# PG4200: Algorithms And Data Structures

# Lesson 06: Hash Maps and Sets

Bogdan Marculescu

# Hash Function

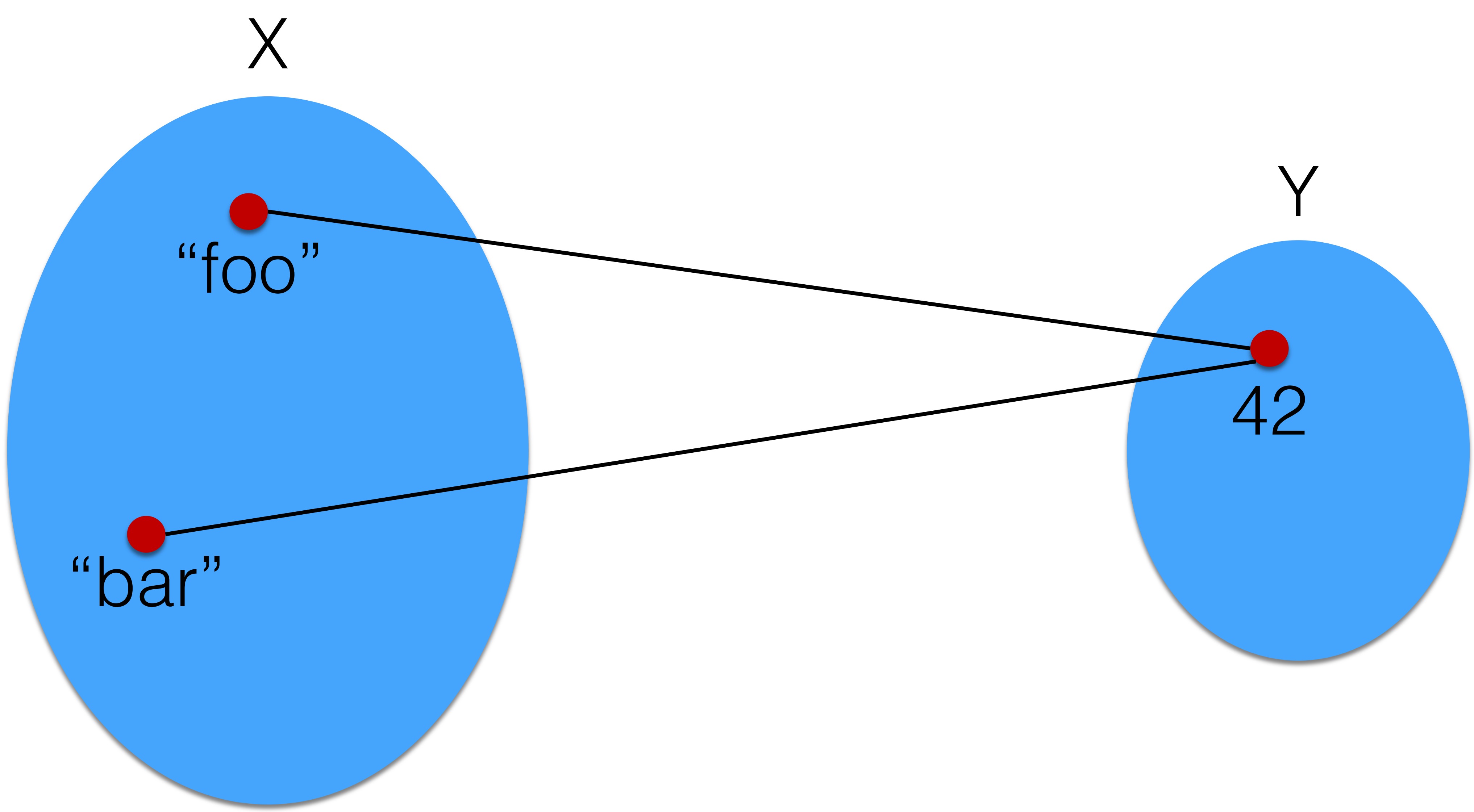
- A function that maps data from an arbitrary size to a specific size
  - eg, mapping strings to a int
- $h(x)=y$ , mapping from domain X to a value in domain Y
- |X| is often much larger than |Y|
  - Note: in mathematics, given a set X, with |X| we represent its cardinality, ie the number of elements in it

# Hash Properties

- *Deterministic*: for a given input *x'*, should always get the same output *y'*
- *Uniform*: mapping from X to Y should be ideally spread uniformly over Y,
  - ie the number of elements in X that map to a specific *y'* should be close to $|X|/|Y|$
- *Performance*: either fast (in this course) or slow (security, eg hashing of passwords)

# Collisions

- If |X| > |Y|, you cannot avoid *h(x')=h(x'')*, two different values in X mapping to the same value in Y
- Ideally, if uniform, no more than |X|/|Y| collisions per element



