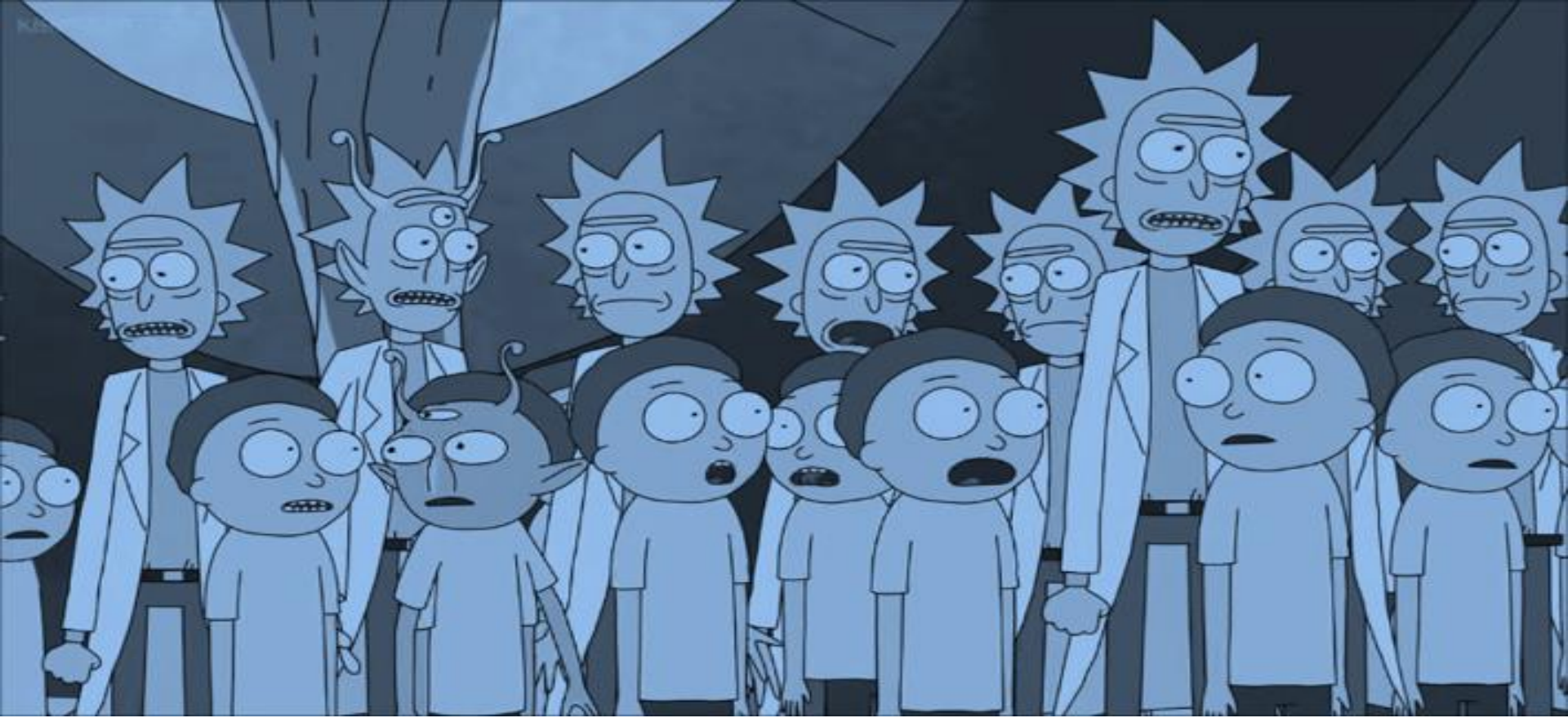


Advanced Object-Oriented Programming

CPT204 – Lecture 12

Erick Purwanto

Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 12

Hash Table, Comparator, Concurrency, Thread Safety

Welcome !

- Welcome to Lecture 12 !
- In this lecture we are going to
  - learn about hash codes and hash functions
  - learn about a data structure called Hash Table
  - learn about comparator
  - learn about anonymous class
  - learn about concurrency

Admin Stuff : Important Dates

Semester 2		17	18	19	20	21	22	23	Week 12
		24	25	26	27	28	29	30	Week 13
	June	31	1	2	3	4	5	6	Week 14
		7	8	9	10	11	12	13	Examination days;
		14	15	16	17	18	19	20	Dragon Boat Day; Marking

- Tuesday, May 25 — Task Sheet released
- Friday, June 4, 10:00 – 11:00 — Part A, B Online Submission
- Friday, June 4, Lab group schedule — Part C Lab Test
- Monday, June 7, 14:00 – 16:00 — Final Exam in Labs

Part 1 : Higher Order Functions, Comparator, Hash Tables

- In the first part of the lecture, we will
  - continue our discussion last week on `equals` and `hashcode`
  - continue our discussion last week on `Comparable`
  - introduce its closely related comparison tool called `Comparator`
  - learn about Higher Order Function technique used to create `Comparator`

- Importantly, we will also build a hash-based `Set`
  - and introduce techniques you will use in lab to build hash-based `Map`

The Object Contract

Review

- The specification of the `Object` class is so important that it is often referred to as the **Object Contract**
  - The contract can be found in the method specifications for the `Object` class
  - Here we will focus on the contract for `equals`
- When you override the equals method, you must adhere to its general contract, that states:
  - equals must define **an equivalence relation** — that is, a relation that is reflexive, symmetric, and transitive
  - equals must be **consistent**: repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified
  - for a **non-null** reference `x`, `x.equals(null)` should return false
  - **hashCode** must produce the same result for two objects that are deemed equal by the `equals` method

will talk about this

Hash Tables (1)

- Last week we discussed about the `Object` contract
- To understand the part of the contract relating to the **hashCode method**, we now discuss about how hash tables work
  - two very common collection implementations, `HashSet` and `HashMap`, use a hash table data structure, and depend on the `hashCode` method to be implemented correctly for the objects stored in the set and used as keys in the map
- A **hash table** is a representation for a mapping: an abstract data type that maps keys to values
  - hash tables offer constant time lookup, so they tend to perform better than trees or lists
  - keys don't have to be ordered, or have any particular property, except for offering `equals` and `hashCode`

Hash Tables (2)

- Here's how a hash table works
  - It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted
  - When a key and a value are presented for insertion, we compute **the hash code** of the key, and *convert* it into an index in the array's range (e.g., by a modulo division)
  - The value is then inserted at that index

- The *rep invariant of a hash table* includes the fundamental constraint that a key can be found by starting from the slot determined by its hash code

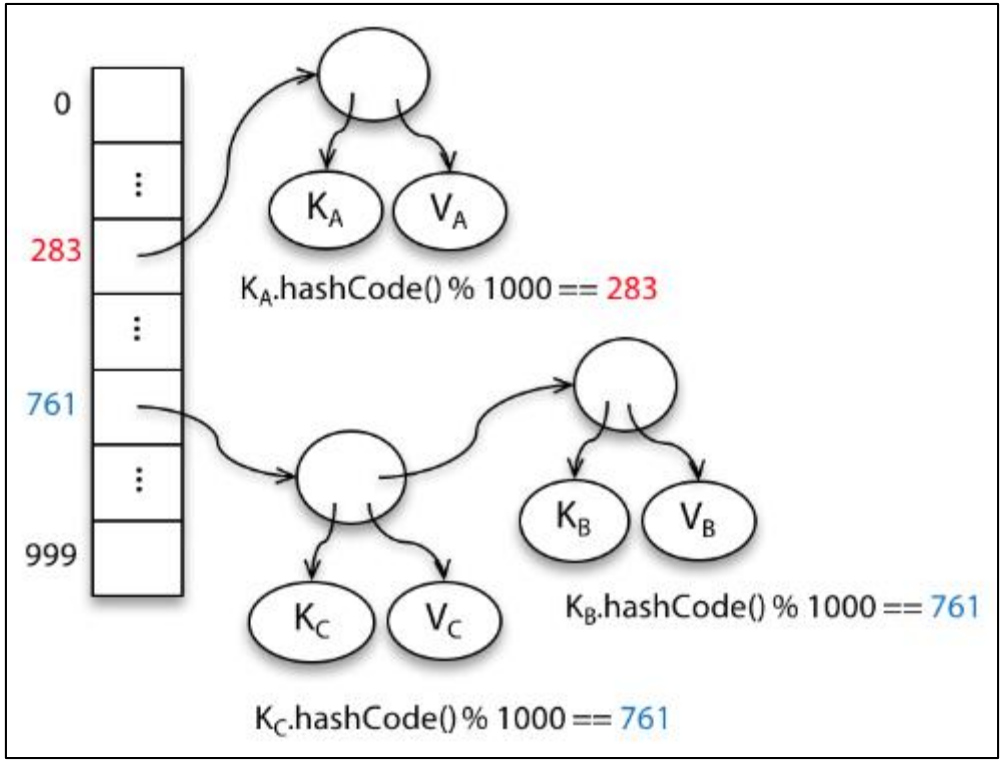
Hash Tables (3)

- Hash codes are designed so that the keys will be spread evenly over the indices
  - But occasionally a conflict occurs, and two keys are placed at the same index
  - So rather than holding a single value at an index, a hash table actually holds *a list of key/value pairs*, usually called **a hash bucket**

- A key/value pair is implemented in Java simply as an object with two fields
  - On *insertion*, you add a pair to the list in the array slot determined by the hash code
  - For *lookup*, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key

Hash Tables (4)

- An example of a hash table:



Breaking Hash Tables (1)

- Now it should be clear why the `Object` contract requires *equal objects to have the same hash code*
  - if two equal objects had *distinct* hash codes, they might be placed in *different* slots
  - so if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may *fail*

- `Object's` **default `hashCode()`** implementation is consistent with its default `equals()`:

```
public class Object {
  . . .
  public boolean equals(Object that) { return this == that; }
  public int hashCode() { return /* the memory address of this */; }
}
```

  - For references `a` and `b`, if `a == b`, then the address of `a ==` the address of `b`, so the `Object` contract ***is*** satisfied

Breaking Hash Tables (2)

- But immutable objects need a different implementation of `hashCode()`
- For `Duration`, since we *haven't* overridden the *default* `hashCode()` yet, we're currently breaking the `Object` contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```

- `d1` and `d2` are `equal()`, but they have *different* hash codes
  - so we need to fix that

Breaking Hash Tables (3)

- A simple and drastic way to ensure that the contract is met is for `hashCode` to always return *some constant value*, so every object's hash code is the same
  - this satisfies the `Object` contract, but it would have a disastrous performance effect, since every key will be stored in the *same* slot, and every lookup will *degenerate* to a linear search along a long list

- The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality
  - usually by calling the `hashCode` method of ***each component***
  - and then ***combining*** these, using a few arithmetic operations

Breaking Hash Tables (4)

- For `Duration`, this is easy because the abstract value of the class is already an integer value:

```
@Override
public int hashCode() {
    return getLength();
}
```

- Our reference textbook, **Effective Java**, explains this issue in more detail, and gives some strategies for writing decent hash code functions and the advice is summarized in a good StackOverflow post: https://stackoverflow.com/questions/113511/best-implementation-for-hashcode-method-for-a-collection
- Recent versions of Java now have a utility method `Objects.hash()` that makes it easier to implement a hash code involving multiple fields

Breaking Hash Tables (5)

- Note, however, that as long as you satisfy the requirement that equal objects have the same hash code value, then the particular hashing technique you use doesn't make a difference to the correctness of your code
  - it may affect its performance, by creating unnecessary collisions between different objects, but *even* a poorly-performing hash function is better than one that breaks the contract

- Most crucially, note that if you *don't override* `hashCode` at all, you'll get the one from `Object`, which is based on the address of the object
  - if you have overridden equals, this will mean that you will have *almost certainly* violated the contract

Breaking Hash Tables (6)

- So as a general rule:
  - **always override `hashCode` when you override `equals`**

- It has happened that a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hash`**`C`**`ode` as `hash`**`c`**`ode` !
  - this created a new method that didn't override the `hashCode` method of `Object` at all, and strange things happened
  - use **`@Override`** !

