Anuranjan Kumar

- Generics – what is it and why do we use it
- Core Collection Interfaces
- List Inteface
- ArrayList
- Iterators
- Set Interface
- HashSet
- Map Interface
- HashMap
- Comparable vs. Comparator
- Implementation of HashMaps

Generics

- What is a *Generic type*?

Generic class/interface parametrized over *types*

- Why use *Generics*?

Re-use the same code with different *types* of inputs

Following non-generic class `Demo` operates for objects of any type, but no way to verify compile-time errors:

```
public class Demo {

   private Object object;

   public void set(Object object) {
      this.object = object;
   }

   public Object get() {
      return object;
   }

}
```

A *generic class* is defined with the following format:

```
class ClassName<T1, T2, ..., Tn> { /* ... */ }
```

The *type parameter* section, delimited by angle brackets (`<>`), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

To update the `Demo` class to use generics, we create a generic type declaration by changing the code "`public class Demo`" to "`public class GenericDemo<T>`".

This introduces the type variable, `T`, that can be used anywhere inside the class.

Following generic class `GenericDemo` operates for any *type variable* which is **non-primitive**:

```
public class GenericDemo<T> {

   private T t;

   public void set(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

}
```

Invoking and Instantiating a Generic Type

We perform a *generic type invocation*, which replaces T with some concrete value, such as `Integer`:

```
Demo<Integer> demo1 = new Demo<Integer>();
```

Or,

```
Demo<Integer> demo1 = new Demo<>();
```

Multiple Type Parameters

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String,Integer> ("Even", 8);
Pair<String, String> p2 = new OrderedPair("hello", "world");
```

It is also valid to pass a String and an int to the class. This is due to "*autoboxing*"

We can also substitute a type parameter (i.e., `K` or `V`) with a parameterized type (eg., `List<String>`). For example, using the `OrderedPair<K, V>` as below:

```
OrderedPair<String, Demo<Integer>> p =
    new OrderedPair("primes", new Demo<Integer>());
```

Autoboxing and Unboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object *wrapper* classes.
- For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on.
- If the conversion goes the other way, this is called *unboxing*.

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short

Why Use Generics?

- Stronger type checks at compile time and issues errors if the code violates type safety.
- Enabling programmers to implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.
- Elimination of casts.

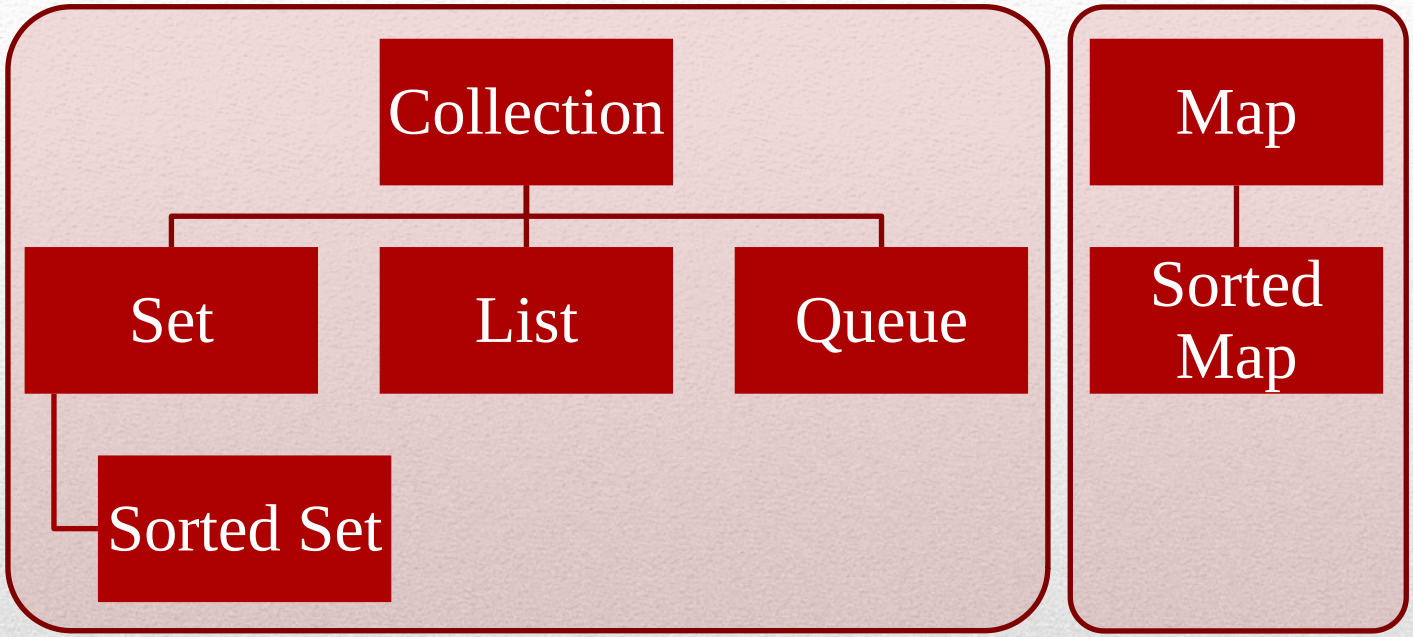
- The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- When re-written to use generics, the code does not require casting:

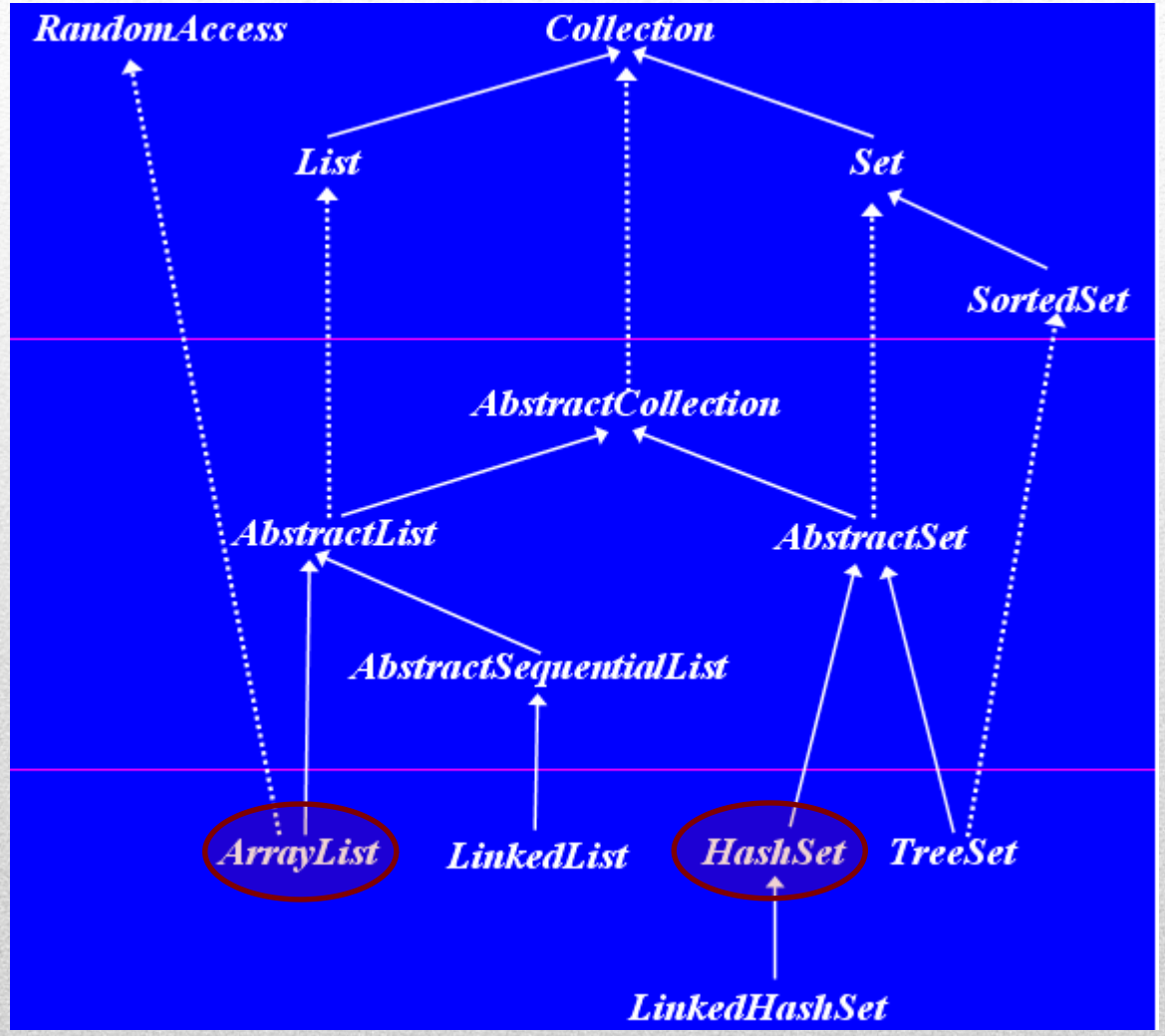
```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast required
```

Core Collection Interfaces



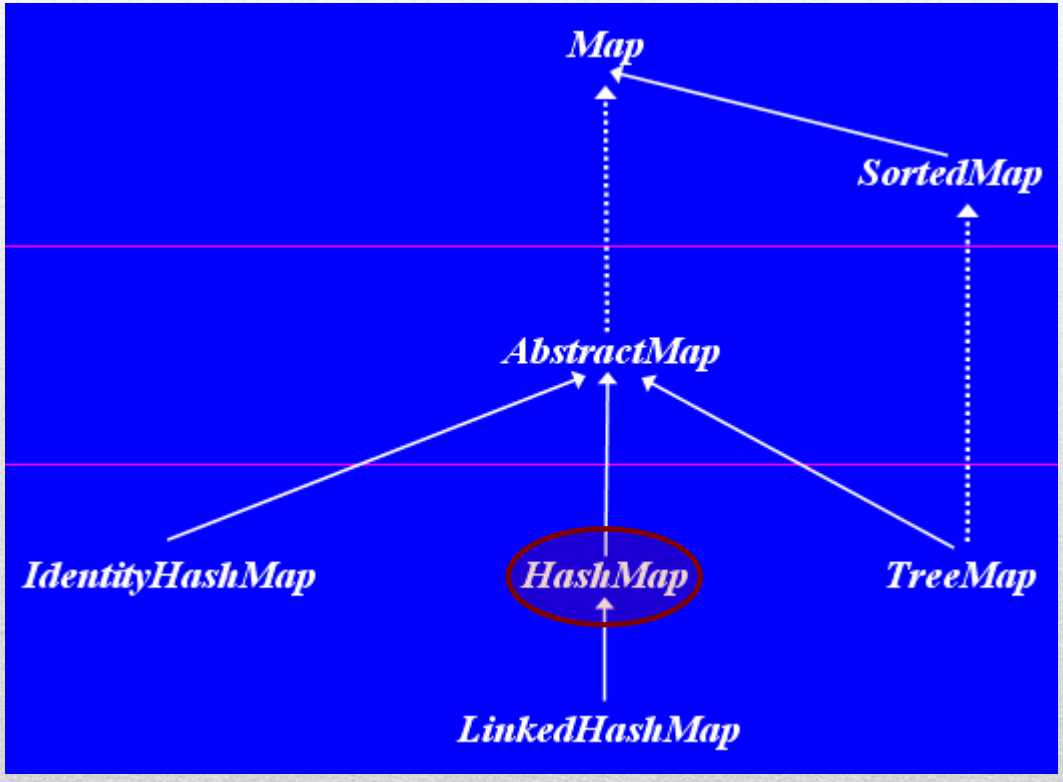
- Collection – root of all Collection hierarchy
  - May/may not allow duplicates
  - May be ordered/unordered
  - No direct implementation

The Complete Collection Interface



Courtesy: http://www.cs.cmu.edu

The Complete Map Interface



Courtesy: http://www.cs.cmu.edu

The List Interface

- An ordered `Collection` (or *sequence*)
- May contain duplicate elements

- **Positional access** — manipulates elements based on their numerical position in the list
- **Search** — searches for a specified object in the list and returns its numerical position
- **Iteration** — extends `Iterator` semantics to take advantage of the list's sequential nature
- **Range-view** — performs arbitrary *range operations* on the list.

ArrayList

- A resizable-array implementation of `List`
- Provides methods to manipulate the size of the array that is used internally to store the list

Constructor Details:

```
public ArrayList();
public ArrayList(int initialCapacity);
public ArrayList(Collection<> c);
```

Positional Access Operations:

```
public E get(int index);
public E set(int index, E element);
public boolean add(E element);
public void add(int index, E element);
public boolean addAll(Collection<E> c);
public boolean addAll(int index, Collection<E> c);
public E remove(int index);
public boolean remove(Object o);
public boolean removeAll(Collection<E> c);
```

Search Operations:

```
public int indexOf(Object o);
public int lastIndexOf(Object o);
```

Iteration:

```
public Iterator<E>  iterator();
public ListIterator<E>  listIterator();
public ListIterator<E> listIterator(int index);
```

Iterator<E> (Uni-directional)	ListIterator<E> extends Iterator<E>
`Public boolean hasNext();`	`Public boolean hasPrevious();`
`Public E next();`	`Public E previous();`
`Public void remove();`	`Public int nextIndex();`
	`Public int previousIndex();`

Range-View Operations:

```
public List<E> subList(int fromIndex, int toIndex);
```

Other Operations:

```
public void clear();
public boolean contains(Object o);
public int size();
public Object[] toArray();
public void trimToSize();
```

The Set Interface

- An unordered `Collection`
- Cannot contain duplicate elements

- **Basic operations** — get the size, adding an element, etc.
- **Bulk operations** — standard set-algebraic operations such as subset, union, intersection and difference
- **Array operations** — similar to those in `List` interface

HashSet

- Backed by `HashTables` to store the elements of the set

Constructor Details:

```
public HashSet();
public HashSet(Collection<> c);
public HashSet(int initialCapacity);
public HashSet(int initialCapacity, float loadFactor);
```

Basic Operations:

```
public int size();
public boolean isEmpty();
public boolean contains(Object o);
public boolean add(E element);
public boolean remove(Object o);
public Iterator<E> iterator();
```

Bulk Operations:

```
public boolean containsAll(Collection<?> c);
public boolean addAll(Collection<? extends E> c);
public boolean removeAll(Collection<?> c);
public boolean retainAll(Collection<?> c);
public void clear();
```

The Map Interface

- Collection of Key-Value pairs
- Cannot contain duplicate keys
- Each key can map to at most one value

- **Basic operations** —adding an pair, removing a pair, etc.
- **Bulk operations** — operations involving complete map objects
- **Collection views** — allow complete map to be viewed as collections of keys, values or key-value pairs
- **Multimaps** – nothing but maps with values being `List` instances. Through this, it is possible to map one key to multiple values in terms of a `Collection` object

HashMap

- Permits `null` values and `null` keys as well

Constructor Details:

```
public HashMap();
public HashMap(Map<K, V> m);
public HashMap(int initialCapacity);
public HashMap(int initialCapacity, float loadFactor);
```

Basic Operations:

```
public V put(K key, V value);
public V get(K key);
public V remove(K key);
public boolean containsKey(K key);
public boolean containsValue(V value);
public int size();
public boolean isEmpty();
```

Bulk Operations:

```
public void putAll(Map<K, V> m);
public void clear();
```

Collection Views:

```
public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K,V>> entrySet();
```

Map Algebra – Application of Collection views:

```
if (m1.entrySet().containsAll(m2.entrySet())) { ...
}

if (m1.keySet().equals(m2.keySet())) {
...
}

Set<KeyType> commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());
```

Comparable vs.Comparator

Comparable	Comparator
capable of comparing itself with another object	capable of comparing two objects , which are instances of a different class
Implements the interface `java.lang.Comparable`	Implements the interface `java.lang.Comparator`
`public int compareTo(Object o);`	`public int compare(Object o1, Object o2);`
returns a negative integer, zero, or a positive integer, if the first argument is less than, equal to, or greater than the second, respectively	

- `Comparable` in Java is used to implement **natural ordering of objects**.

- If any class implements `Comparable` interface then collection of that object (`List` or `Array`) can be sorted automatically by using `Collections.sort()` or `Arrays.sort()` method and object will be sorted based on their natural order defined by `compareTo()` method.

- Objects which implement `Comparable` can be used as keys in a `SortedMap` like `TreeMap` or elements in a `SortedSet` like `TreeSet`, without specifying any `Comparator`.

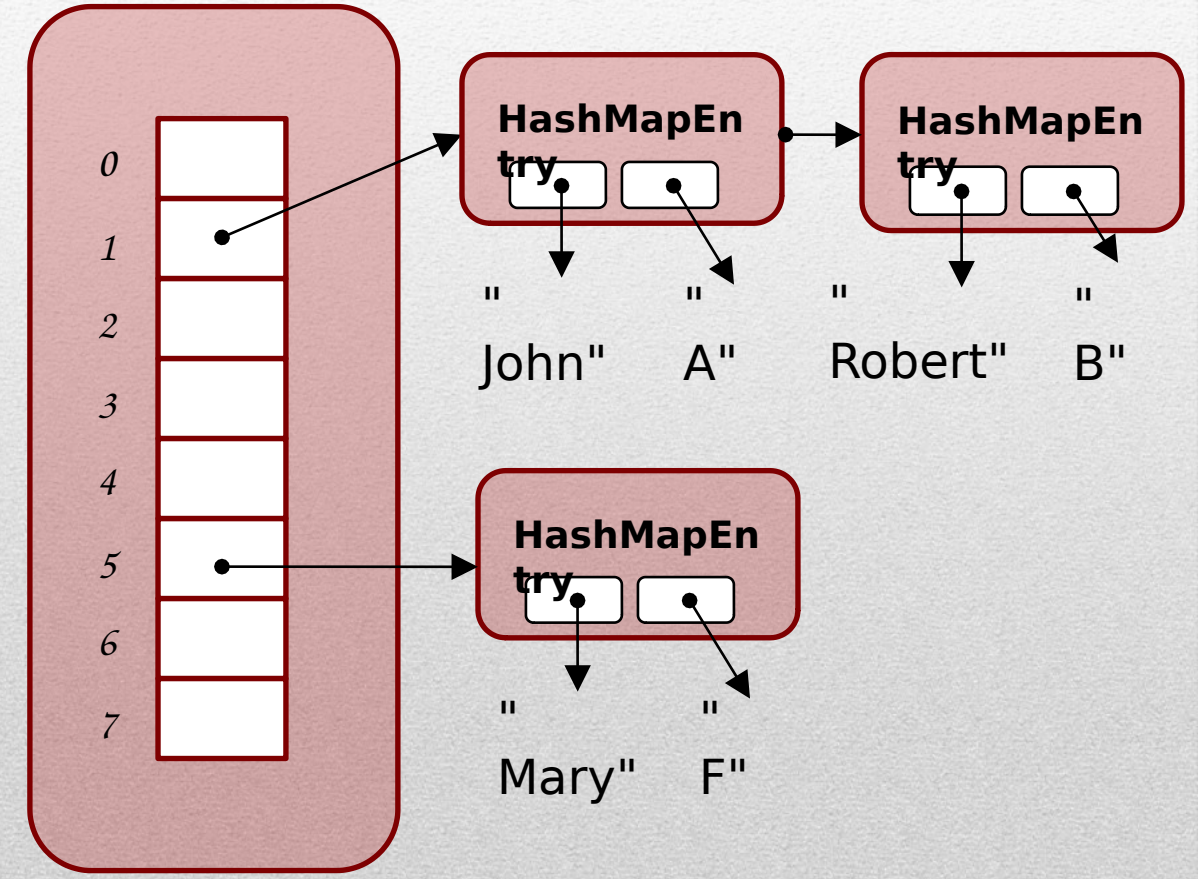
```
public class Person implements Comparable {
    private int personId;
    private String name;

    // define natural ordering
    public int compareTo(Person p) {
        return this.personId - p.personId ;
    }
    // getters and setters
}
public class SortPersonByName implements Comparator{
    public int compare(Person p1, Person p2) {
        return p1.getName() - p2.getname();
    }
}

Collections.sort(personsList, new SortPersonByName());
```

HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");
```