# Hash Function Examples

- int to int
  - same domain, so same size
- long to int
  - from $2^{64}$ to $2^{32}$ values, which means that for every single int, there are $2^{64}/2^{32} = 2^{32}$, ie 4 billion, longs
- String to int
  - considering that a Java String can be composed of up to 2 billion characters (max length of its internal byte[] array), the space of all possible strings is astronomically huge…

```
public static int hash(int x) {

    return 1;
}
```

```java
public static int hash(int x) {

    return 1;
}
```

- Not particularly good…
- It is in NOT *uniform*, as all values are mapped to the same value 1

```java
public static int hash(int x) {

    return x;
}
```

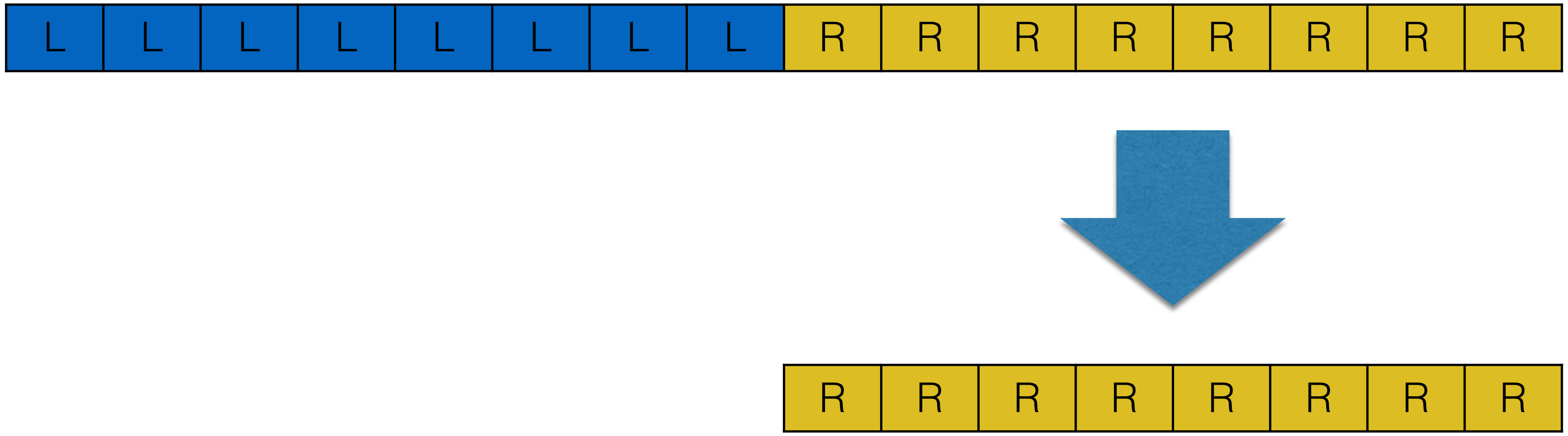
- So called *identity* function
- Technically, it is a valid hash function (eg, deterministic and uniform)
- Potential issue: trivial to revert, ie, knowing output, it is trivial to find the input
  - this would had be a huge problem if we were in the context of **security**, eg hashing of passwords, but not here in this course

```java
public static int hash(int x) {

    return x + 100;
}
```

- Still a valid hashing function
  - the addition overflow for high values is irrelevant here
- But still trivial to revert, eg find input given output
- As in this course we do not deal with security, we are mainly interested in hash functions from larger to smaller sets

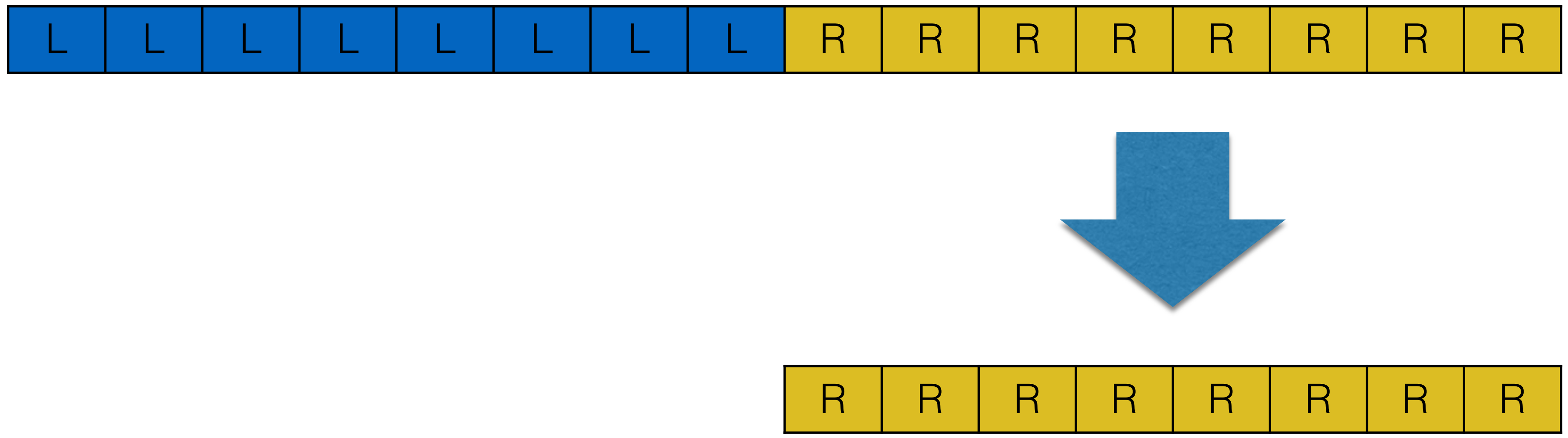
```java
public static int hash(long x) {
    return (int) x;
}
```

