Today:
- Asymptotic complexity (from last time)
- Overview of standard Java Collections classes.
  - Iterators, ListIterators
  - Containers and maps in the abstract
  - Views

Readings for Today:  Data Structures, Chapter 2.

Readings for next Topic:  Data Structures, Chapter 3.

---

**Some Intuition on Meaning of Growth**

- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size $N$.
- Entries show the size of problem that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.

$N = \text{problem size}$

<table>
<thead>
<tr>
<th>Time ($\mu$sec) for problem size $N$</th>
<th>1 second</th>
<th>1 hour</th>
<th>1 month</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg N$</td>
<td>$10^{100000}$</td>
<td>$10^{100000000}$</td>
<td>$10^{6\cdot10^{11}}$</td>
<td>$10^{9\cdot10^{14}}$</td>
</tr>
<tr>
<td>$N$</td>
<td>$10^6$</td>
<td>$3.6 \cdot 10^9$</td>
<td>$2.7 \cdot 10^{12}$</td>
<td>$3.2 \cdot 10^{15}$</td>
</tr>
<tr>
<td>$N \lg N$</td>
<td>$63000$</td>
<td>$1.3 \cdot 10^8$</td>
<td>$7.4 \cdot 10^{10}$</td>
<td>$6.9 \cdot 10^{13}$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>$1000$</td>
<td>$60000$</td>
<td>$1.6 \cdot 10^6$</td>
<td>$5.6 \cdot 10^7$</td>
</tr>
<tr>
<td>$N^3$</td>
<td>$100$</td>
<td>$1500$</td>
<td>$14000$</td>
<td>$150000$</td>
</tr>
<tr>
<td>$2^N$</td>
<td>$20$</td>
<td>$32$</td>
<td>$41$</td>
<td>$51$</td>
</tr>
</tbody>
</table>

---

**New Topic: Data Types in the Abstract**

- Most of the time, should not worry about implementation of data structures, search, etc.
- What they do for us—their specification—is important.
- Java has several standard types (in java.util) to represent collections of objects
  - Six interfaces:
    * Collection: General collections of items.
    * List: Indexed sequences with duplication
    * Set, SortedSet: Collections without duplication
    * Map, SortedMap: Dictionaries (key $\mapsto$ value)
  - Concrete classes that provide actual instances: LinkedList, ArrayList, HashSet, TreeSet.
  - To make change easier, purists would use the concrete types only for new, interfaces for parameter types, local variables.

---

**Collection Structures in java.util**

![Diagram of Collection Structures]

- Collection
  - List
  - Set
    - SortedSet
  - Map
    - Stack
    - Key:
      - interface
      - class
    - : extends
    - ---: implements
The Collection Interface

- **Collection interface. Main functions promised:**
  - **Membership tests:** `contains (∈)`, `containsAll (⊆)`
  - **Other queries:** `size`, `isEmpty`
  - **Retrieval:** `iterator`, `toArray`
  - **Optional modifiers:** `add`, `addAll`, `clear`, `remove`, `removeAll (set difference)`, `retainAll (intersect)`

- **Design point (a side trip): Optional operations may throw** `UnsupportedOperationException`

- **An alternative design would have separate interfaces:**
  ```java
  interface Collection { contains, containsAll, size, iterator, ... }
  interface Expandable { add, addAll } 
  interface Shrinkable { remove, removeAll, difference, ... }
  interface ModifiableCollection extends Collection, Expandable, Shrinkable { } 
  ```

  *You'd soon have lots of interfaces. Perhaps that's why they didn't do it that way."

The List Interface

- **Extends Collection**
- **Intended to represent indexed sequences (generalized arrays)**
- **Adds new methods to those of Collection:**
  - **Membership tests:** `indexOf`, `lastIndexOf`
  - **Retrieval:** `get (i)`, `listIterator()`, `sublist (B, E)`
  - **Modifiers:** `add` and `addAll` with additional index to say where to add. Likewise for removal operations. Set operation to go with `get`.
- **Type** `ListIterator<Item> extends Iterator<Item>`:
  - Adds `previous` and `hasPrevious`.
  - `add`, `remove`, and `set` allow one to iterate through a list, inserting, removing, or changing as you go.
- **Important Question:** What advantage is there to saying `List L` rather than `LinkedList L` or `ArrayList L`?

Views

**New Concept:** A view is an alternative presentation of (interface to) an existing object.

- For example, the sublist method is supposed to yield a "view of" part of an existing list:

  ```
  List<String> L = new ArrayList<String>();
  L.add("at"); L.add("ax"); ...
  List<String> SL = L.sublist (1,4);
  ```

- Example: after `L.set(2, "bag")`, value of `SL.get(1)` is "bag", and after `SL.set(1,"bad")`, value of `L.get(2)` is "bad".
- Example: after `SL.clear()`, `L` will contain only "at" and "cat".
- Small challenge: "How do they do that?!"

Maps

- **A Map is a kind of "modifiable function:"

  ```java
  package java.util;
  public interface Map<Key,Value> {
  Value get (Object key); // Value at KEY.
  Object put (Key key, Value value); // Set get(KE) -> VALUE
  ...
  }
  ```

  ```java
  Map<String,String> f = new TreeMap<String,String> ();
  f.put ("Paul", "George"); f.put ("George", "Martin");
  f.put ("Dana", "John");
  // Now f.get ("Paul").equals ("George")
  //  f.get ("Dana").equals ("John")
  //  f.get ("Tom") == null
  ```
public interface Map<Key, Value> {
    /* VIEWS */
    Set<Key> keySet();
    Collection<Value> values();
    Set<Map.Entry<Key, Value>> entrySet();
}

Using example from previous slide:

for (Iterator<String> i = f.keySet().iterator(); i.hasNext();)
    i.next() ===> Dana, George, Paul

// or, just:
for (String name : f.keySet())
    name ===> Dana, George, Paul

for (String parent : f.values())
    parent ===> John, Martin, George

for (Map.Entry<String, String> pair : f.entrySet())
    pair ===> (Dana, John), (George, Martin), (Paul, George)

f.keySet().remove("Dana"); // Now f.get("Dana") == null