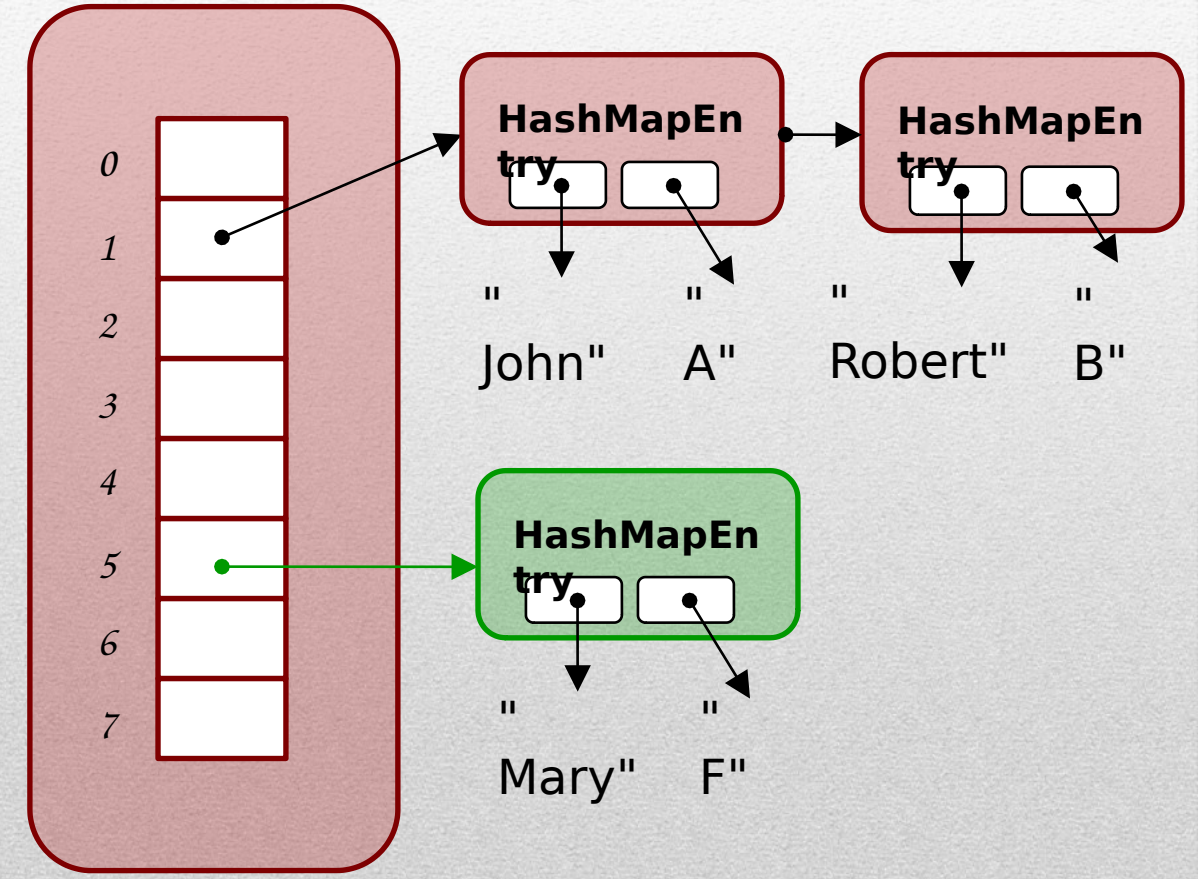


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");
```

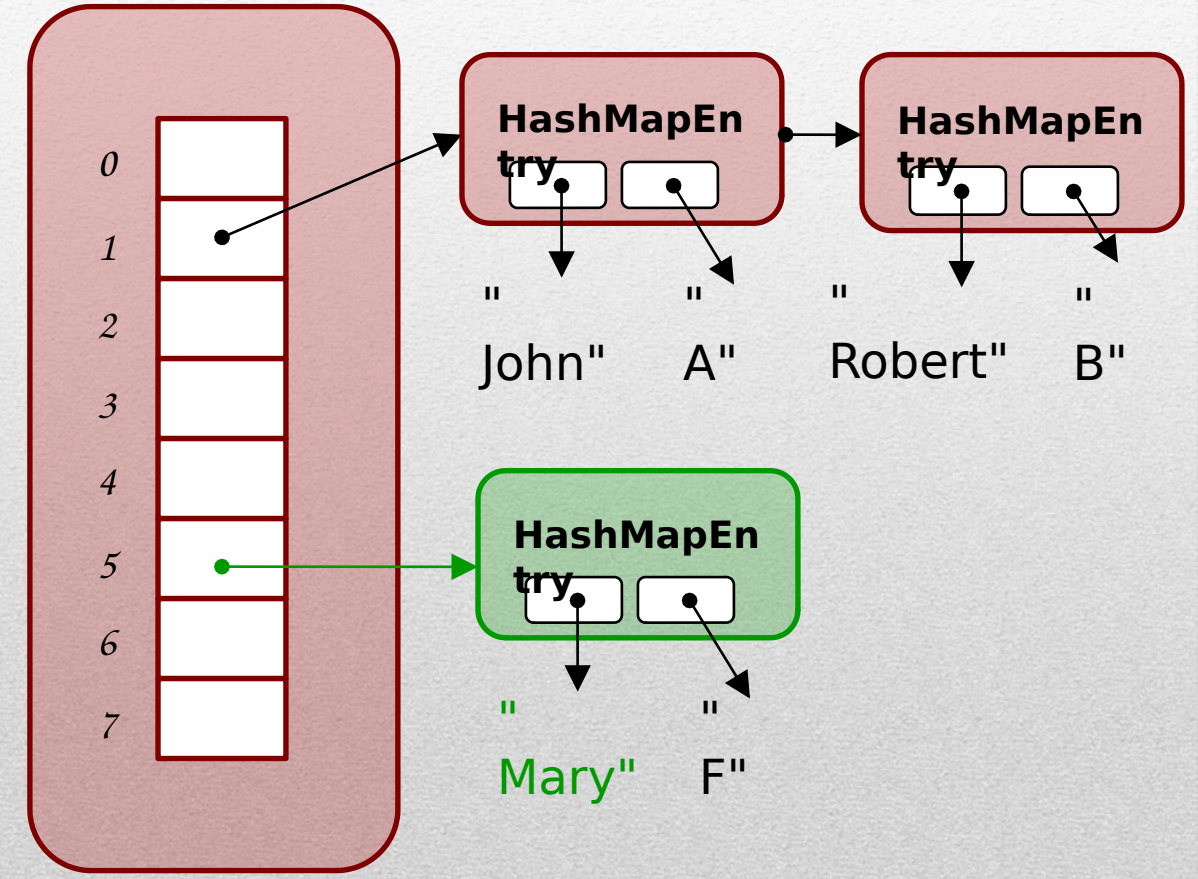


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");
```

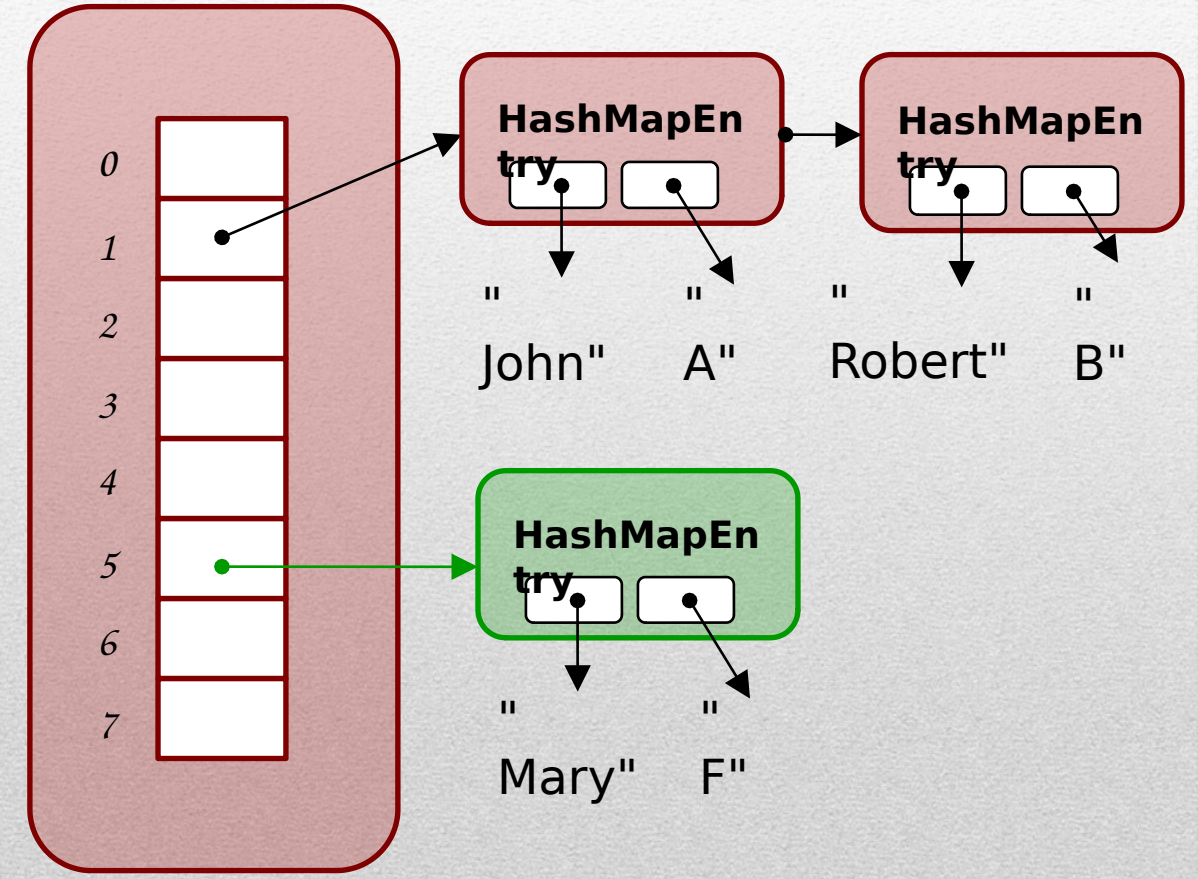


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
```

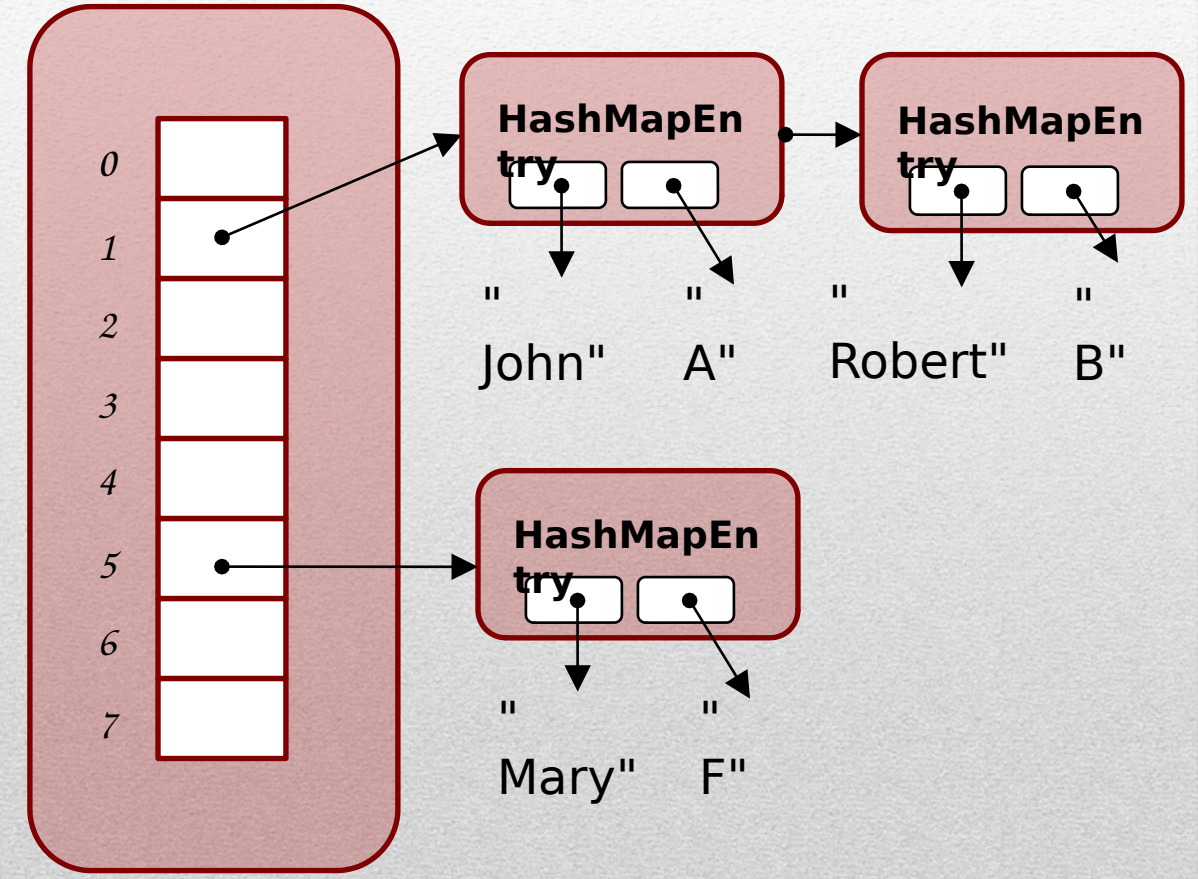


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");
```