- What does it mean to "*cast*" a **long** to a **int**?
- **long** is 64 bits, whereas **int** is just 32
- Somehow, here we lose information




- Ignore first 32 bits in the **long**, and take last 32 to make an **int** out of them

- WARNING: if the first bit in the right part is a 1, then the resulting **int** will be negative
    - as first bit in a **int** is used for the sign




- Issue: any change in a long in its left part will end up in exactly the same hash value
- What if we want to guarantee that a single bit change must result in a different hash?

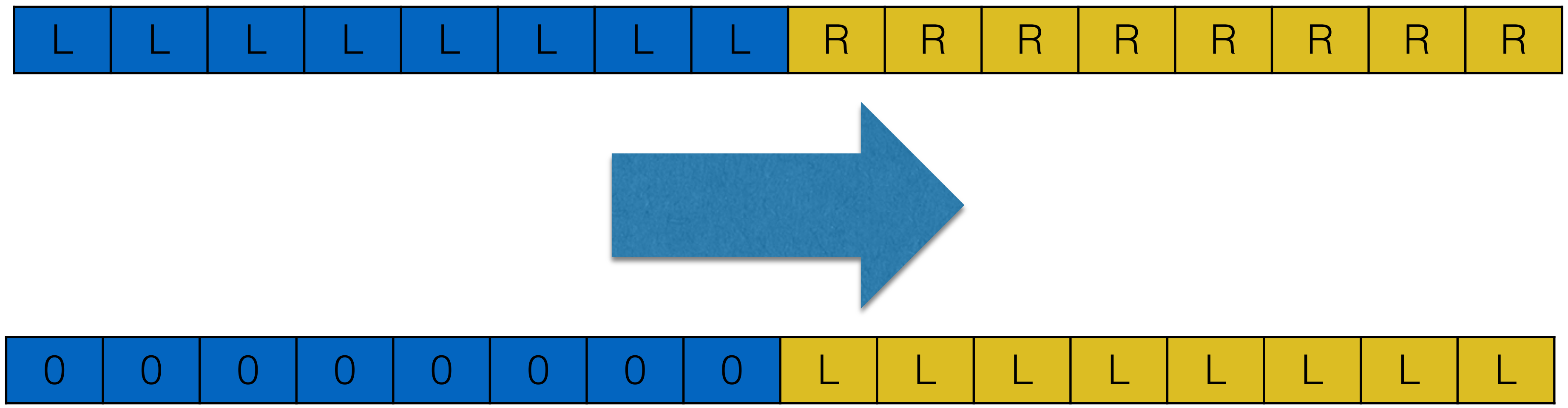
```java
public static int hash(long x) {

    return (int) (x ^ (x >>> 32));
}
```

- If, looking at it, your first reaction is “WTF?!?” do not worry… it is a normal reaction…

- ^ : xor operator
    - WARNING: usually in mathematics the symbol ^ represents an exponent, eg, 2^32 in mathematical notation is $2^{32}$