Equality of Mutable Types

Review

Now about mutable objects, recall our definition last week: two objects are equal when they cannot be distinguished by observation

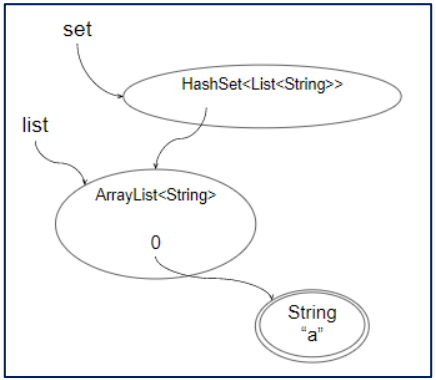
With mutable objects, there are two ways to interpret this:

- when they cannot be distinguished by observation t*hat doesn't change the state of the objects*, i.e., by calling *only* observer, producer, and creator methods
  This is called **observational equality**, since it tests whether the two objects *look* the same in the current state of the program
- when they cannot be distinguished by *any* observation, even state changes
  This interpretation allows calling *any* methods on the two objects, including mutators
  This is called **behavioral equality**, since it tests whether the two objects will *behave* the same, in this and all future states

Equality and hashCode Mutable Types Bug (1)

- For immutable objects, observational and behavioral equality are identical, because there aren't any mutator methods
  - For mutable objects, it's tempting to implement strict observational equality
- Java uses **observational equality** for most of its mutable data types, in fact
  - if two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal
  - but using observational equality leads to subtle bugs and in fact allows us to easily break the rep invariants of other collection data structures
- Suppose we make a `List`, and then drop it into a `HashSet`:

```
List<String> list = new ArrayList<>();
list.add("a");
Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```



In-Class Quiz 1

- We can check that the set contains the list, and it does:

```
set.contains(list) → true
```

- But now we mutate the list:

```
list.add("goodbye");
```

- What is the output of :

```
set.contains(list) → ?
```

- And, what is the output of:

```
for (List<String> l : set) {
    set.contains(l) → ?
}
```

Select one:

- true
- false
- true, true
- false, true
- true, false
- false, false

Equality and hashCode Mutable Types Bug (2)

- We can check that the set contains the list, and it does:

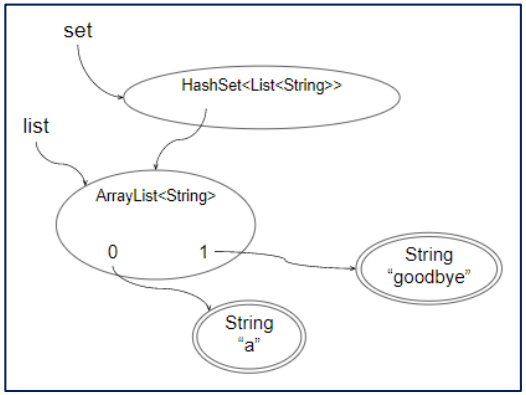
```
set.contains(list) → true
```

- But now we mutate the list:

```
list.add("goodbye");
```

- And it in the set!

```
set.contains(list)
```



- It's worse than that, in fact: when we **iterate** over the members of the set, we **still find** the list , but **`contains()`** says it's **not there**!

```
for (List<String> l : set) {
    set.contains(l)
}
```

- If the set's own `iterator` and its own `contains()` method ***disagree*** about whether an element is in the set, then the set clearly is broken

Equality and hashCode Mutable Types Bug (3)

- What's going on?
- `List<String>` is a mutable object, and in the standard Java implementation of collection classes like `List`, mutations affect the result of `equals()` and `hashCode()`
- When the list is first put into the `HashSet`, it is stored in the hash bucket *corresponding* to its `hashCode()` result at that time
- When the list is subsequently *mutated*, **its `hashCode()` changes**, but `HashSet` **doesn't** realize it should be moved to a different bucket
  - so it can *never* be found again
- When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key
- Here's a telling quote from the specification of `java.util.Set`:
  *Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is **not** specified if the value of an object is **changed** in a manner that affects equals comparisons while the object is an element in the set*

Equality and hashCode Mutable Types Bug (4)

- The Java library is unfortunately *inconsistent* about its interpretation of `equals()` for mutable classes
  - Collections like `List`, `Set`, and `Map` use observational equality, but other mutable classes like `StringBuilder` and arrays use behavioral equality
- The lesson we should draw from this example is that `equals()` **should implement behavioral equality** for mutable objects
  - In general, that means that two references should be `equals()` if and only if they are aliases for the same object
  - So mutable objects should just inherit `equals()` and `hashCode()` from `Object`
- For clients that need a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a new method, e.g., `similar()`

The Final Rule for equals() and hashCode() (1)

- For immutable types:
  - `equals()` should compare abstract values
    - this is the same as saying `equals()` should provide behavioral equality
  - `hashCode()` should map the abstract value to an integer

- So immutable types **must** override both `equals()` and `hashCode()`

The Final Rule for equals() and hashCode() (2)

- For mutable types:
  - `equals()` should compare references, just like `==`
    - Again, this is the same as saying `equals()` should provide behavioral equality
  - `hashCode()` should map the reference into an integer
- So mutable types **should not** override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object`
- Java ***doesn't*** follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above

in this course, also, sometimes we still use observational equality for mutables objects

Autoboxing and Equality (1)

- One more instructive pitfall in Java
- We've talked about primitive types and their object type equivalents — for example, `int` and `Integer`
- The object type implements `equals()` in the correct way, so that if you create two Integer objects with the same value, they'll be `equals()` to each other:

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y)       // returns true
```

- But there's a subtle problem here; `==` is overloaded :
  For *reference* types like `Integer`, it implements **referential equality**:

```
x == y            // returns false
```

  but for *primitive* types like `int`, `==` implements **behavioral equality**:

```
(int)x == (int)y  // returns true
```

Autoboxing and Equality (2)

- So you **can't** really use `Integer` interchangeably with `int`
- The fact that Java ***automatically converts*** between `int` and `Integer` (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs!
- You have to be aware what the compile-time types of your expressions are
- Consider this:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
a.put("c", 130);    // put ints into the map
b.put("c", 130);
```

Autoboxing : when an `int` 130 is put into the map, it is automatically *boxed* up into a new `Integer` object

Autoboxing and Equality (3)

- So you **can't** really use `Integer` interchangeably with `int`
- The fact that Java ***automatically converts*** between `int` and `Integer` (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs!
- You have to be aware what the compile-time types of your expressions are
- Consider this:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
a.put("c", 130);     // put ints into the map
b.put("c", 130);
int i = a.get("c")  // what do we get out of the maps?
```

Autounboxing : `get()` return an `Integer` object representing 130,
but after that, if we assign the `get()` result to an `int` variable,
the assignment automatically *unboxes* the `Integer` object into `int` value 130

Fast Searching

- In the next part of this lecture, we will look at a data structure for ***efficiently searching for the existence of items*** within the data structure

- Yes, it is the **Hash Table**
  - we actually have learned to use it in Week 2 to do problem solving, as we learn and practice unit testing
  - we want to use hash code to cleverly to build a data structure

I personally know a couple friends who were asked about this very topic in their coding interview!

Using an Array

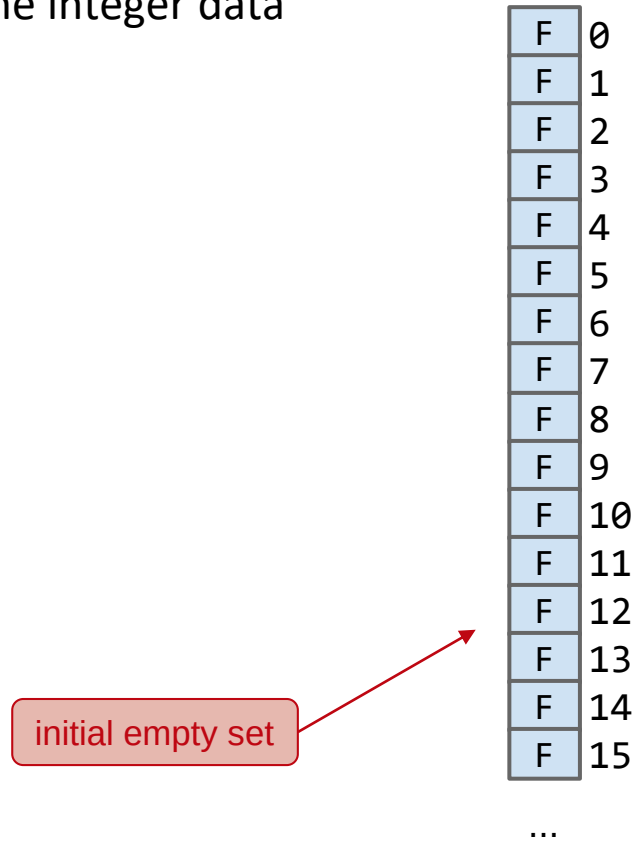
- Recall that in lab, you have created **`ARSet`**, an array-based set, that can do search
  - however, the searching method `contains()` cost is O(N)
    - in the worst case, you have to search the whole array
  - we want to do much better than that, and yet *still using an array* as the rep !

- Let us start simple by *only* storing *integer* data
  - and call our set implementation **`INTSet`**
  - remember, our goal is to *make searching as fast as possible!*