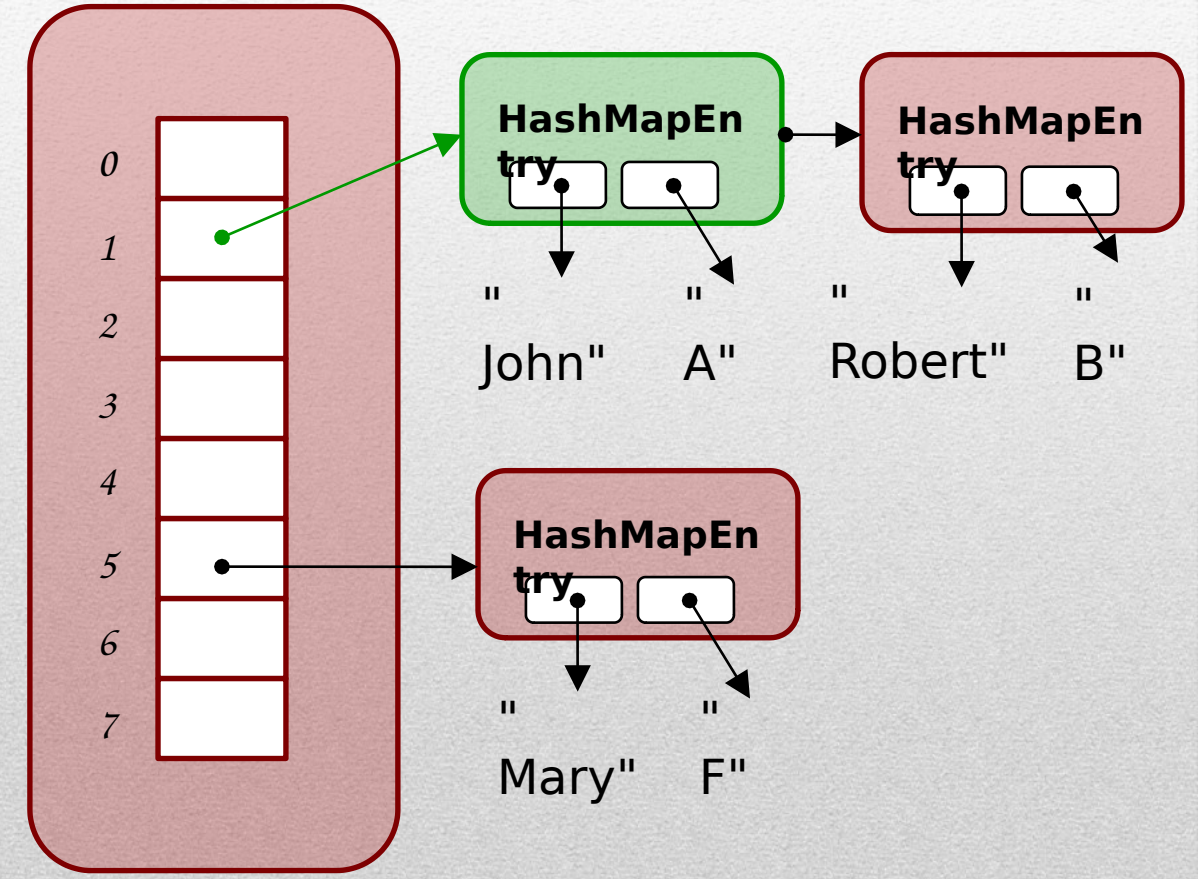


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");
```

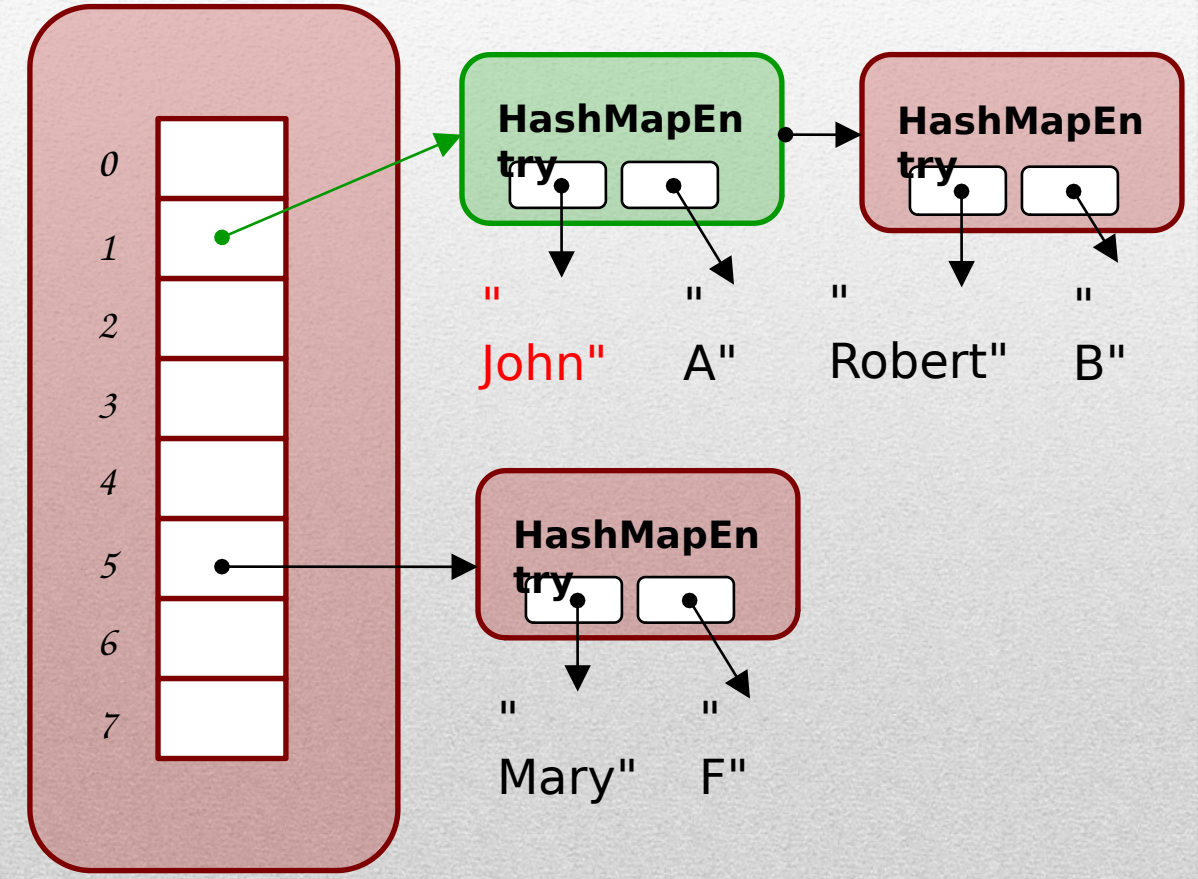


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");
```