- >>>: right-shift operator

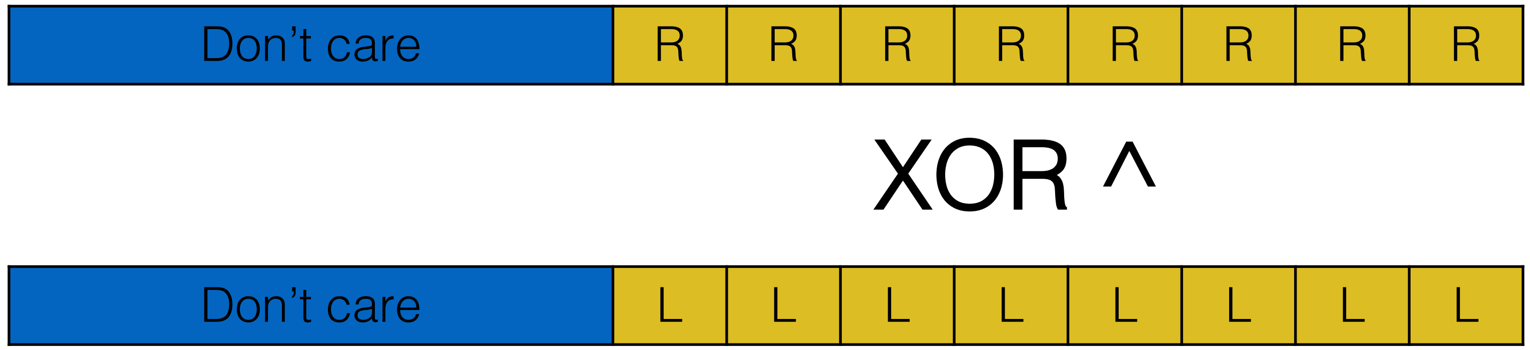
# x >>> 32




- Move all bits to the right by 32 positions
- On the left, empty positions will be filled with 0s
- On the right, old elements are discarded
- Note: there is a difference between “>>>” and “>>”
    - “>>>” ignores the sign, and always fill left part with 0s

# (int) (x ^ (x >>>32))

- Xor ^:
  - 0 ^ 0 = 0
  - 0 ^ 1 = 1
  - 1 ^ 0 = 1
  - 1 ^ 1 = 0




- Changing any single bit in either the left or right part will guarantee that the resulting xor will be different

- Note: changing more than 1 bit does not guarantee it… we are still going to have 4 billion collisions per **int**

- Recall that with **(int)** we discard the left part of the resulting **long** value

```java
public static int hash(String x) {
    int sum = 0;
    for (int i = 0; i < x.length(); i++) {
        sum += x.charAt(i);
    }
    return sum;
}
```

- A **String** could have any size…
  - actually bounded by its internal **byte[]** representation, where sizes of arrays are **int**, so at most 2GB of data
- We can look at each **char** as 16-bit number, and sum all of them
  - don’t care of integer overflow

```java
public static int hash(String x) {
    int sum = 0;
    for (int i = 0; i < x.length(); i++) {
        sum += x.charAt(i);
    }
    return sum;
}
```

- Problem: many small strings will end up with the same hash, especially permutations

- eg, hash(“ab”) = hash(“ba”)

- eg, hash(“B”) = hash(“!!”)
    - B has numeric code 66, whereas ! has code 33

```java
public static int hash (String x) {
    final int delta = 31;

    int sum = 0;
    for (int i = 0; i < x.length(); i++) {
        sum = (delta * sum) + x.charAt(i);
    }
    return sum;
}
```

- Multiply by a constant at each character

- We are still going to have collisions, but less between short strings using Latin letters (which are the ones we usually work with)

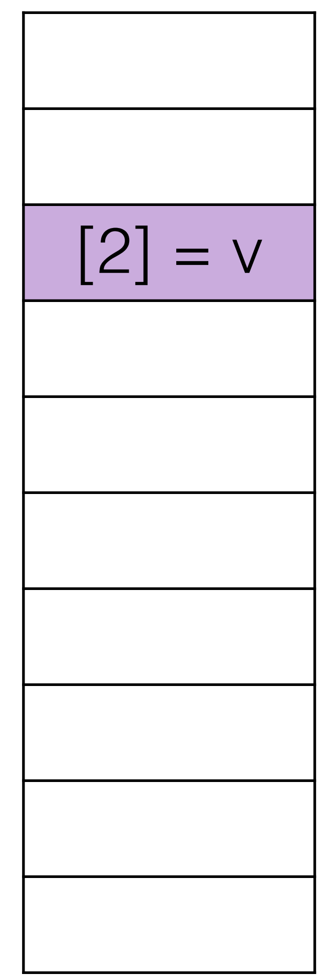
# Hash Maps

- Still a map from a K key to a V value
- No requirement on ordering of K keys, just being able to compute an *hash* of it
- In Java, all objects inherits from *java.lang.Object*, which defines a *hashCode()* method
- Hash code used as an index for an internal array

# Example

- *put("foo", v)*
- *h("foo")=42*
- *h("foo")%10 = 2*
- Benefit: operations (insert/search/etc) have cost due to hash independent of size N of the collection