INTSet

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false

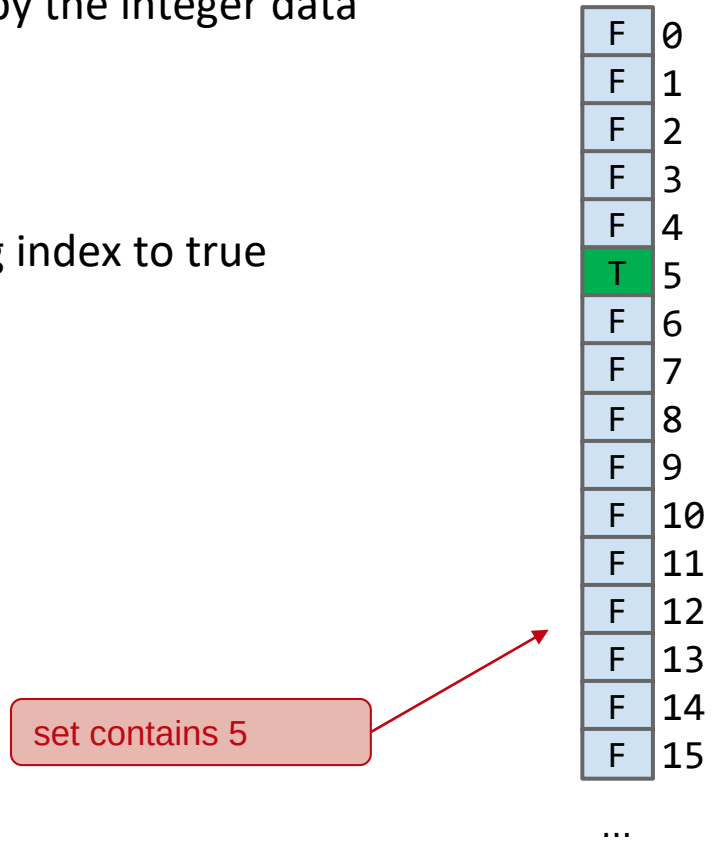
```
INTSet set;
set = new INTSet();
```



INTSet

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

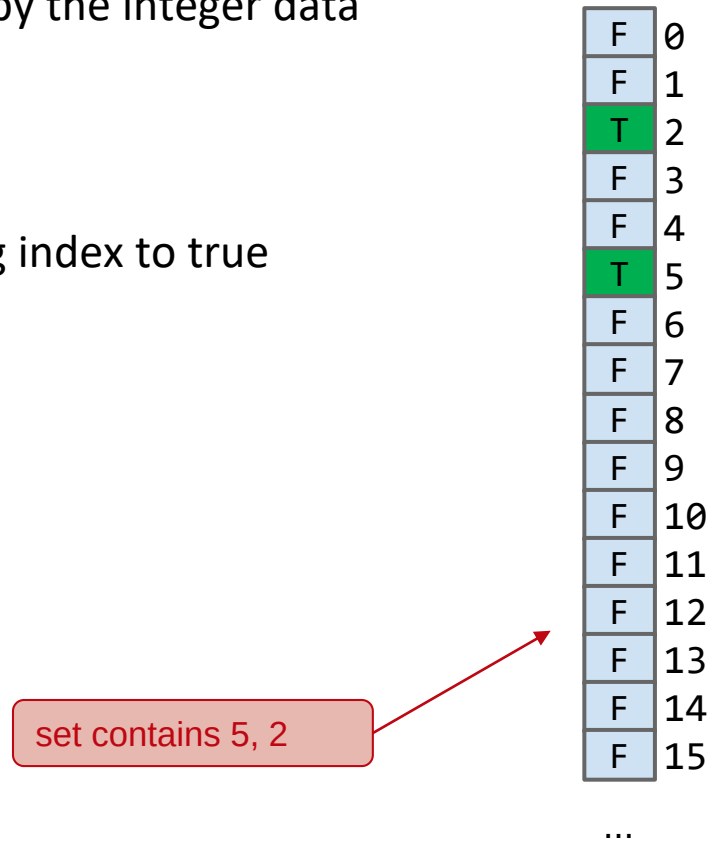
```
INTSet set;
set = new INTSet();
set.add(5);
```



INTSet

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

```
INTSet set;
set = new INTSet();
set.add(5);
set.add(2);
```

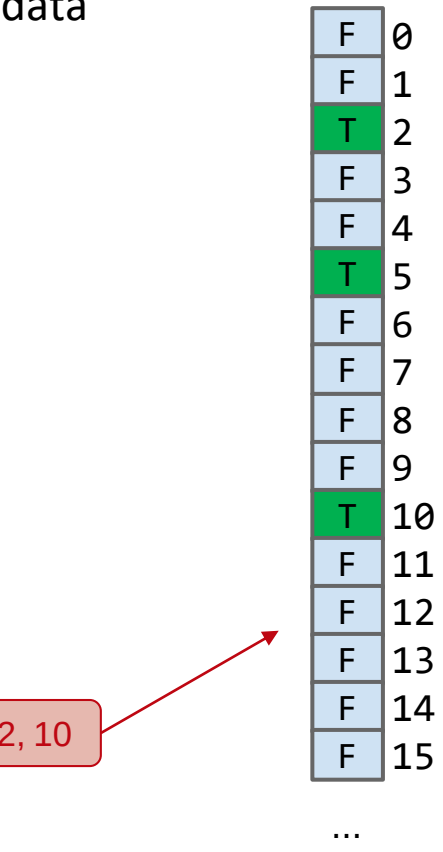


INTSet

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

```
INTSet set;
set = new INTSet();
set.add(5);
set.add(2);
set.add(10);
```

`contains(i)` simply check whether value at index i is T or F



set contains 5, 2, 10

INTSet Implementation

- The implementation is very simple:

```
public class INTSet {
   private boolean[] items;

   public INTSet() {
       items = new boolean[2000000000];
   }

   public void add(int i) {
       items[i] = true;
   }

   public boolean contains(int i) {
       return items[i];
   }
}
```

roughly the number of integers

since accessing elements in an array is constant time, these operations = O(1) time

INTSet Drawbacks

- The are some obvious problems with this implementation:

```
public class INTSet {
    private boolean[] items;

    public INTSet() {
        items = new boolean[2000000000];
    }

    public void add(int i) {
        items[i] = true;
    }

    public boolean contains(int i) {
        return items[i];
    }
}
```

1. extremely wasteful of memory! even if you only want to store a few integers you still need two billion booleans

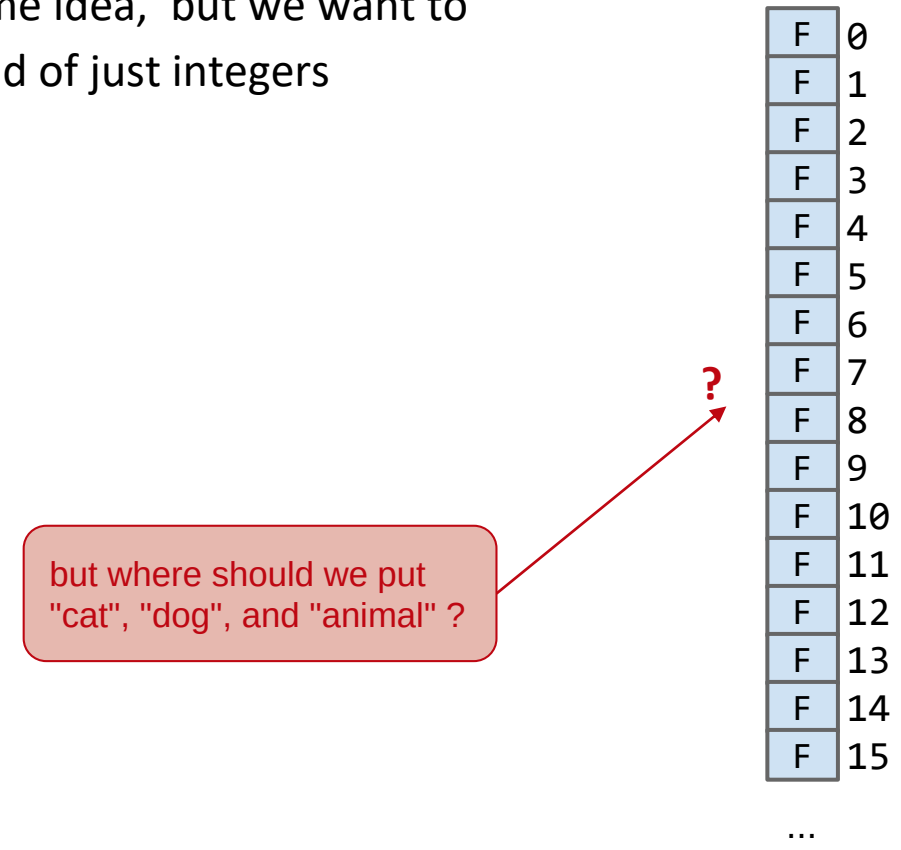
2. you can only store integers need a way to generalize

essentially, when we can solve those two issues, we will get ourselves a hash table !

ENGSet

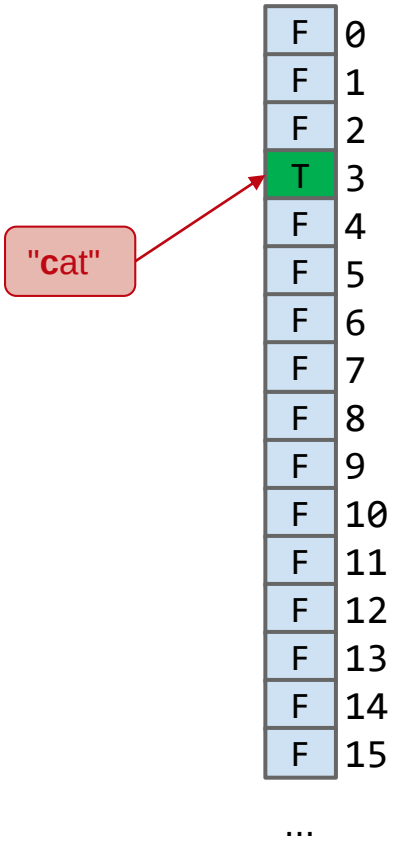
- Now suppose we want to use the same idea, but we want to store **lowercase English words** instead of just integers
- We want a set `ENGSet` that can:

```
ENGSet set;
set = new ENGSet();
set.add("cat");
set.add("dog");
set.add("animal");
```



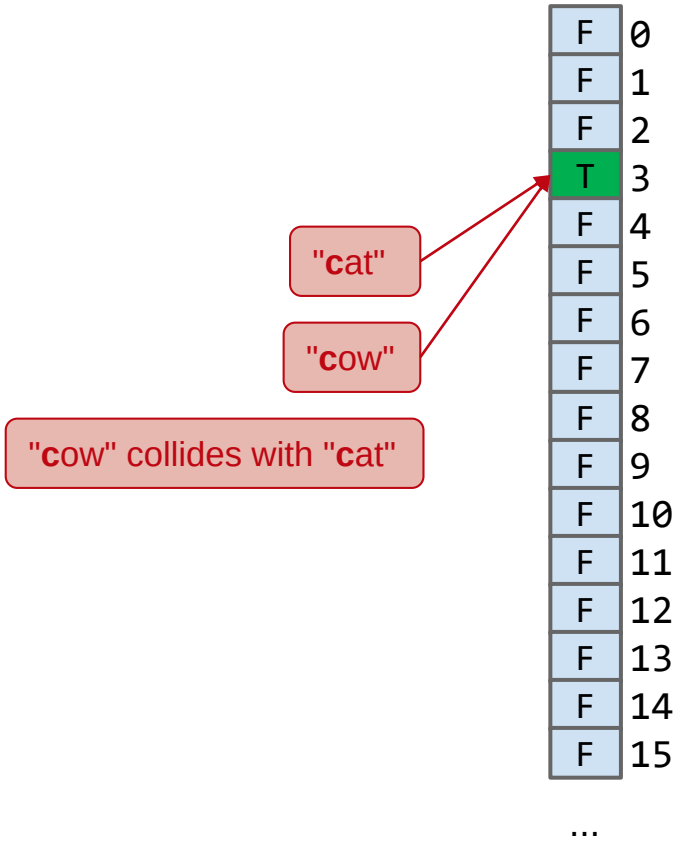
Add into ENGSet (1)

- Suppose we want to `add("cat")`
  - what is the index for "cat" ?
  - idea: use the first letter as index
    a = 1, b = 2, **c = 3**, …, z = 26



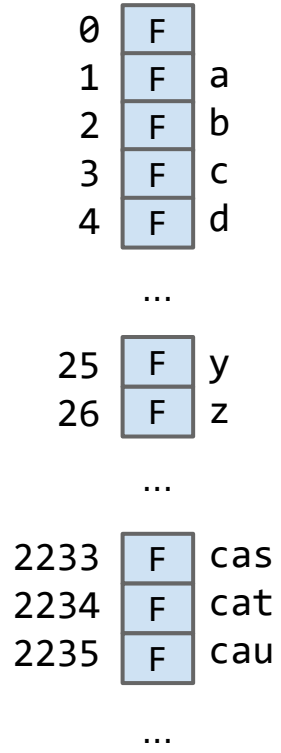
Add into ENGSet (2)

- Suppose we want to `add("cat")`
  - what is the index for "cat" ?
  - idea: use the first letter as index
    a = 1, b = 2, **c = 3**, …, z = 26
  - however, contains("**c**ow") → T
  - we have **a collision** !



