One of the cryptids is aged 1500
>> One of the cryptids is aged 40000
>> One of the cryptids is aged 20
>> One of the cryptids is aged 100
```
Some Grammar/Set/Map Tips!

- BNF is another recursive structure!

- Sets and Maps are two more collections each with their own places

- Sets are for storing data **uniquely**

- Maps are for storing **relationships** between data; they only work in **one direction**

- `foreach` loops are a great tool for looping through collections

- You should know the syntax for `foreach` loops and that Hash and Tree are types of sets and maps