Internal array buffer of size M=10



# What About Collisions?

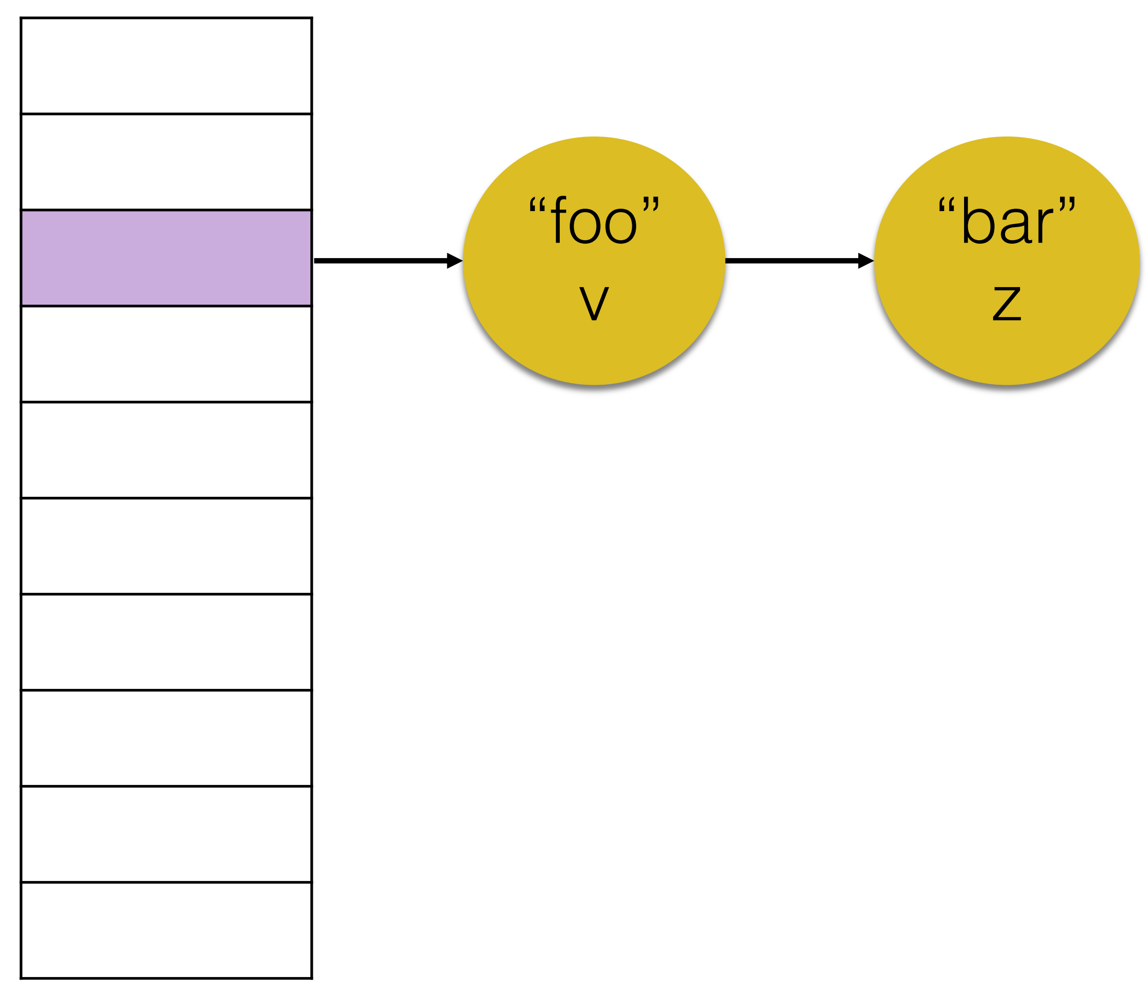
- *put("foo", v)*
- *put("bar", z)*
- *h("foo")=h("bar")*
    - ie, collision due to same hash
- *h("foo")%10 = 2*
- What to do?

