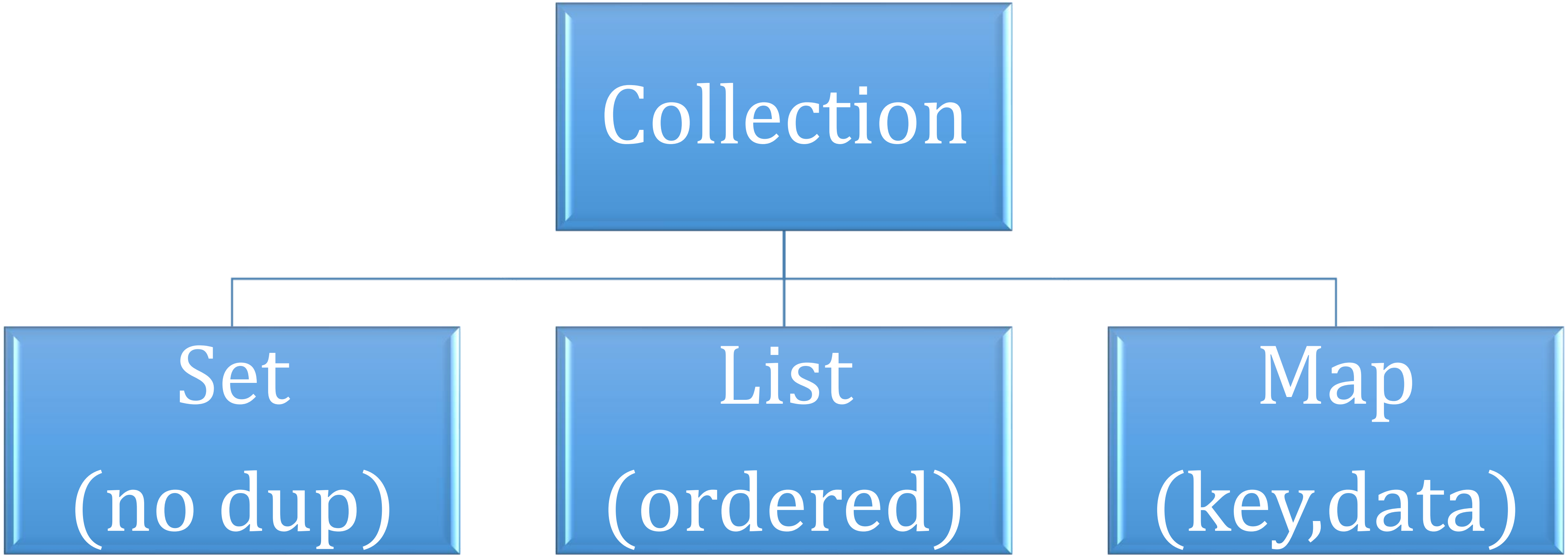
Using Collections

The Java Collections Infrastructure



Typical Collection Usage

- All you need is (for instance) the List interface
- Find a concrete class which implements List (e.g. ArrayList)
- Declare field/variable as List
- Instantiate field/variable using concrete class

```
private List blockList;
…
blockList = new ArrayList<Block>();
```

To use a different implementation, change this line

Joel Spolsky*: Law of Leaky Abstractions

- We like to think of the world abstractly
  - My program uses a list – an ordered collection of elements
- Sometimes we need to know about the concrete implementation
  - If the backing store of a list is an array, insertion/deletion can be slow
  - If the backing store of a list is a linked list, direct indexing can be slow
  - Both act as lists, but one does some list things better than the other
  - We may want to choose an implementation based on our application

*Author of *Joel on Software* blog, co-founded Stack Overflow

Some Classes implementing Set

- EnumSet – Backing store: bit vector
  - Requires small fixed enumerated domain
  - Very fast add, remove, contains (one cycle)
  - + methods: allOf(t) clone() complementOf(s) copyOf(c) noneOf(t) of(…e) range(from,to)
- HashSet - Backing store: HashMap
  - constant time add, remove, contains, and size
  - Slow traversal
- LinkedHashSet – Backing Store: HashMap + linked list
  - stabilizes "order" of the set
- TreeSet – Backing Store: TreeMap
  - log(n) time add, remove, and contains

Some Classes implementing List

- ArrayList – Backing store: array
  - Fast direct access to elements
  - Occasional slow add/delete to enlarge/shrink array
  - Slower insert/delete from beginning of list
- LinkedList - Backing store: Doubly linked list of nodes
  - constant time add, remove
  - Slow direct access – iterate is faster
  - Also implements Deque and Queue interfaces
- Vector – Backing Store: array-like, but with shrink and grow
  - Thread safe, but slower than ArrayList