Add into ENGSet (3)

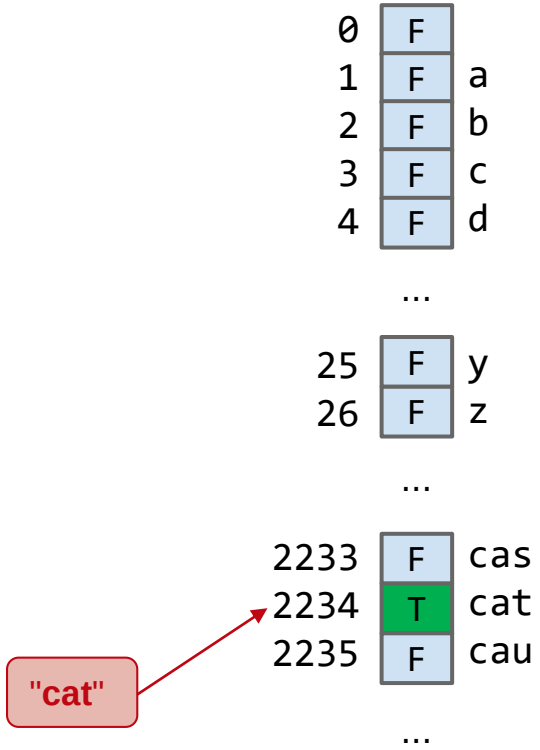
- Suppose we want to `add("cat")` with no collision
  - to avoid collision, idea: use **all** letters by multiplying each by a power of 27
  - 27 because there are 26 letters in English alphabet, and multiply by powers like a decimal number in base 10, e.g. $521_{10} = (5 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$



Add into ENGSet (4)

- Suppose we want to `add("cat")` with no collision
  - to avoid collision, idea: use **all** letters by multiplying each by a power of 27
  - 27 because there are 26 letters in English alphabet, and multiply by powers like a decimal number in base 10, e.g. $521_{10} = (5 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$
  - so, the index of "cat" is $(3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234$

since we pick a base ≥ 26, it is guaranteed to give each lowercase English word a unique number! we will have **no** collision!



Convert English String to Integer

- Method `letterNum` converts letter to number, and `englishToInt` computes the index

```
/* Converts ith character of String to a letter number.
 * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
private static int letterNum(String s, int i) {
    int ithChar = s.charAt(i);
    if ((ithChar < 'a') || (ithChar > 'z'))
        { throw new IllegalArgumentException(); }
    return ithChar - 'a' + 1;
}

private static int englishToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i++) {
        intRep = intRep * 27;
        intRep = intRep + letterNum(s, i);
    }
    return intRep;
}
```

in Java, char 'a' is equivalent to 97
this will give 1 to 'a', 2 to 'b', …

ENGSet Implementation

- Similar to `INTSet`, simply needs to compute the index first

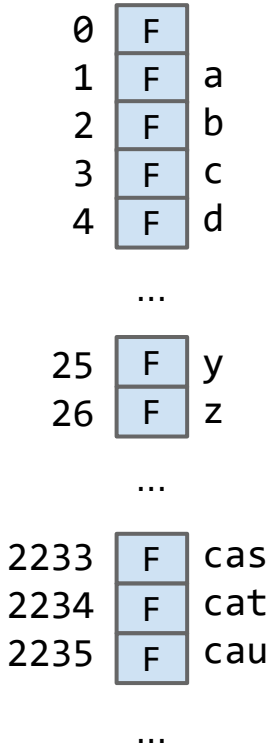
```
public class ENGSet {
   private boolean[] items;

   public ENGSet() {
      items = new boolean[2000000000];
   }

   public void add(String s) {
      items[englishToInt(s)] = true;
   }

   public boolean contains(String s) {
      return items[englishToInt(s)];
   }
}
```

compute index of s

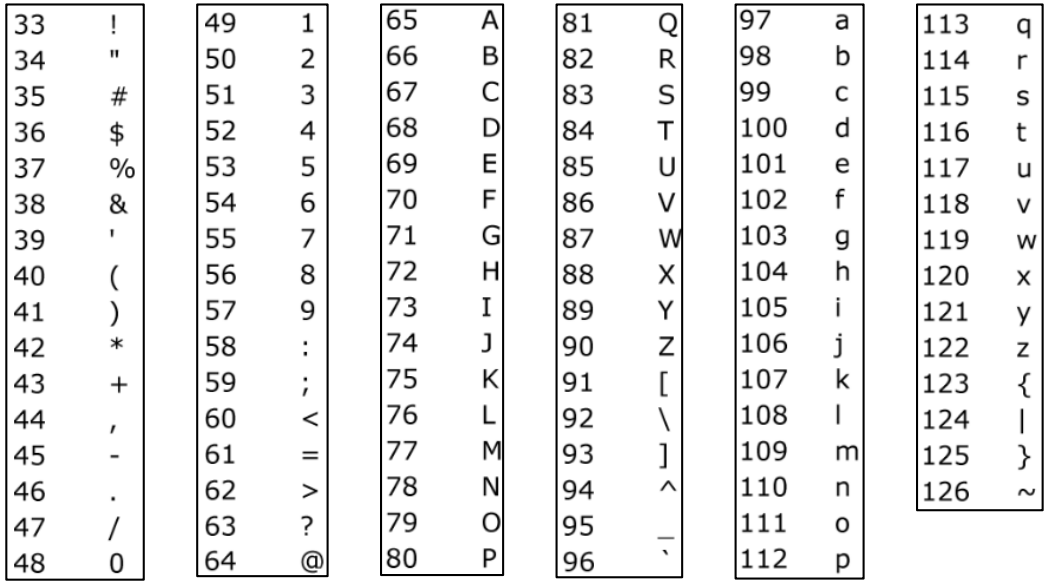


ASCII Characters

- In addition to lowercase letter, what if we want to store uppercase? numbers? special chars?
- We can use ASCII encoding value as the index
  - each char is assigned a value between 0 and 127: `char c = 'a'` is equiv with `char c = 97` in Java

characters 33 - 126 are printable

the others do other things, like the new line character



33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@

65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P

81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`

97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p

113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

ASCII-based SET

- Maximum possible value for english-only text including punctuation is 126
- We can use 126 as our base in order to ensure unique values for all possible strings
- For examples:
  - $bee_{126} = (98 \times 126^2) + (101 \times 126^1) + (101 \times 126^0) = 1{,}568{,}675$
  - $2pac_{126} = (50 \times 126^3) + (112 \times 126^2) + (97 \times 126^1) + (99 \times 126^0) = 101{,}809{,}233$

ASCII-based SET Implementation

- We can use the following method to convert the index:

```
public static int asciiToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i++) {
        intRep = intRep * 126;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

but what we want store even more characters? e.g., Chinese characters?

Unicode Characters

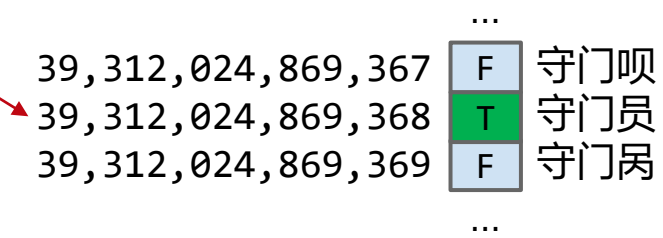
- We can also use Unicode encoding value, also supported in Java
  - for example, `char c = '鳌'` is equivalent to `char c = 40140`

just some tiny part,
the table is too big to display

Unicode-based SET

- The largest possible value for Chinese characters is 40,959
  - (probably more based on the most recent version of Unicode that is continuously updated)
- So we would need to use that number as our base if we want to have a *unique* representation for all possible strings of Chinese characters
- For example:

$$守门员_{40959} = (23432 \times 40959^2) + (38376 \times 40959^1) + (21592 \times 40959^0) =$$
$$= 39{,}312{,}024{,}869{,}368$$



Integer Overflow

- However, it is not possible to have that large of an integer index in Java
- In Java, the largest possible integer is 2,147,483,647
  - If you go over this limit, you **overflow**, starting back over at the smallest integer, which is -2,147,483,648

```
int x = 2147483647;
System.out.println(x);
System.out.println(x + 1);
```

→ 2147483647
→ -2147483648

Overflow causes Invalid Index and Collision

- Because Java has a maximum integer, we may not get the number we expect
  - With base 126, we will run into overflow *even* for short strings
    For example: $\texttt{omens}_{126}$= 28,196,917,171
    which is much greater than the maximum integer,
    and `asciiToInt('omens')` will give us -1,867,853,901 instead
  - Overflow can result in collisions, causing incorrect answers

```
public void testOverflow() {
    ENGSet set = new ENGSet();
    set.add("melt banana");
    set.contains("subterrestrial anticosmetic");
    // asciiToInt for these strings is 839099497
}
```

→ `true`

Hash Code

- The term for the index/number we're computing is **hash code**
- According to Wolfram Alpha, a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
  - here, our target set is the set of Java integers, which is of size 4294967296
  - thus, there can only be 4294967296 possible hash codes

Pigeonhole Principle

there are 9 boxes and 10 pigeons, no matter how you arrange the pigeons, it is guaranteed that at least 1 box is going to contain more than 1 pigeon

- Pigeonhole principle tells us that if there are *more* than 4294967296 possible items, *multiple* items will share the *same* hash code
  - there are more than 4294967296 planets
    - Each has `mass`, `xPos`, `yPos`, `xVel`, `yVel`, `name`
  - there are more than 4294967296 strings
    - `"one"`, `"two"`, …, `"nineteen quadrillion"`, …

- Thus, **collisions are inevitable** !

Two Fundamental Challenges

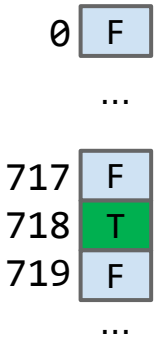
Two fundamental challenges:

- How do we resolve the hash code collisions?
  - let us call this **collision handling**
- How do we compute a hash code for *arbitrary* objects?
  - let us call this **computing a hash code**

Handling Collision

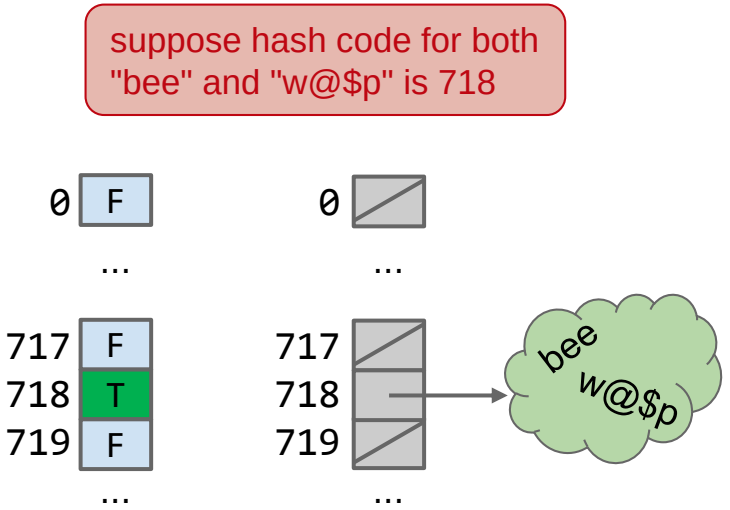
- Suppose N items have the same hash code h

suppose hash code for both "bee" and "w@$p" is 718



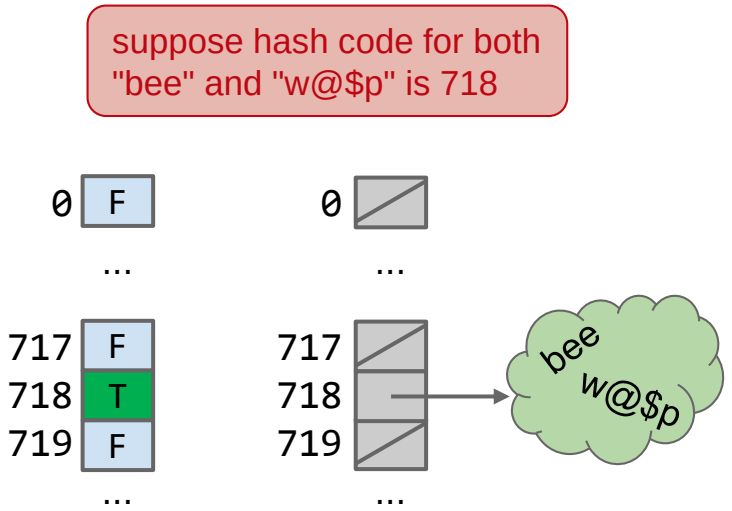
Handling Collision

- Suppose N items have the same hash code h
  - instead of storing `true` in position h, store a **bucket** of these N items at position h



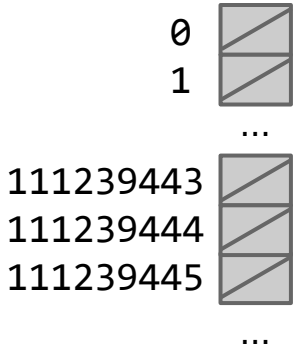
Handling Collision

- Suppose N items have the same hash code h
  - instead of storing `true` in position h, store a **bucket** of these N items at position h
- How to implement a bucket?
  - use `SLList`
  - could use `ARList`
  - could also use `ARSet`
  - any data structure which you can search and iterate
- This idea is called **separate chaining**



Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

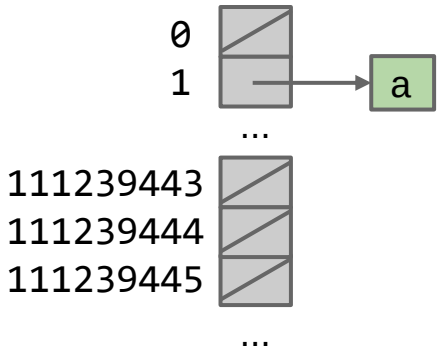


initially, all buckets are empty

Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