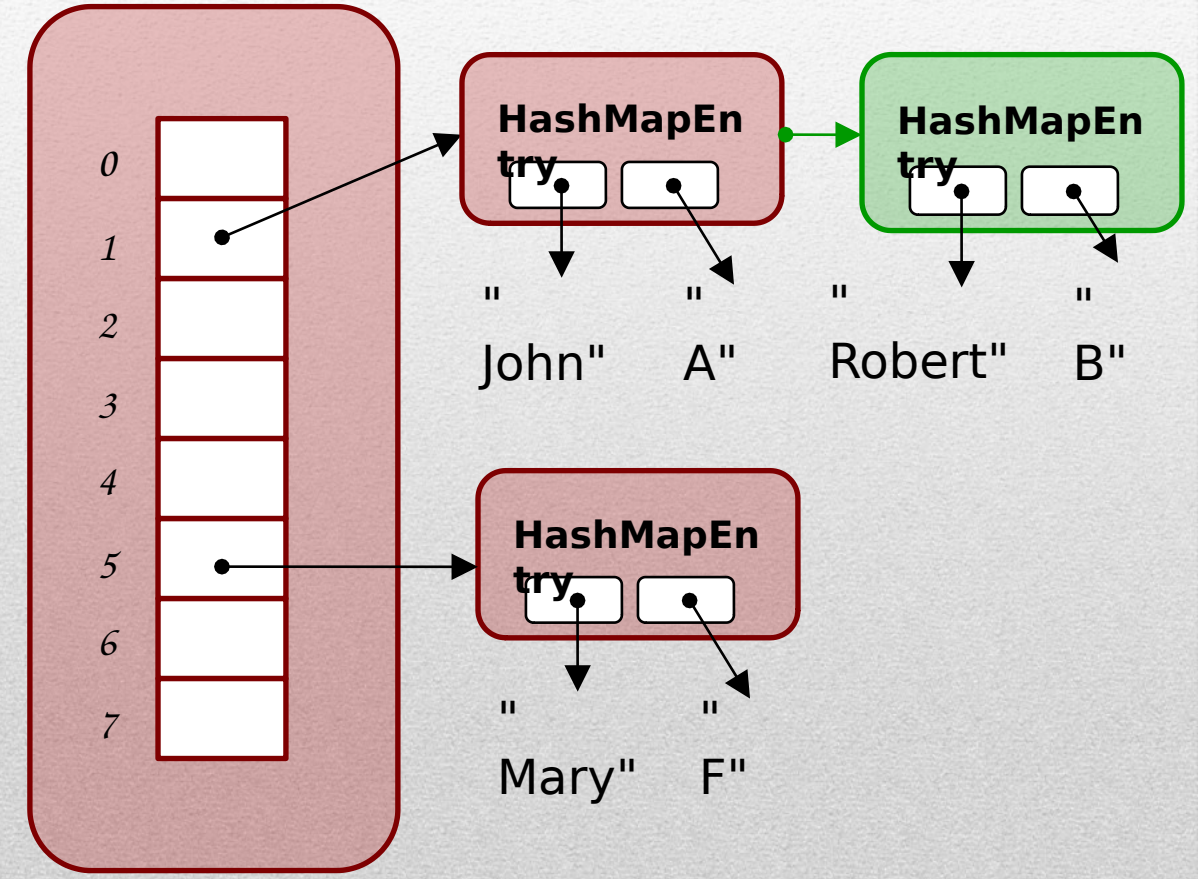


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");
```

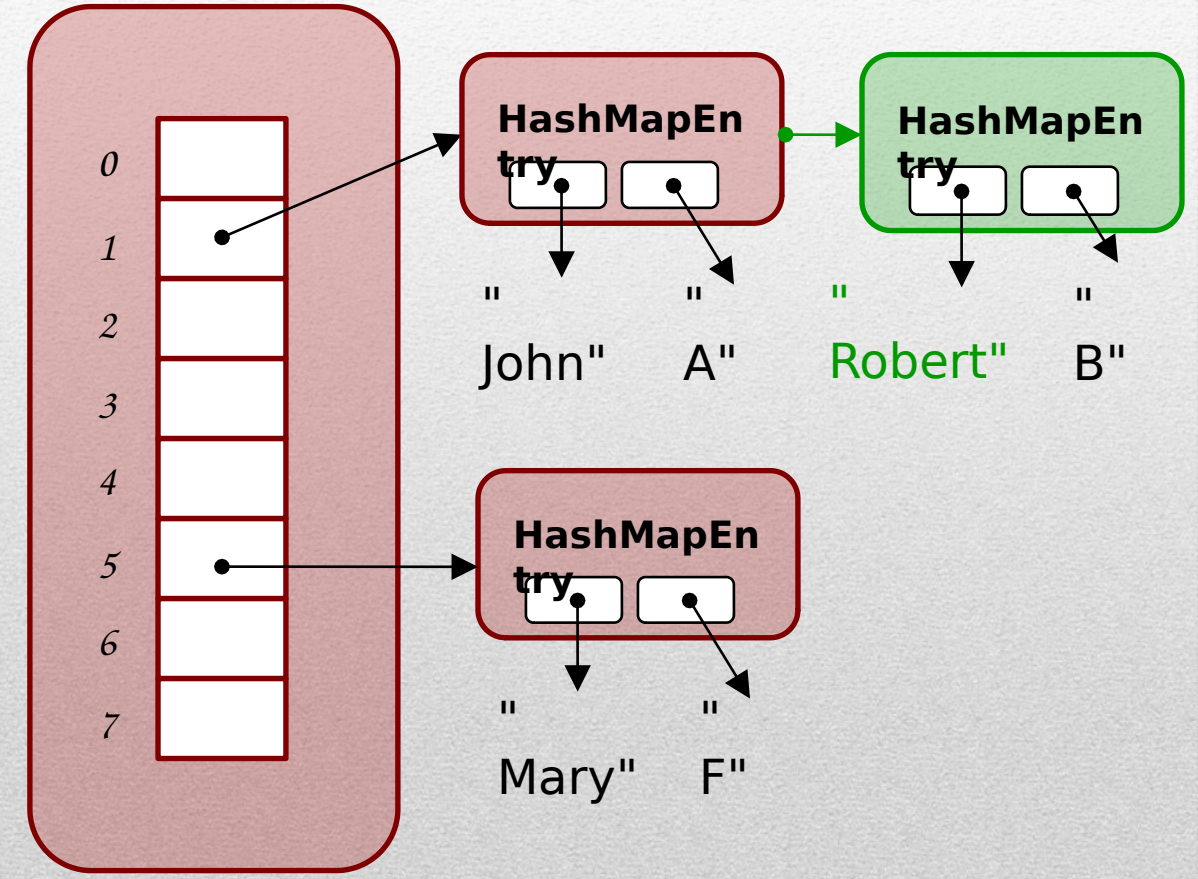


HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

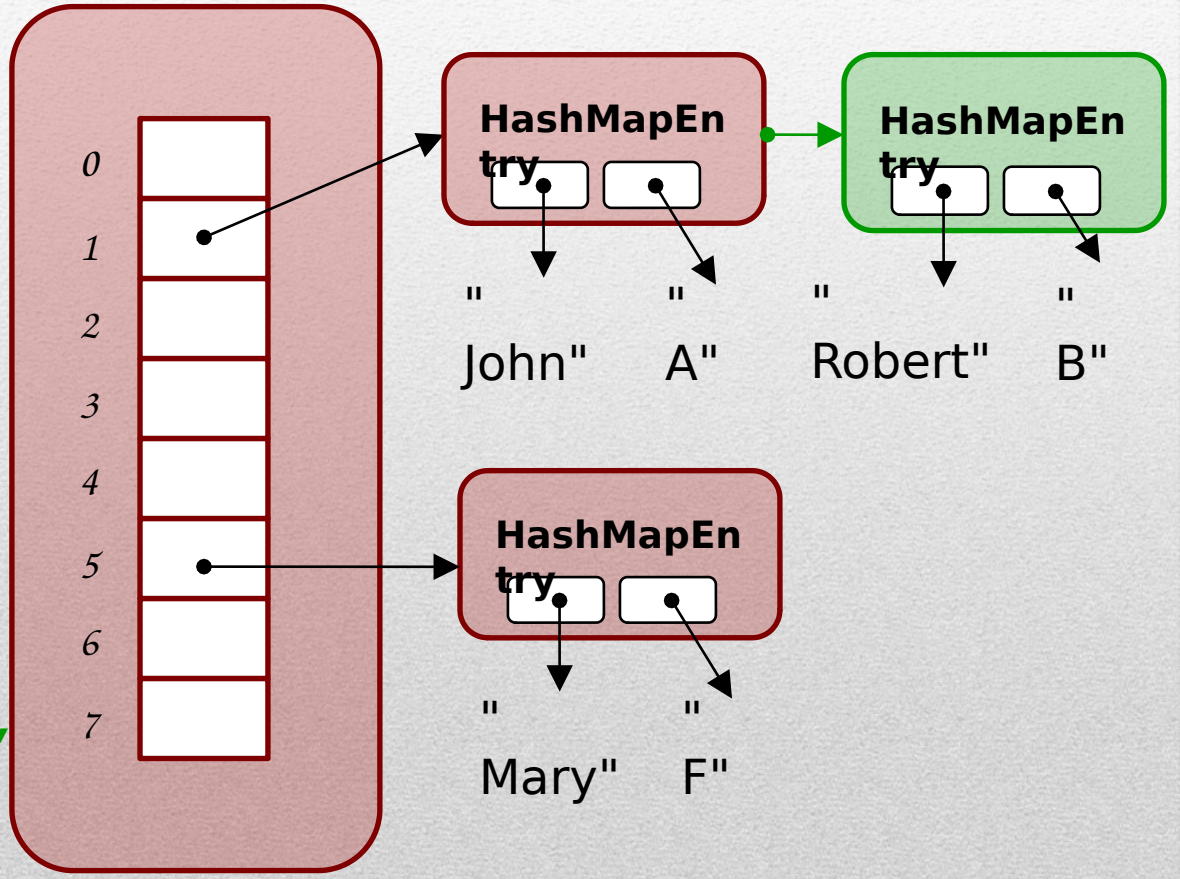
```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");