ArrayList vs. array

Function	array	ArrayList
Declare	*type*[] *var*	ArrayList<*type*> *var* (type must extend Object)
Instantiate	new *type*[*size*]	new ArrayList<*type*>()
Read element i	*var*[i]	*var*.get(i)
Write element i	*var*[i]=*value*	*var*.set(i,*value*)
add element at end	---	*var*.add(*value*)
all element in the middle	---	*var*.add(i,*value*)
Shrink/Enlarge	instantiate new larger/smaller array and copy old to new	Automatic
Enhanced loop	for(*type v* : *var*) { }	for(*type v* : *var*) { }
Performance	Good	Equal except inserting early and when grow or shrink is needed

Some Classes Implementing Map

- EnumMap – Backing Store: array
  - Requires small fixed enumerated key domain
  - Very fast
- HashMap – backing store – array?
  - Very fast insert/delete
  - Slow/unstable traversal
- HashTable – Thread safe hash map
- LinkedHashMap – backing store: hash map with linked list
  - Stabilizes and speeds up traversal
- TreeMap
- WeakHashMap

Binning for Unsorted Items

- Keep two or more bins… lists of objects… bins are a list of lists
- Quick function to determine what bin an element belongs in
- Trick is to equalize binsize… so for m bins, binsize ~= n/m
- Time to insert : find bin, add to bin - fast
- Time to search : find bin, search in bin – O(n/m)
- Time to delete : find bin, find in bin, delete – O(n/m)

- More bins mean faster access, but more memory

Ultimate binning

- Separate bin for each item
- Problem… need a specialized function to determine what bin element x is in
  - Needs to run fast
  - Needs to guarantee that if two elements are the same, they go to the same bin
  - Needs to guarantee that two different elements go to different bins
- Problem: Sparse usage… most bins remain unused
  - Consider a bin for each Lottery number – e.g. pick 6: 45 55 32 91 40 46
  - There are 10^{12} possible lottery tickets!

Hashing

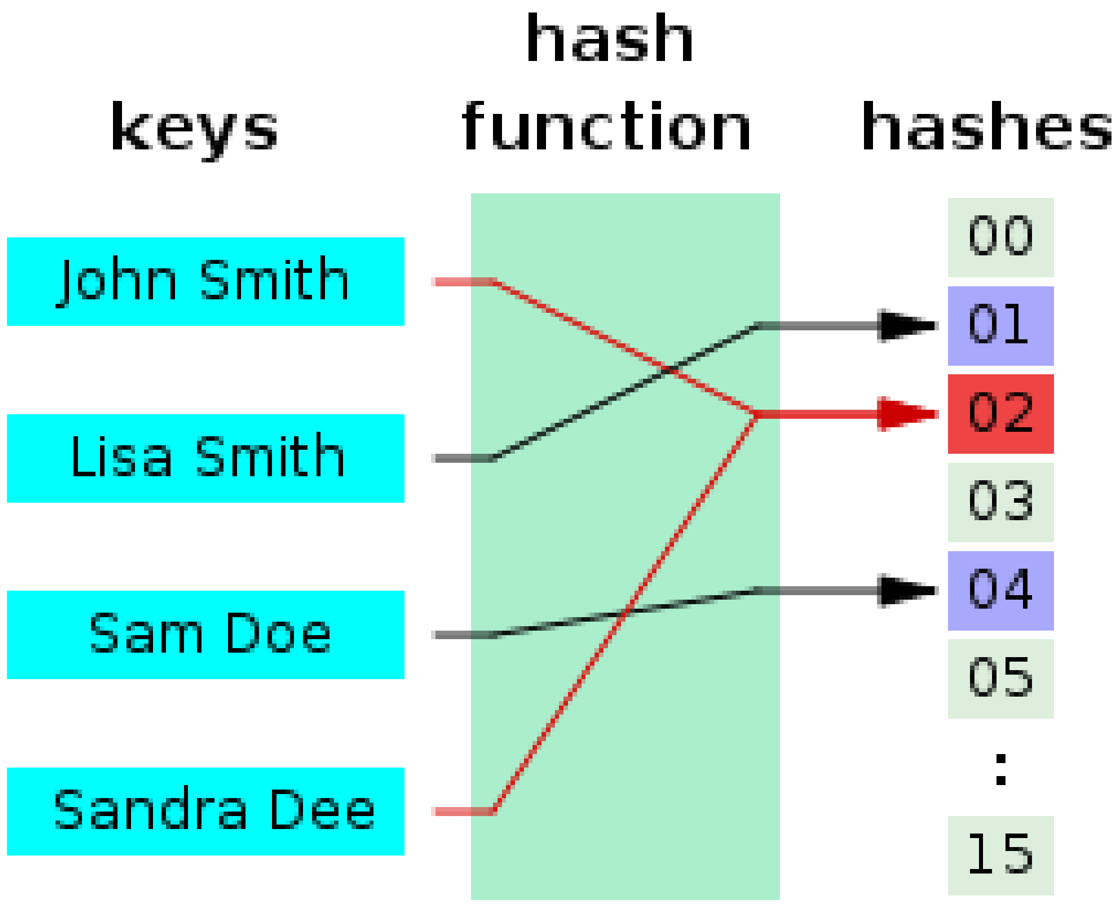
- Hashing is a form of a binning algorithm
- Function to translate from "key" to an index in an array
- Number of bins = number of elements in the array
- The function that performs the translation is a "hash" function