```
add("a")
```

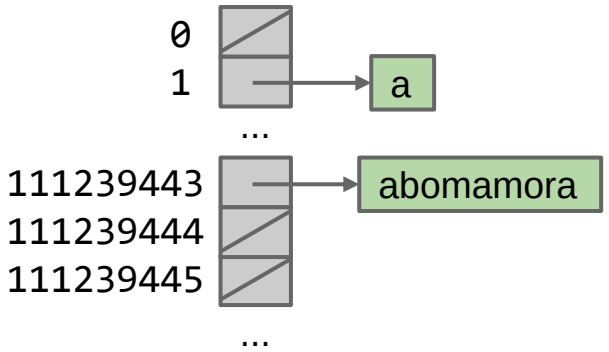


bucket 1 now has a list of length 1

Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

```
add("a")
add("abomamora")
```

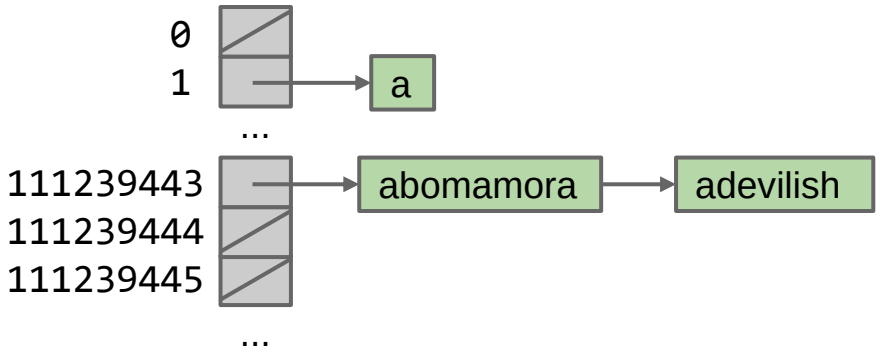


using englishToInt, abomamora hashes to 111239443, create new list

Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

```
add("a")
add("abomamora")
add("adevilish")
```

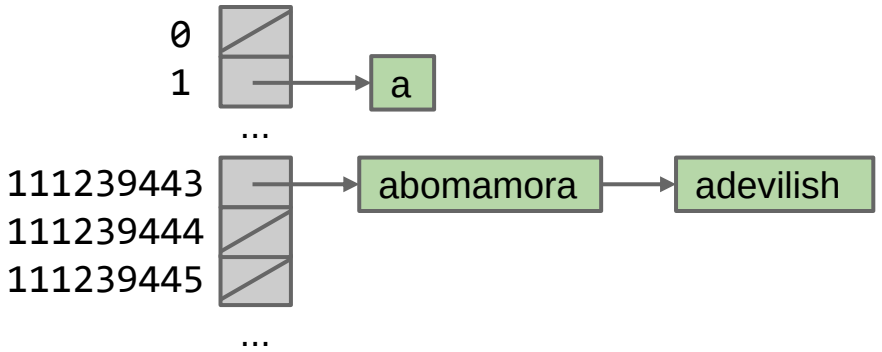


adevilish also hashes to 111239443, added in list

Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