Internal array buffer of size M=10

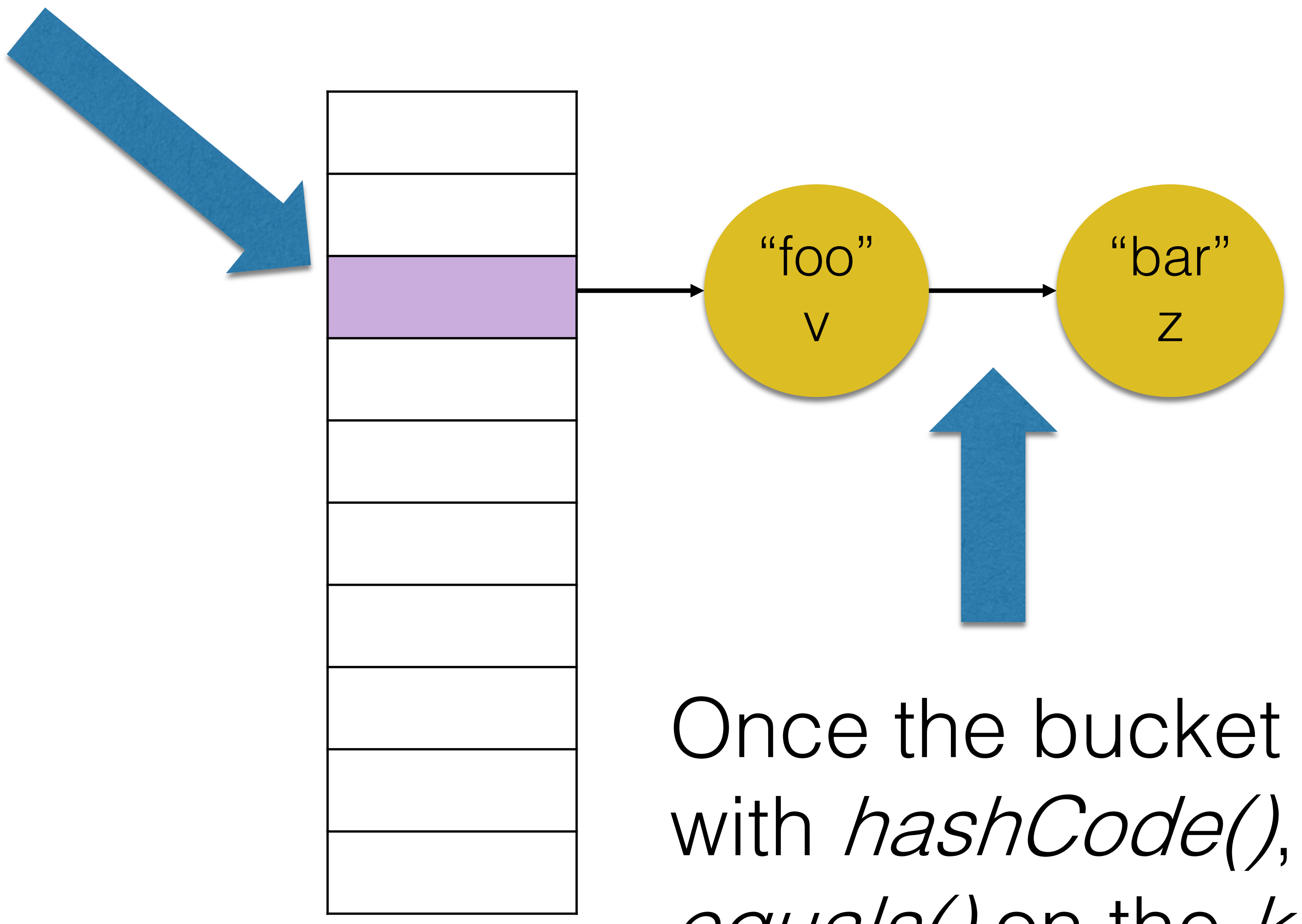
# Different Strategies

- *put("foo", v)*
- *put("bar", z)*
- *h("foo")=h("bar")*
  - ie, collision due to same hash
- Use list at each position sharing same hash
- Nodes containing keys and values

Internal array buffer of size M=10



*hashCode()* computed on the keys to determine their bucket. In this example, assuming *"foo".hashCode()=="bar".hashCode()*, because same bucket. However, *"foo".equals("bar")* is false




Once the bucket is determined with *hashCode()*, we use *equals()* on the *keys* in the list (one at a time), to see if there is a match

# java.lang.Object

- *Object* does define two methods: *hashCode()* and *equals()*
- Those methods will depend on the internal fields of the object
- *Important*: if two objects are equals, then they **MUST** have same hash code
    - *A.equals(B)* implies *A.hashCode()==B.hashCode()*
    - The vice-versa is not necessarily true, ie *A.hashCode()==B.hashCode()* does not imply *A.equals(B)*, although that could happen
- What if constraint is not satisfied? Expect weird bugs when using maps and sets…

# Cost

- Worst case: O(N) if all elements end up in same "bucket" (ie same value for *h()%M*), the map would be equivalent to a list
  - operations to search on list would be O(N), albeit insert would be O(1)

- But, if M large enough compared to N, and hash function is uniform enough, you can have a O(1) cost in many cases
  - even if you have some collisions, it will not be a problem, as you would have a small number of elements in the list

# Hash or RBT?

- Hash Maps is the most popular and widely used
- If you know how much data you ll insert at most, can choose a good large enough M
- So in most cases, we are in *O(1)* Hash vs *O(log N)* RBT
- But Hash can be *O(N)* in worst case, vs RBT guarantees *O(log N)* in all cases
  - eg, in critical systems where you MUST guarantee a response within a certain amount of time, might want to use RBT
- Hash does not need ordering of keys

