# CSE 143
# Lecture 11

Maps

Grammars

slides created by Alyssa Harding

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

# Example: `studentGrades`

- Let's pretend it's midterm time and all the TAs are tired of grading
- We decide to randomly generate grades for all our students!

```
// generate random grade between 0 and 99
// so that no one aces the test
Random r = new Random();
int grade = r.nextInt(100);
```

- …I promise this won't really happen

# Example: `studentGrades`

- But this gets tiring too
- We don't want to hand generate a grade for each student
- If we have a list of all of our students, we could write a program to loop through them and assign them grades

```
List<String> students =
                new ArrayList<String>();
students.add("Joe");
students.add("Sally");
students.add("Mike");
```

# Example: `studentGrades`

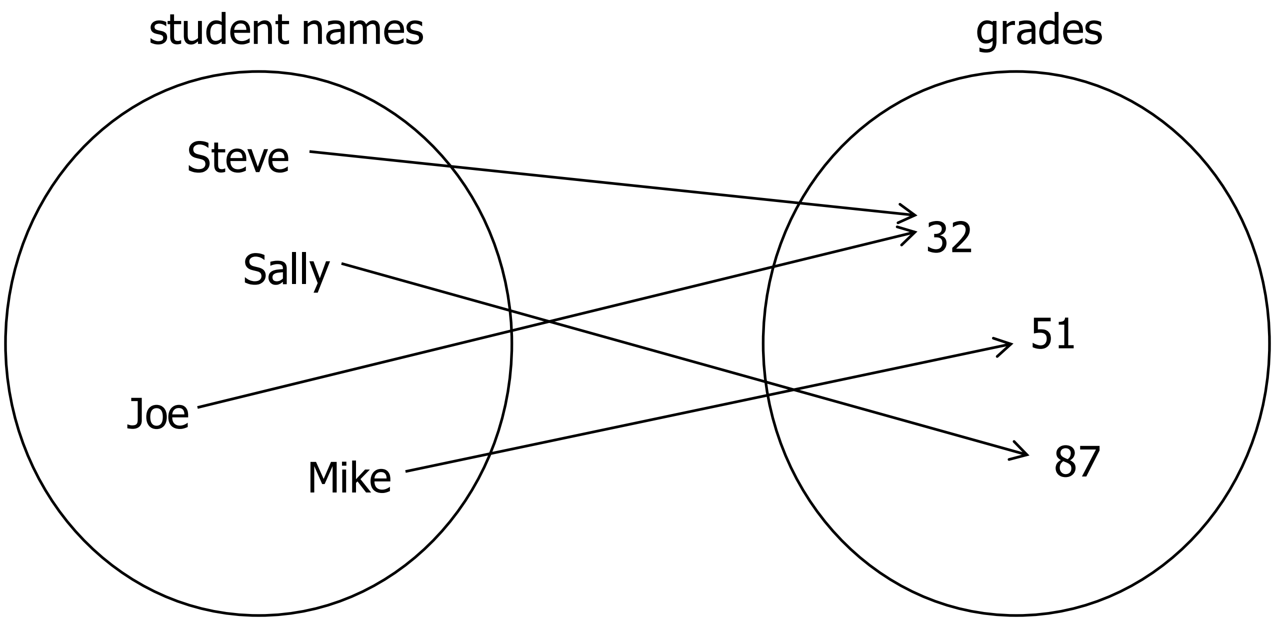
- But we need a way to keep track of which grade goes with which student

| | 0 | 1 | 2 |
|---|---|---|---|
| students | Joe | Sally | Mike |
| grades | 32 | 87 | 51 |

- We could keep another list of all the grades
  - Student at index 0 has grade at index 0, and so on
  - But that's tedious

# Maps

- Solution: maps allow us to associate key/value pairs
  - For example, a student name with a grade




- Also known as a dictionary, associative array, hash
  - Can think of it as an array where you can have indexes of any type of `Object` instead of just `ints`

# Maps

- Java's **`Map<K,V>`** interface that uses generic key/value types

```
// adds a mapping from the given key to the given value
void put(K key, V value)

// returns the value mapped to the given key (null if none)
V get(K key)

// returns true if the map contains a mapping for the given key
boolean containsKey(K key)

// returns a Set of all keys in the map
Set<K> keySet()

// removes any existing mapping for the given key
remove(K key)
```

# Maps

- We will use two implementations of the **`Map`** interface:

- **`TreeMap`**:
  - provides O(log(n)) access to elements
  - stores keys in sorted order

- **`HashMap`**:
  - provides O(1) access to elements
  - stores keys in unpredictable order

- The **`SortedMap`** interface is also implemented by **`TreeMap`**

# Example: `studentGrades`

- Using this, can solve our problem of grading by making a map:

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

- And storing the grades in it:

```
Random r = new Random();
for (String name : students) {
    int grade = r.nextInt(100);
    studentGrades.put(name, grade);
}
```

# Example: `studentGrades`

- How can we see the grades?