```



HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

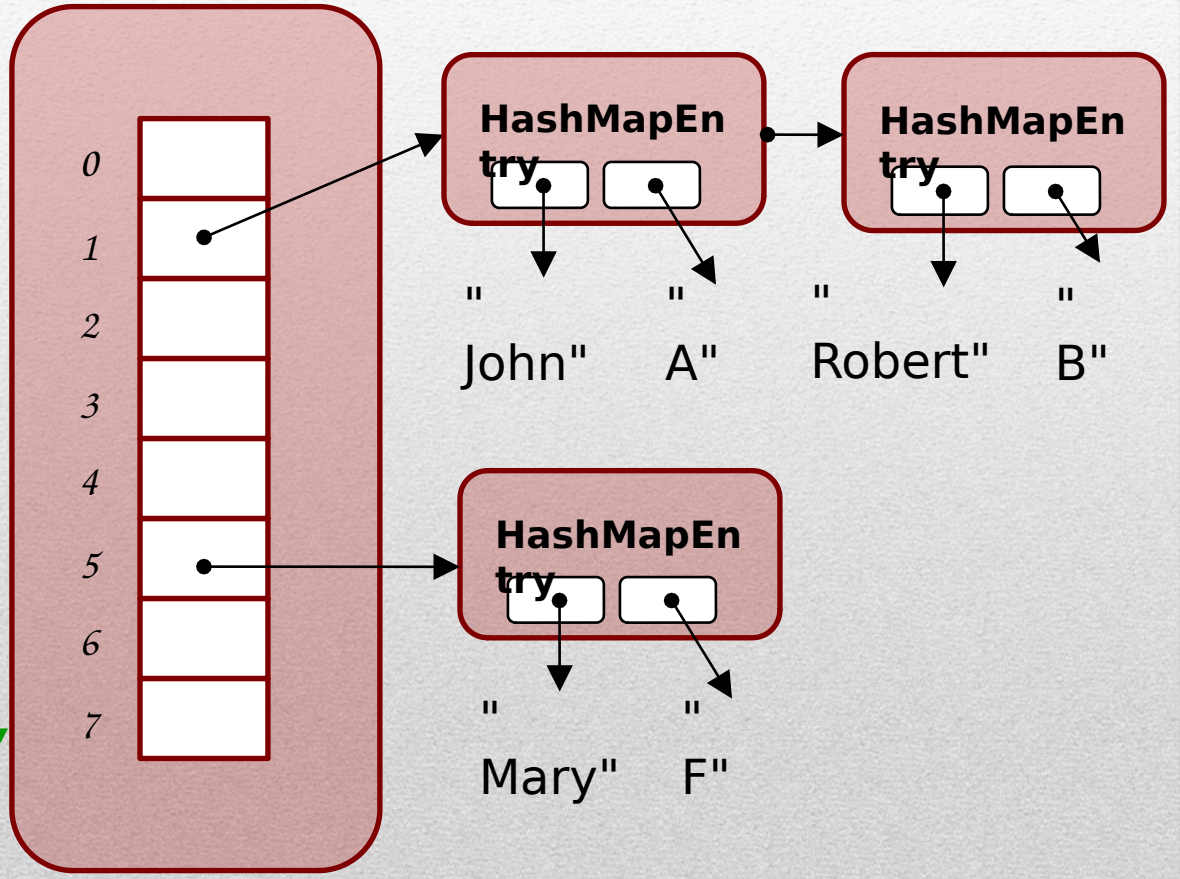



```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");// "B"
```

HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

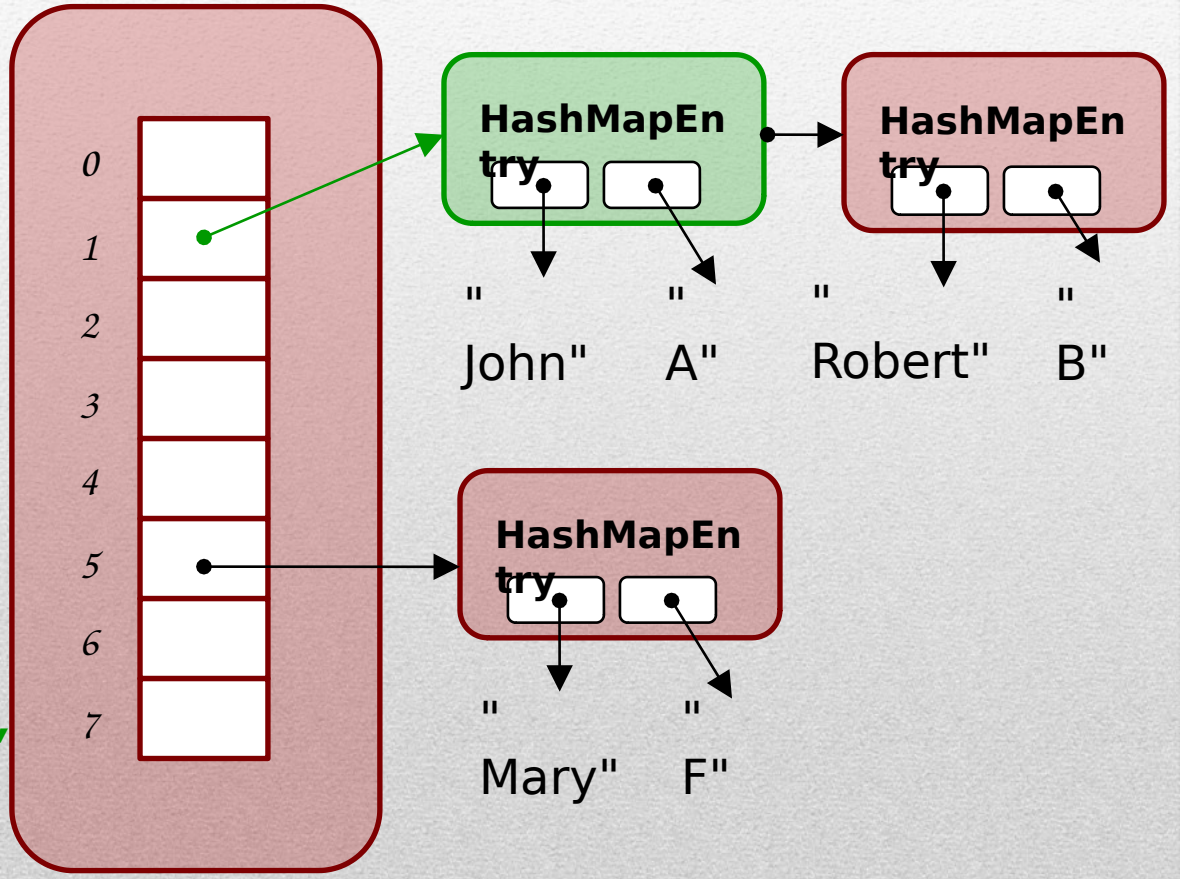