# Set

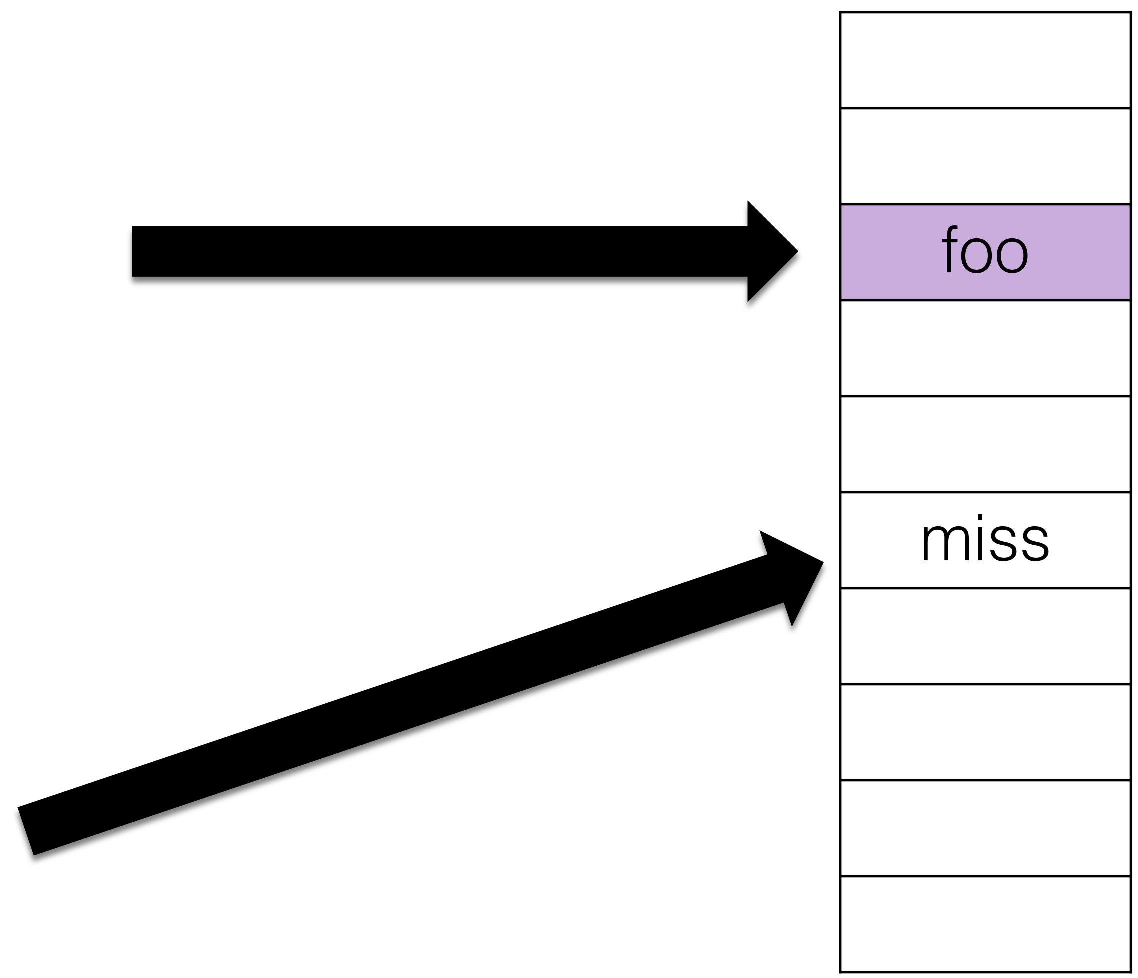
- In mathematics, a *set* is a collection of elements where:
  - 1) *ordering is not important*: ie {1,2,3} is equivalent to {2,3,1}
  - 2) *no repetitions*: ie {1,2} is the same as {2,1,1,2,2,1,1,2,1}
- How to implement a Set in Java?
- Easy: use an internal *Map<K,V>* were your values in the set are the keys *K*, and you just ignore the values *V*

# Keys and Immutability

- *Immutable Object*: an object whose state cannot be changed once created
- Example: Strings are immutable
  - eg, concatenation with + and methods like *toUpperCase()* and *substring()* do NOT change the String, but rather *create* a NEW one
- Keys in a Map/Set **MUST** be *immutable*… why?

# Different Hash

```
Foo foo = new Foo();

set.add(foo);

assertTrue(set.contains(foo));

// h(foo) = 42  ,  42 % M = 2

foo.setSomeVariable(…);

// h(foo) = 55  ,  55 % M = 5

assertFalse(set.contains(foo));
```



# Using Maps and Sets

- Should only use a *Set* for **immutable** types
- What if you need a collection of mutable types *<X>*?
  - in most cases, creating a *Set<X>* would be wrong!
- Option 1: rather use a list, eg *List<X>*
  - however, it would allow duplicates
- Option 2: use a map *Map<K,X>* where the key is an immutable field derived from *X*
  - eg, if mutable *User*, *map.put(user.getId(), user)*, where the id could be a String (recall strings are immutable)

# Creating an Immutable Object

```
public class UserImmutable {

    private final String name;
    private final String surname;
    private final int id;
```

- The first step is to make each field "*final*"
  - this means that they cannot be modified
- This is enough for primitive values (eg, **int** and **double**) but not for objects!
  - those must be immutable as well!!!