$$index = hash(key)$$

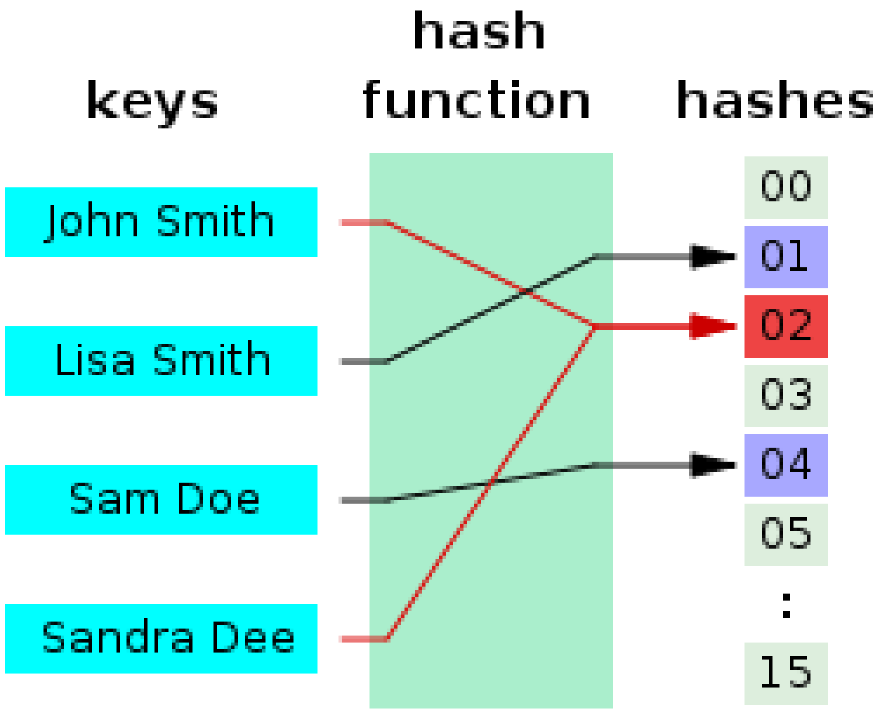
- Guarantee: if $\text{key}_1 == \text{key}_2$, then $\text{hash}(\text{key}_1) == \text{hash}(\text{key}_2)$
- Not guaranteed: if key_1 != key_2, then $\text{hash}(\text{key}_1) =? \text{hash}(\text{key}_2)$
- "Hash Collision" if key_1 != key_2, but $\text{hash}(\text{key}_1) == \text{hash}(\text{key}_2)$
- Hash function designed to minimize collisions

Example Hash

- Translate keys to index 0-15
- Each key hashes to the same index every time
- Multiple keys may map to a single index



Example Hash Table



	Name	Town	ID
0			
1	Lisa Smith	Vestal	6894
2	John Smith	Endicott	1548
	Sandra Dee	Binghamton	6442
3			
4	Sam Doe	Johnson City	2954
5			
…			
15			

Warning: Modifying Collections in loops

```
for ( Block b : blockList) {
    if (!b.isUsed()) blocklist.remove(b);
}
```

```
/* ALTERNATE… */
for (int i=0; i<blocklist.size();i++) {
    if (!blocklist.get(i).isUsed())
    blocklist.remove(i);
}
```

- Question: do any of these work?

```
/* ALTERNATE 2 … */
for ( Iterator it=blocklist.iterator(); it.hasNext();) {
    Block b = it.next();
    if (!b.isUsed()) it.remove();
}
```