```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");// "B"

map.put("John", "C");
```

HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

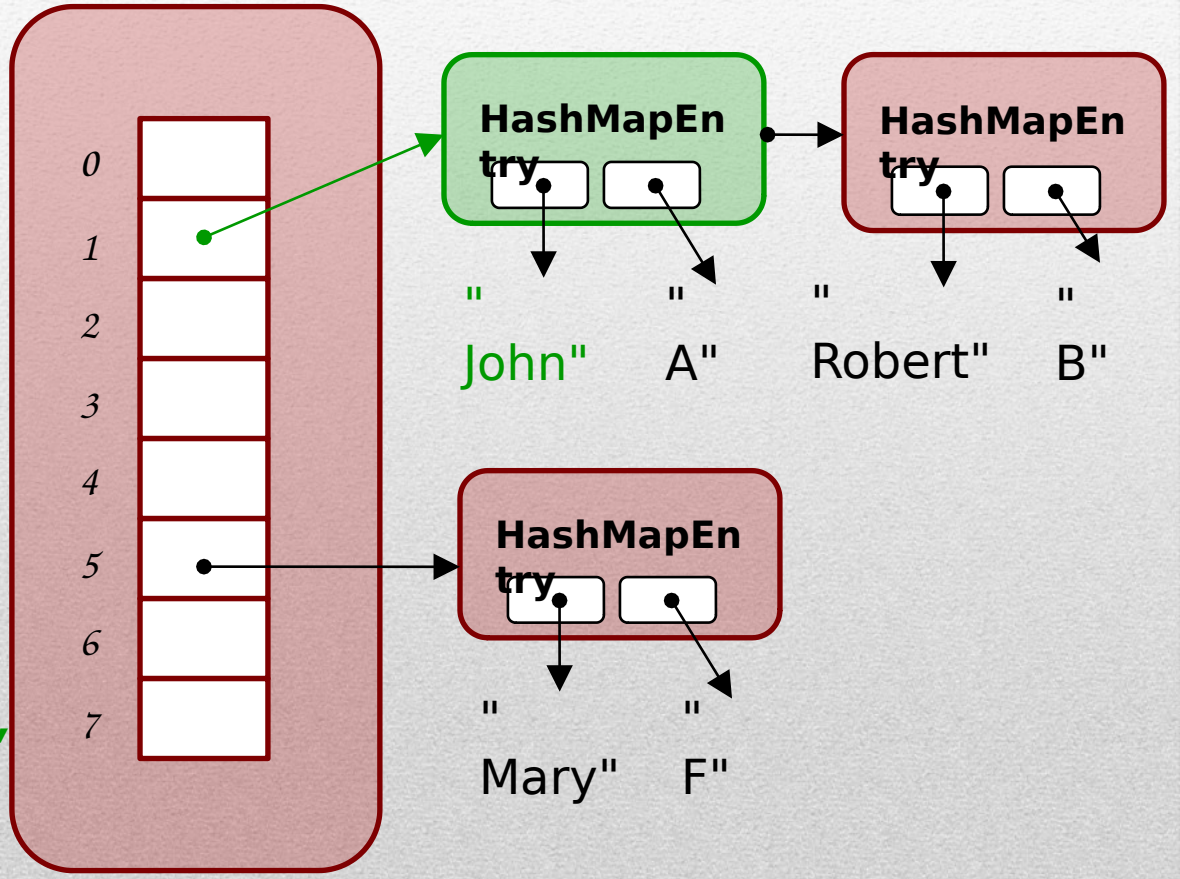



```
map.put(“John", "A");
map.put(“Mary", “F");
map.put(“Robert", “B");

map.get(“Mary");// “F”
map.get(“Robert”);// “B”

map.put(“John", “C");
```

HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1

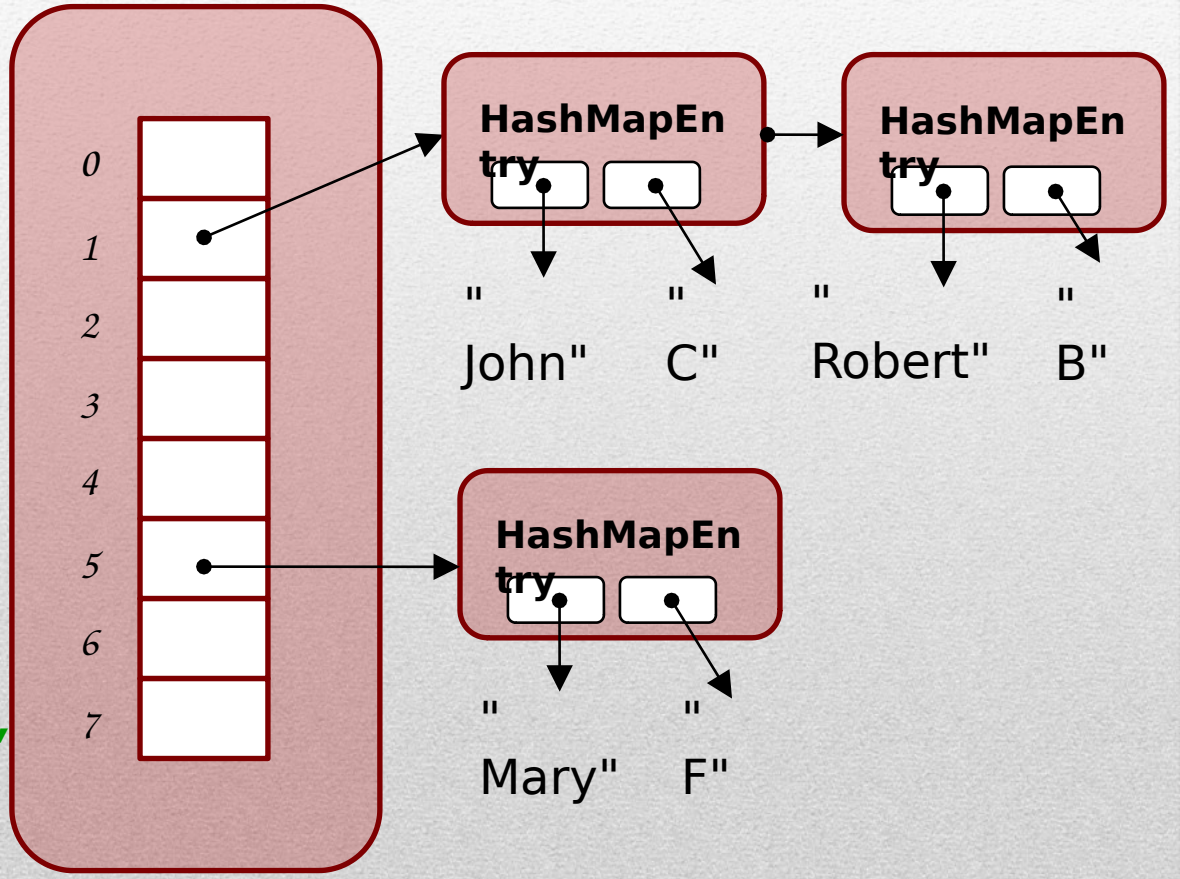



```
map.put("John", "A");
map.put("Mary", "F");
map.put("Robert", "B");

map.get("Mary");// "F"
map.get("Robert");// "B"

map.put("John", "C");
```

HashMap Implementation

- The class should implement `.hashCode()` and `.equals()` methods.

String	HashValue
John	1
Mary	5
Robert	1




```
map.put(“John", "A");
map.put(“Mary", “F");
map.put(“Robert", “B");

map.get(“Mary");// “F”
map.get(“Robert”); // “B”

map.put(“John", “C");
```

Thank you!