# “final” Limitation

```
final int[] foo = {1,2,3};
~~foo = {4,5};~~  // would not compile
foo[0] = 42; // no problem!
```

- A “final” reference cannot be modified, but you can modify the referenced state!
- Note: no problem for “*final String*”, as String is immutable
- So, if you have a mutable object M inside an immutable one X, must guarantee that no operation in X can change the state of M once initialized

# Modifying an Immutable Object

```
public UserImmutable withName(String s){
    return new UserImmutable(s, surname, id);
}
```

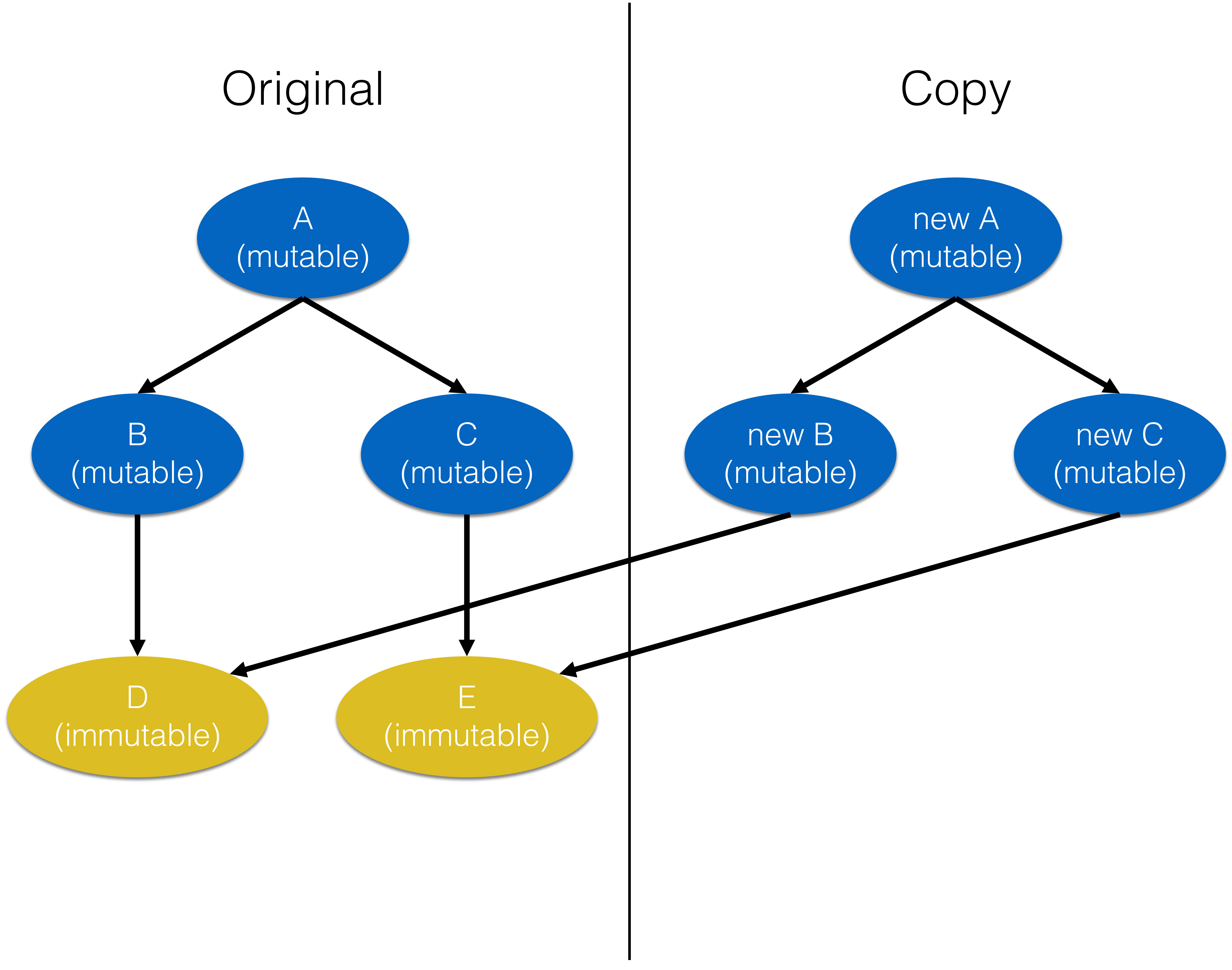
- Technically, you cannot modify an immutable object
- However, can return a new instance with the modified state
- Recall all methods in String return a new copy of the String, as String is immutable

# Object Copy

- When copying an object, it is a huge difference on whether its fields are **mutable** or **immutable** objects

- **Immutable**: can just copy the reference, ie, actual objects can be shared
  - aka "*shallow copy*"

- **Mutable**: must make a **new** complete copy, with **new** memory allocation on the heap
  - aka "*deep copy*"



Original
Copy
A
(mutable)
B
(mutable)
C
(mutable)
D
(immutable)
E
(immutable)
new A
(mutable)
new B
(mutable)
new C
(mutable)

# Homework

- Study Book Chapter 3.4 and 3.5
- Study code in the *org.pg4200.les06* package
- Do exercises in *exercises/ex06*
- Extra: do exercises in the book