- We can get a **`Set`** of all the keys
  - we don't know anything about a **`Set`**
  - but it's **`Iterable`** so we can use a foreach loop

```
for (String name : studentGrades.keySet() )  {
   System.out.println(name + " " +
                       studentGrades.get(name));
}
```

# Example: `wordCount`

- Let's try a tougher problem now
- Given some text file, we want to count how many times each word occurs

```
// open the file
Scanner console = new Scanner(System.in);
System.out.print("What is the name of the text file?
");
String fileName = console.nextLine();
Scanner input = new Scanner(new File(fileName));
```

# Example: `wordCount`

- Make a **`SortedMap`** to hold the words and their counts:

```
SortedMap<String, Integer> wordCounts =
                    new TreeMap<String, Integer>();
```

# Example: `wordCount`

- Put the words into the map:

```
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    wordCounts.put(next, 1);
}
```

But what if the word is already in the map?
This would always keep its count at 1.

# Example: `wordCount`

- Instead, we test whether it was there, and if so, increment it:

```
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    if (!wordCounts.containsKey(next)) {
        wordCounts.put(next, 1);
    } else {
        wordCounts.put(next,
                       wordCounts.get(next) + 1);
    }
}
```

Note that each key can only map to one value. When we put a key in multiple times, only the last value is recorded

# Example: `wordCount`

- We can also print out all the word counts:

```
for (String word : wordCounts.keySet()) {
    int count = wordCounts.get(word);
    System.out.println(count + "\t" + word);
}
```

Note that the keys (the words) occur in sorted order because we are using a **`SortedMap`**.

# Grammars

- **Grammar**:
  A description of a language that describes which sequences of symbols are allowed in that language.

- Grammars describe syntax (rules), not semantics (meaning)

- We will use them to produce syntactically correct sentences

# Grammars

- Use simplified Backus-Naur Form (BNF) for describing language:

```
<symbol> : <expression> | <expression> | ...
```

- ":" means "is composed of"

- "|" means "or"

# Grammars

- We can describe the basic structure of an English sentence as follows:

  **`<s>:<np> <vp>`**

- "A sentence (**`<s>`**) is composed of a noun phrase (**`<np>`**) followed by a verb phrase (**`<vp>`**)."

# Grammars

- We can break down the `<np>` further into proper nouns:

```
<np>:<pn>
<pn>:John|Jane|Sally|Spot|Fred|Elmo
```

- The vertical bar ("|") means that the a `<pn>` can be "John" OR "Jane" OR "Sally" OR …

# Grammars

- Nonterminals:
  - **`<s>`**, **`<np>`**, **`<pn>`**, and **`<vp>`**
  - we don't expect them to appear in an actual English sentence
  - they are placeholders on the left side of rules

- Terminals:
  - "John", "Jane", and "Sally"
  - they can appear in sentences
  - they are final productions on the right side of rules

# Grammars

- We also need a verb phrase rule, **<vp>**:

```
<vp>:<tv> <np>|<iv>
<tv>:hit|honored|kissed|helped
<iv>:died|collapsed|laughed|wept
```

# Grammars

- We can expand the `<np>` rule so that we can have more complex noun phrases:

```
<np>:<dp> <adjp> <n>|<pn>
<pn>:John|Jane|Sally|Spot|Fred|Elmo
<dp>:the|a
<n>:dog|cat|man|university|father|mother|child
```

# Grammars

- We could just make an **<adj>** rule:

```
<adj>:big|fat|green|wonderful|faulty
```

- But we want to have multiple adjectives:

```
<adjp>:<adj>|<adj> <adj>|<adj> <adj> <adj>…
```

- We can use recursion to generate any number of adjectives:

```
<adjp>:<adj>|<adj> <adjp>
```

# Grammars

- Similarly, we can add rules for adverbs **<advp>** and prepositional phrases **<pp>**:

```
<adv>:quickly|drunkenly|stingily|shamelessly
<advp>:<adv>|<adv> <advp>

<pp>:<p> <np>
<p>:on|over|inside|by|under|around
```