```
add("a")
add("abomamora")
add("adevilish")
add("abomamora")
```



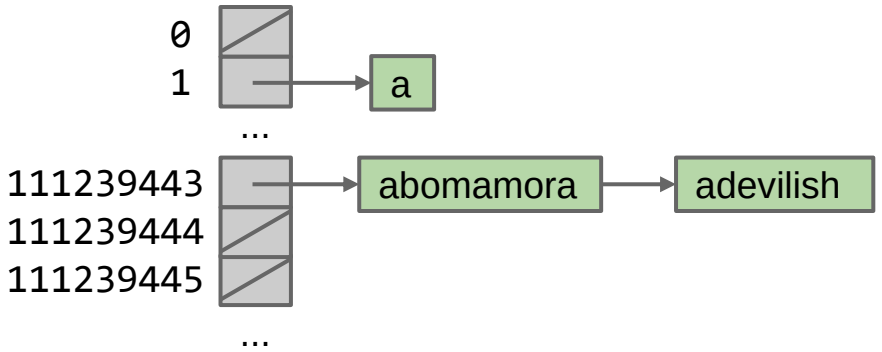
already present, not added

Separate Chaining

- In separate chaining, when an item x gets added at index h:
  - If bucket h is empty, we create a new list containing x and store it at index h
  - If bucket h is already a list, we add x to this list *if* it is *not* already present

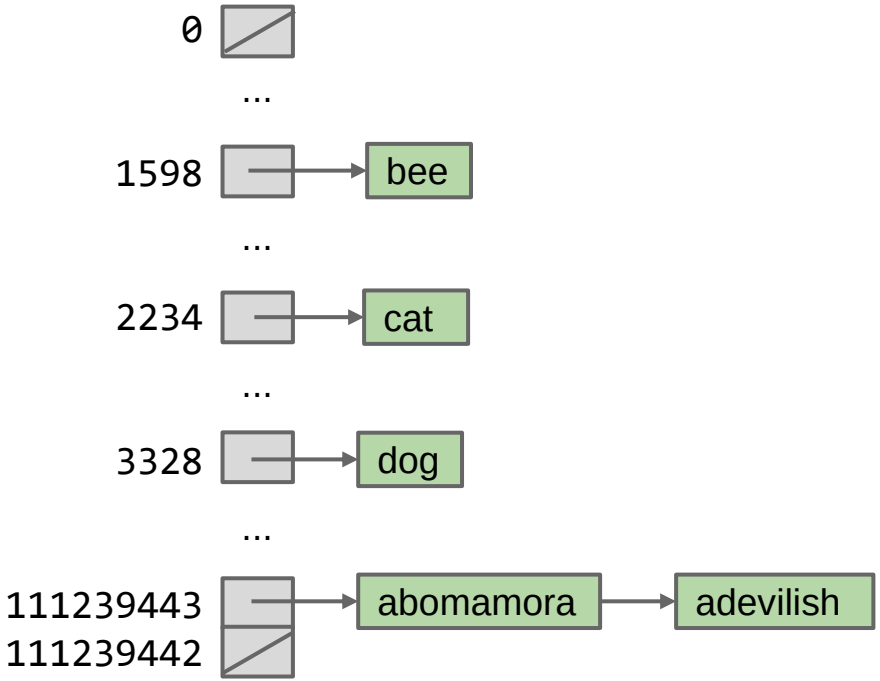
```
add("a")
add("abomamora")
add("adevilish")
add("abomamora")
contains("adevilish")
```

→ return true



Separate Chaining Performance

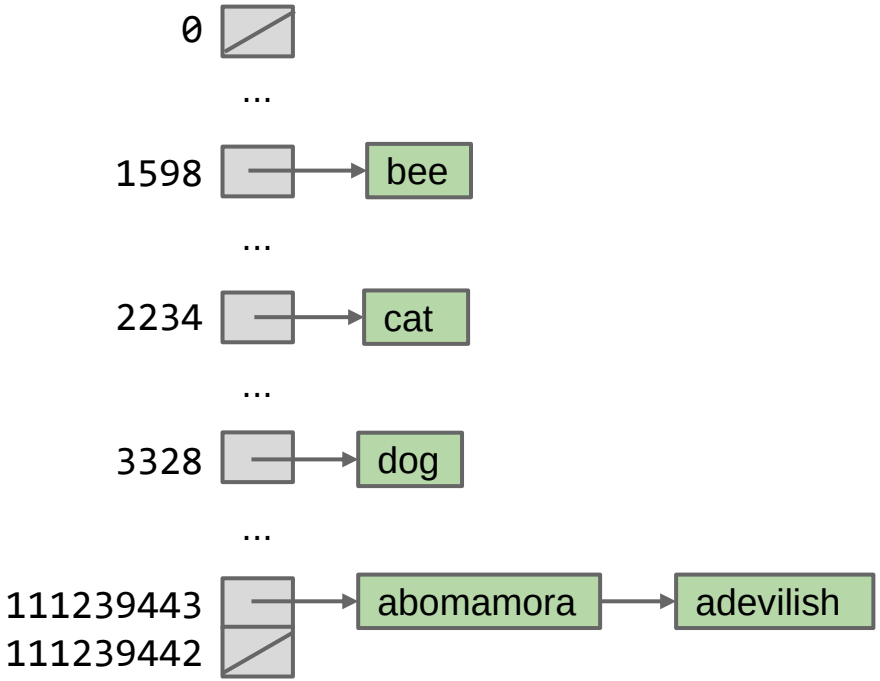
- The running time of add and contain is O(Q), where Q is the length of the longest list
  - no longer O(1), since to add, need to check whether the item is already in the list

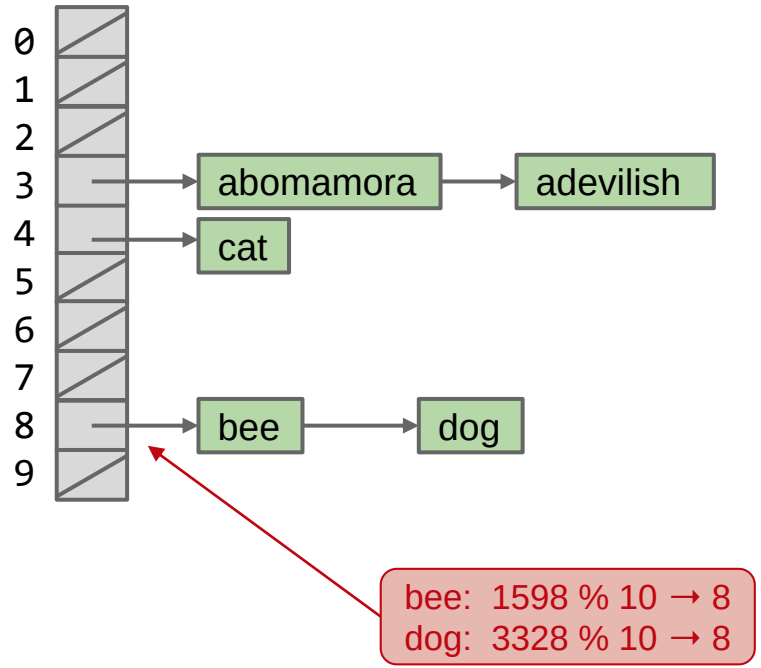


let us improve our design.
notice that we don't need billions of buckets.
what can we do to save space?

Saving Memory

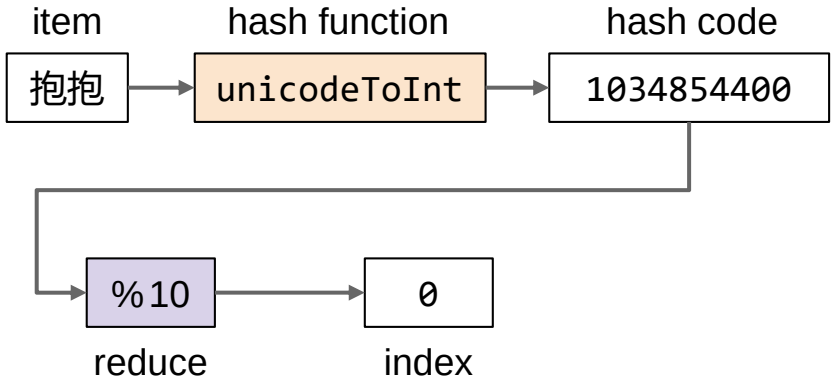
- We could reduce the number of buckets, and use modulus of hash code to compute index
  - for example, use 10 buckets, and compute bucket number = hash code % 10

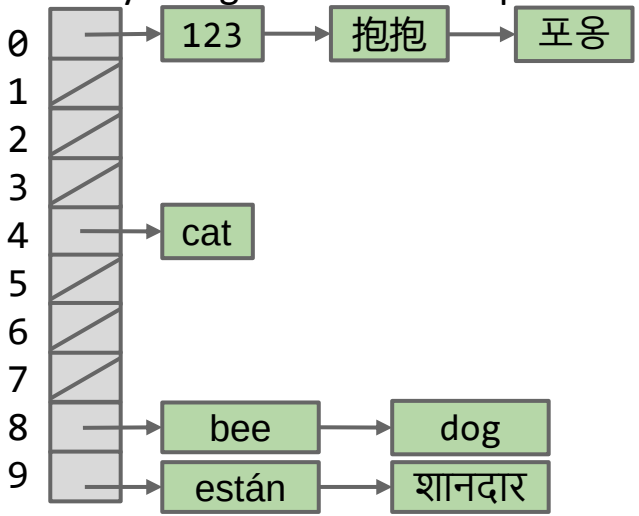




Hash Table

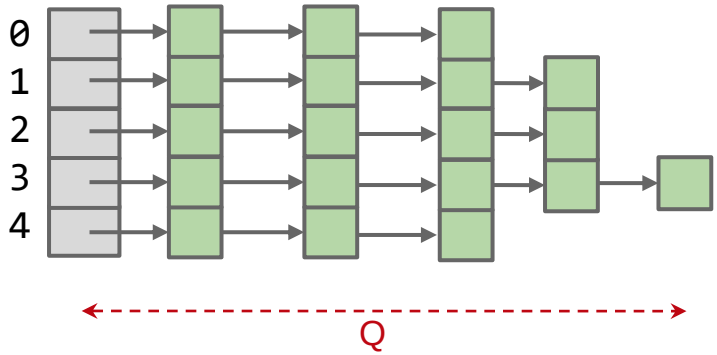
- We have just created the so-called **hash table**
  - The item is converted by **a hash function** into an integer representation called **a hash code**
  - The hash code is then reduced to **a valid index**, usually using the modulus operator





Hash Table Performance

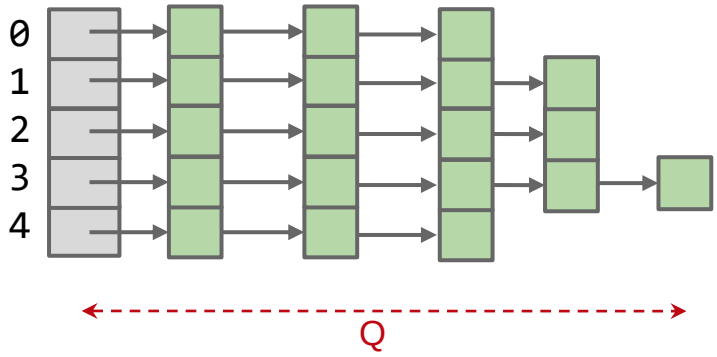
- Hash table uses way less memory/space than `INTSet` or `ENGSet`,
  but now the running time is no longer constant O(1), it becomes O(Q)
  - where Q is the length of the longest list



for this hash table that has 5 buckets,
and stores N items,
what is the order of growth of Q
in terms of N ?

Hash Table Performance

- Hash table uses way less memory/space than `INTSet` or `ENGSet`,
  but now the running time is no longer constant O(1), it becomes O(Q)
  - where Q is the length of the longest list

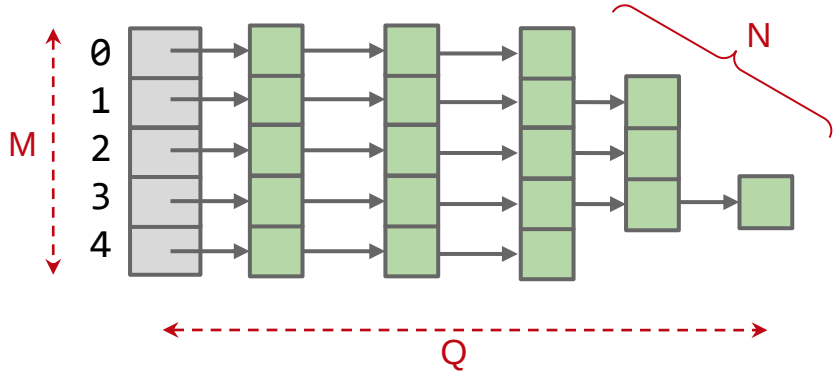


for this hash table that has 5 buckets,
and stores N items,
what is the order of growth of Q
in terms of N ?

In the best case, the items are evenly distributed, the length of the longest list will be N/5.
In the worst case, all items in a single list, it will be N.
So, Q(N) is O(N), which is not good

Improving Hash Table

- Suppose we have:
  - A fixed number of buckets M
  - An increasing number of items N



- Even if items are spread out evenly, lists are of length Q = N/M
  - For M = 5, that means Q = O(N)
  - Results in linear time operations, which is bad

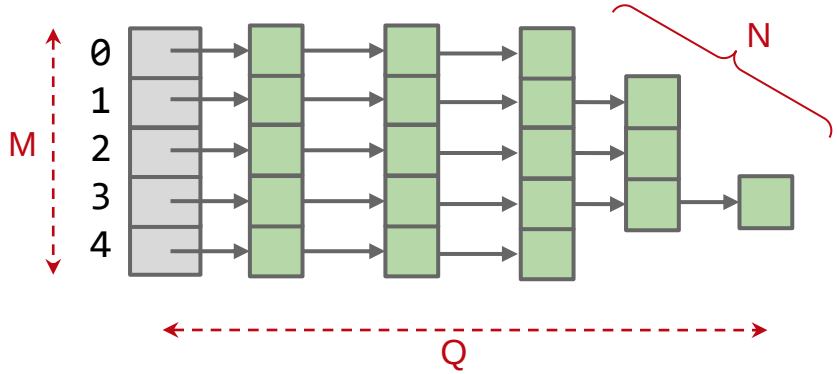
- How can we improve our design to guarantee that N/M is O(1)?

Improving Hash Table

- Suppose we have:
  - ~~A fixed~~ number of buckets M

    An increasing
  - An increasing number of items N

improve the ratio by growing M at the same rate of N



- Even if items are spread out evenly, lists are of length Q = N/M
  - For M = 5, that means Q = O(N)
  - Results in linear time operations, which is bad

- How can we improve our design to guarantee that N/M is O(1)?

Improving Hash Table 2

- As long as M = O(N), then O(N/M) = O(1)

- Idea: When N/M is ≥ 1.5, then double M
  - when it gets too full, double the buckets!
  - we call this process of increasing M **resizing**
  - N/M is often called the **load factor**
  - it represents how full the hash table is

- This rule ensures that the average list is never more than 1.5 items long!

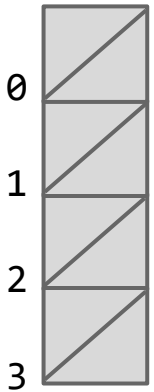
use this idea in your Lab 15: HAMap !

Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M

suppose we start with 4 empty buckets

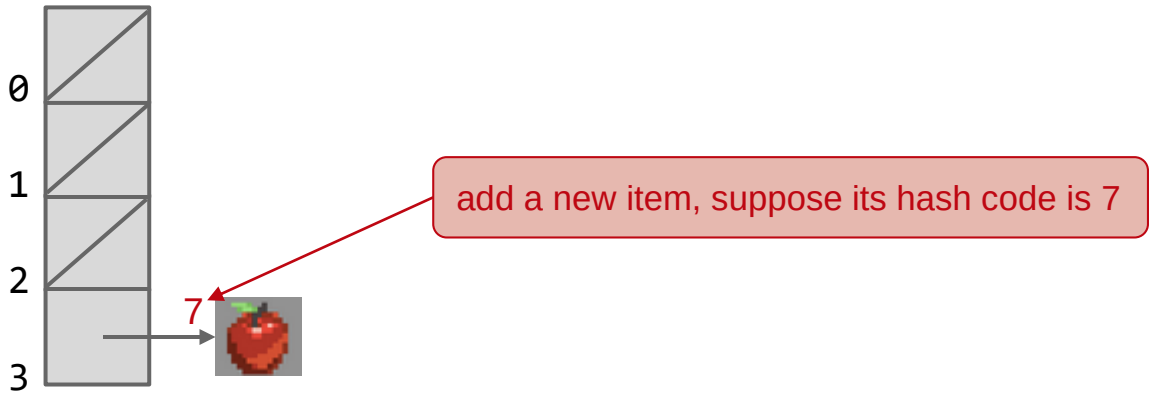
N = 0 M = 4 N/M = 0



Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M

N = 1 M = 4 N/M = 0.25



Improving Hash Table Example

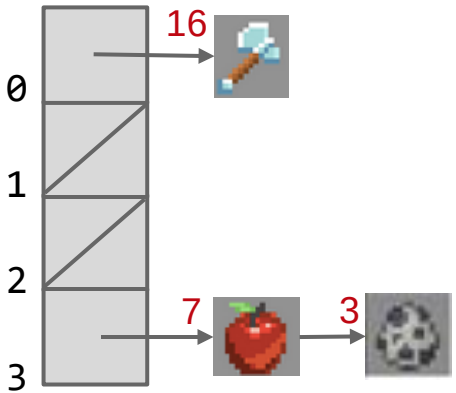
- Idea: When N/M is ≥ 1.5, then double M

N = 2 M = 4 N/M = 0.5

Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M

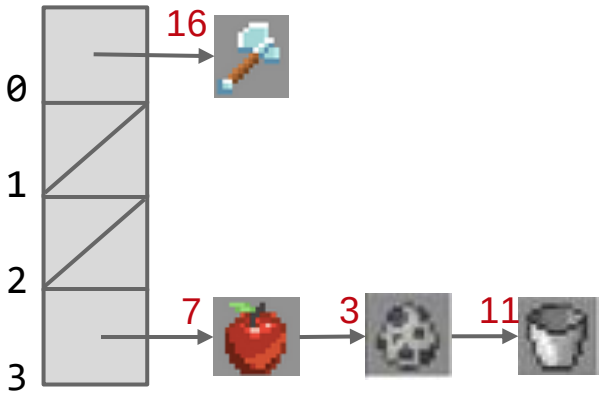
N = 3 M = 4 N/M = 0.75



Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M

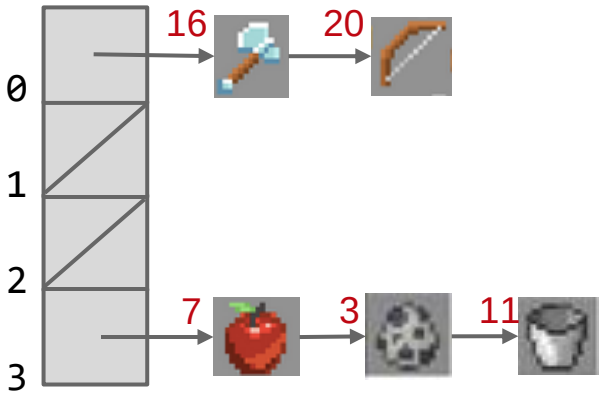
N = 4 M = 4 N/M = 1



Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M

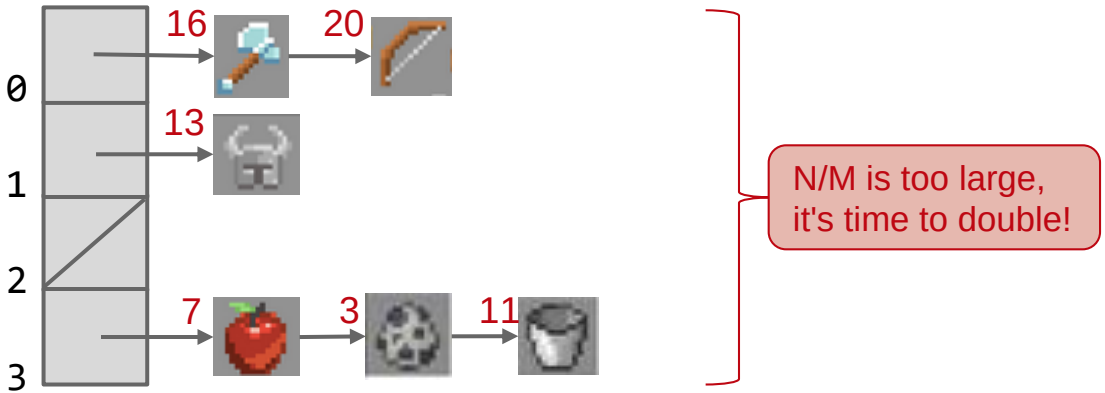
N = 5 M = 4 N/M = 1.25



Improving Hash Table Example

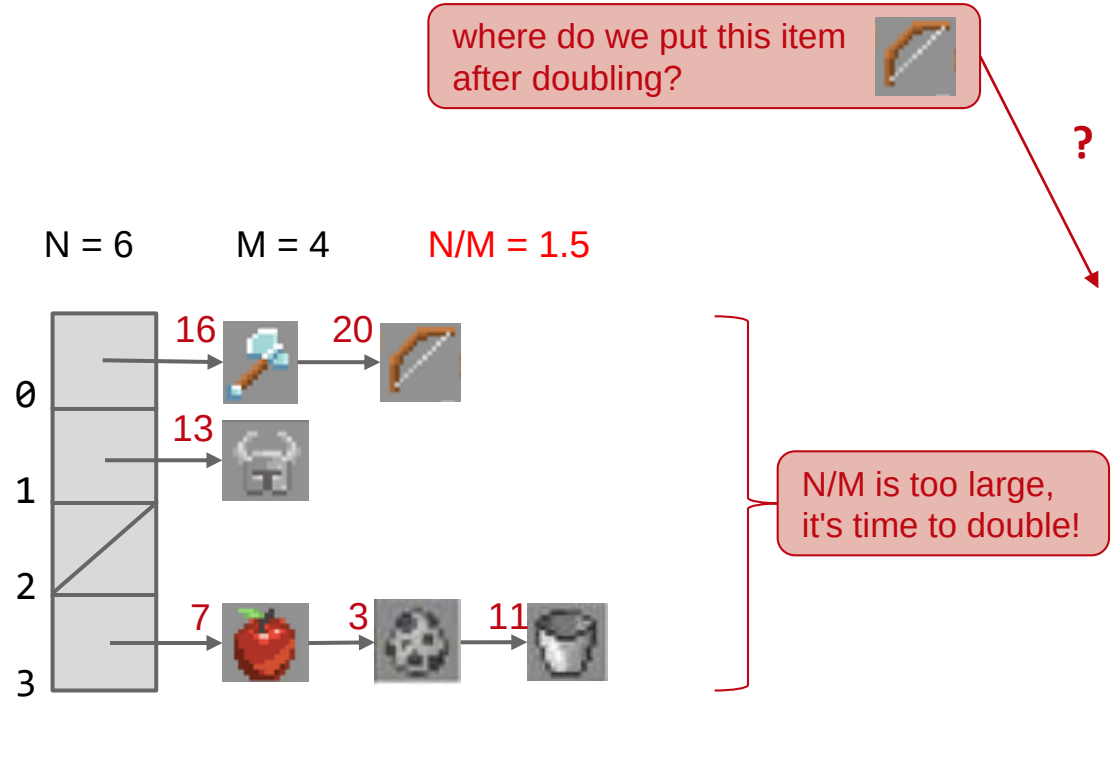
- Idea: When N/M is ≥ 1.5, then double M

N = 6 M = 4 N/M = 1.5



In-Class Quiz 2

- Idea: When N/M is ≥ 1.5, then double M

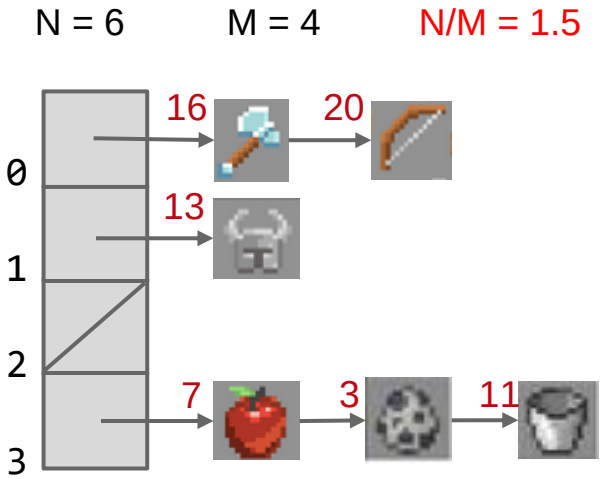


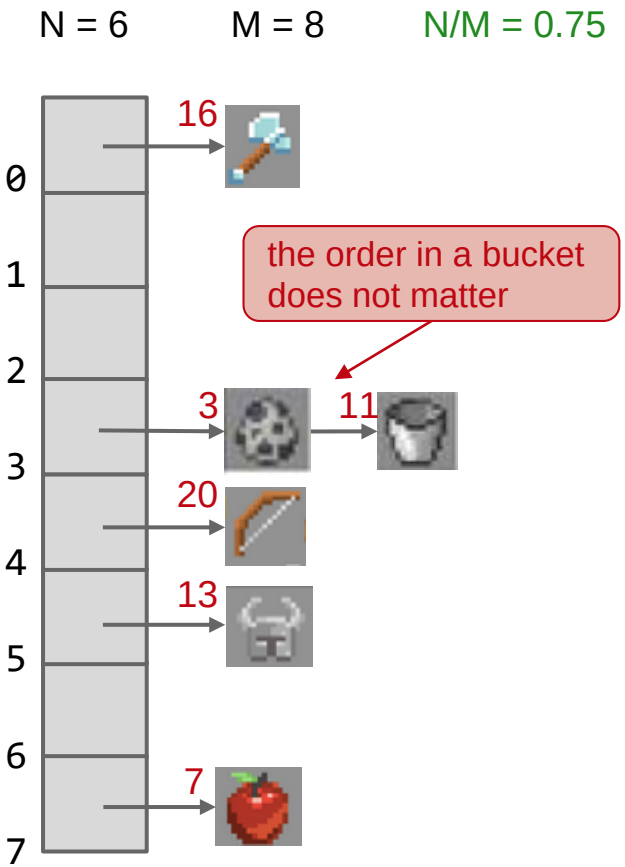
Select one:

- 0
- 4
- 8
- 20

Improving Hash Table Example

- Idea: When N/M is ≥ 1.5, then double M





Resizing Hash Table Running Time

- As long as M = O(N), then O(N/M) = O(1)
- *Assuming items are evenly distributed*, lists will be approximately N/M items long, resulting in O(N/M) runtimes
  - Our doubling strategy ensures that N/M = O(1)
  - Thus, worst case runtime for all operations is O(N/M) = O(1)
  - unless that operation causes a resize
- What is the runtime of resizing?

Resizing Hash Table Running Time

- As long as M = O(N), then O(N/M) = O(1)
- *Assuming items are evenly distributed*, lists will be approximately N/M items long, resulting in O(N/M) runtimes
  - Our doubling strategy ensures that N/M = O(1)
  - Thus, worst case runtime for all operations is O(N/M) = O(1)
  - unless that operation causes a resize
- Resizing takes O(N) time, because we have to redistribute all items!
  - Most add operations will be O(1), but some will be O(N) time to resize
  - Similar to our ARList, as long as we resize by a multiplicative factor, the average runtime will still be O(1)

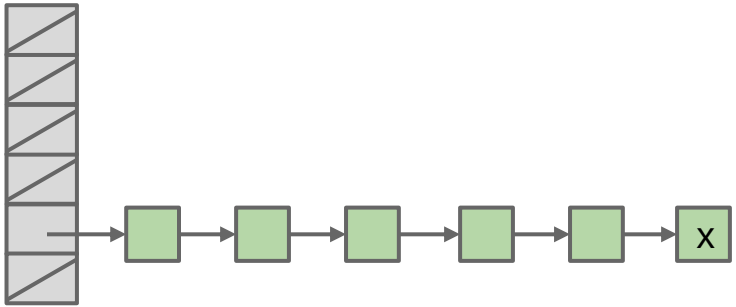
Even Distribution

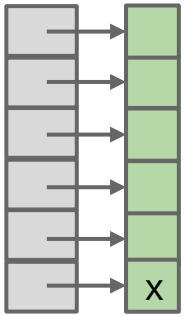
- Thus, Resizing Hash Table operations are on average constant time if:
  1. we double M to ensure constant average bucket length
  2. items are evenly distributed

Even Distribution

- Thus, Resizing Hash Table operations are on average constant time if:
  1. we double M to ensure constant average bucket length
  2. items are evenly distributed
- Even distribution of item is critical for good hash table performance
  - both tables below have load factor of N/M = 1
  - but left table is much worse, contains is O(N) for item x

we need to discuss how to ensure even distribution !





Object's Hash Code

- Recall that all classes are subclass of `Object`

Modifier and Type	Method	Description
protected `Object`	**`clone`**()	Creates and returns a copy of this object.
boolean	**`equals`**(`Object` obj)	Indicates whether some other object is "equal to" this one.
protected void	**`finalize`**()	**Deprecated.** The finalization mechanism is inherently problematic.
`Class`<?>	**`getClass`**()	Returns the runtime class of this `Object`.
int	**`hashCode`**()	Returns a hash code value for the object.
void	**`notify`**()	Wakes up a single thread that is waiting on this object's monitor.
void	**`notifyAll`**()	Wakes up all threads that are waiting on this object's monitor.
`String`	**`toString`**()	Returns a string representation of the object.
void	**`wait`**()	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
void	**`wait`**(long timeout)	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
void	**`wait`**(long timeout, int nanos)	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

default implementation of `hashCode` returns the memory address of the object

String's Hash Code

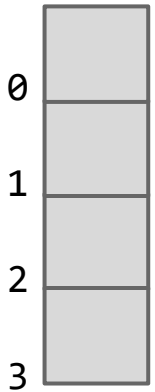
- Java's `String` overrides `hashCode`
  - for example:

```
System.out.println("a".hashCode());
System.out.println("bee".hashCode());
System.out.println("포옹".hashCode());
System.out.println("kamala lifefully".hashCode());
System.out.println("đậu hũ".hashCode());
```

```
→ 97
→ 97410
→ 1732557
→ 1732557
→ -2108180664
```

Negative Hash Code in Java

- Suppose that the hash code of an item is -1
- Unfortunately, -1 % 4 → -1
  - will result in index errors!
- Solution: use `Math.floorMod` instead

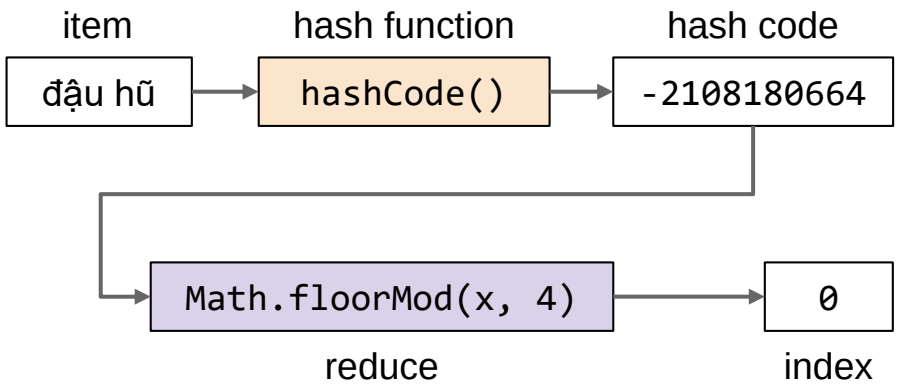


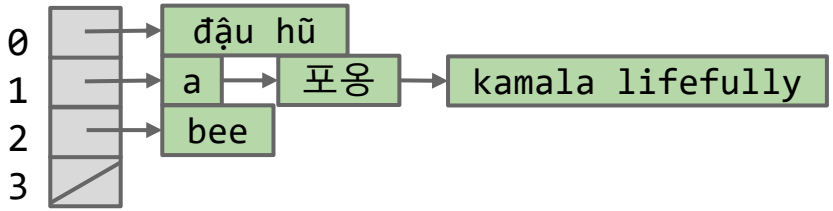

```
public class ModTest {
  public static void main(String[] args) {
    System.out.println(-1 % 4);
    System.out.println(Math.floorMod(-1, 4));
  }
}
```

→ -1
→ 3

Our Hash Table Set Implementation HASet

- Here is our set implementation using hash tables in Java, called HASet:
  - item is converted by the **hashCode method** into an integer representation called a **hash code**
  - the hash code is then reduced to a **valid index**, using the `Math.floorMod` method





HASet: Member Variables and Constructor

- Here is our hash-based implementation of set

```java
/*
 * Hash-based Set
 */
public class HASet<T> {

    private static final int DEFAULT_CAPACITY = 16;

    private ArrayList<ArrayList<T>> buckets;
    private int size;

    private int reduce(T item) {
        return Math.floorMod(item.hashCode(), DEFAULT_CAPACITY);
    }

    public HASet() {
        buckets = new ArrayList<>();
        for (int i = 0; i < DEFAULT_CAPACITY; i++) {
            buckets.add(new ArrayList<>());
        }
        size = 0;
    }
```

as a rep of a (fixed) hash table, we use ArrayList of ArrayList

hash function and reduce

initializing the buckets

HASet: Contains

- Recall that in a set, an item is either in a set or not

```java
/**
 * Checks whether an item is inside the set.
 * @param item to be checked
 * @return true iff the set contains the item
 */
public boolean contains(T item) {
    if (item == null) {
        return false;
    }
    int index = reduce(item);
    return buckets.get(index).contains(item);
}
```

compute the index,
go to the right bucket,
and call ArrayList.contains

HASet: Add

- Recall that in a set, we must have no duplicates

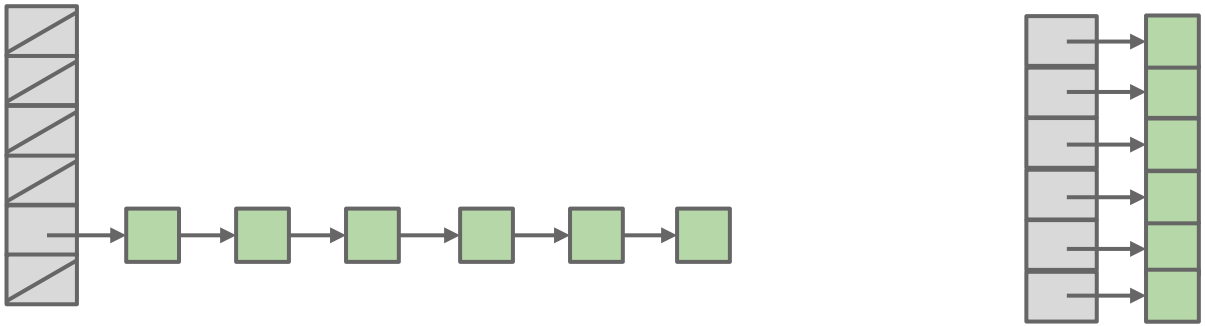
```java
/**
 * Adds an item into the set if it is not already inside.
 * @param item to be added inside the set.
 * @throws IllegalArgumentException if item is null.
 */
public void add(T item) {
    if (item == null) {
        throw new IllegalArgumentException();
    }
    if (contains(item)) {
        return;
    }
    int index = reduce(item);
    buckets.get(index).add(item);
    size++;
}
```

compute the index,
go to the right bucket,
and add the item,
if it is not already in there

Bad hashCode

- The following are examples of bad hash functions
  - return 0
  - just returning the first character of a word, e.g. "cat" → 3
  - adding chars together, e.g. "ab" collides with "ba"

- Recall that we want hash tables that look like the table on the right:



String hashCode Method

- The following is the Java 8 `hashCode` for `String`:

```
@Override
public int hashCode() {
    int h = cachedHashValue;
    if (h == 0 && this.length() > 0) {
        for (int i = 0; i < this.length(); i++) {
            h = 31 * h + this.charAt(i);
        }
        cachedHashValue = h;
    }
    return h;
}
```

it cached the value, only compute if it is not computed before

it uses a small base

Choosing a Base

- Java's `hashCode()` function for `Strings`:
  - $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \ldots + s_{n-1}$
- Our `asciiToInt` function for `Strings`:
  - $h(s) = s_0 \times 126^{n-1} + s_1 \times 126^{n-2} + \ldots + s_{n-1}$

- Which one is better?
  - recall that overflow is problem for base 126
  - and in turns, that causes collision

Base 126

- Major collision problem:
  - `"Laughter is the best thing on the earth.".hashCode()` yields 634199182
  - `"Muesli is the best thing on the earth.".hashCode()` yields 634199182
  - `"Being heard is the best thing on the earth.".hashCode()` yields 634199182
  - `"Memes is the best thing on the earth.".hashCode()` yields 634199182

- Any string that ends in the same last 32 characters has the same hash code !
  - Why? Because of overflow
  - The causes $126^{32} = 126^{33} = 126^{34} = \ldots = 0$
  - Thus, upper characters are all multiplied by zero

Typical Base

- A typical hash code base is a **small prime**
- Why prime?
  - Never even: avoid the overflow issue on previous slide
  - Lower chance of resulting hash code having a bad relationship with the number of buckets, e.g. the hash code is a multiple of the number of buckets
- Why small?
  - Lower cost to compute

Collection hashCode Method

- Suppose we have a collection of items, and each item has its own hashCode
  - somewhat similar to the one in `String`:

```
@Override
public int hashCode() {
    int hashCode = 1;
    for (Object o : this) {
        hashCode = hashCode * 31;
        hashCode = hashCode + o.hashCode();
    }
    return hashCode;
}
```

iterate over all items

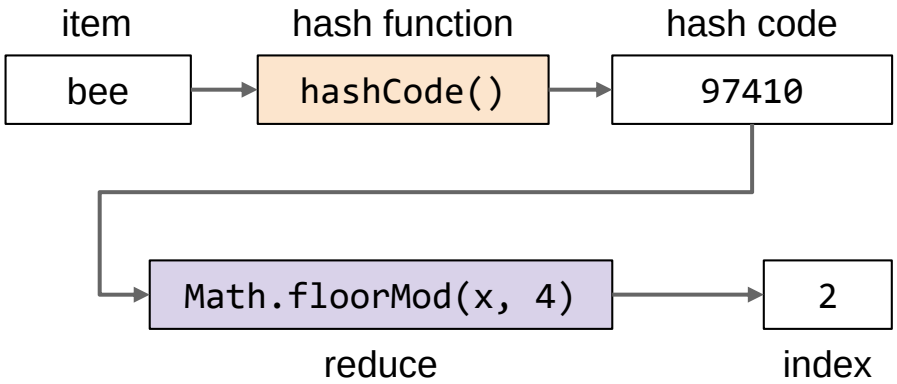
"raise" to the "next power" with small prime

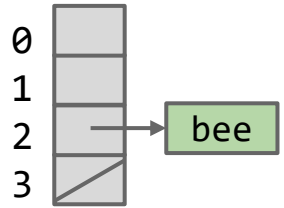
add the hashCode of an item

we may only look at the first few items, since the Collection maybe very large

Summary for Hash Table

- Hash table (`HASet`):
  - Item is converted into a hash code using good `hashCode` method
  - The hash code is then reduced to a valid index
  - Item is then stored in a bucket corresponding to that index
- Resizing hash table (for your `HAMap`):
  - Double when load factor N/M exceeds some constant, say 1.5 or 0.75
  - If items are spread out nicely, you get O(1) average runtime





Higher Order Functions

- Next, we will talk about `Comparator` which uses higher order function
- **Higher order function** is a function that takes another function as *input*
  - for example, the function `applyTwice` below takes another function `func`:

    function `applyTwice(func, x) : func(func(x))`
  - so, given function `multByTen(y) : 10 * y`

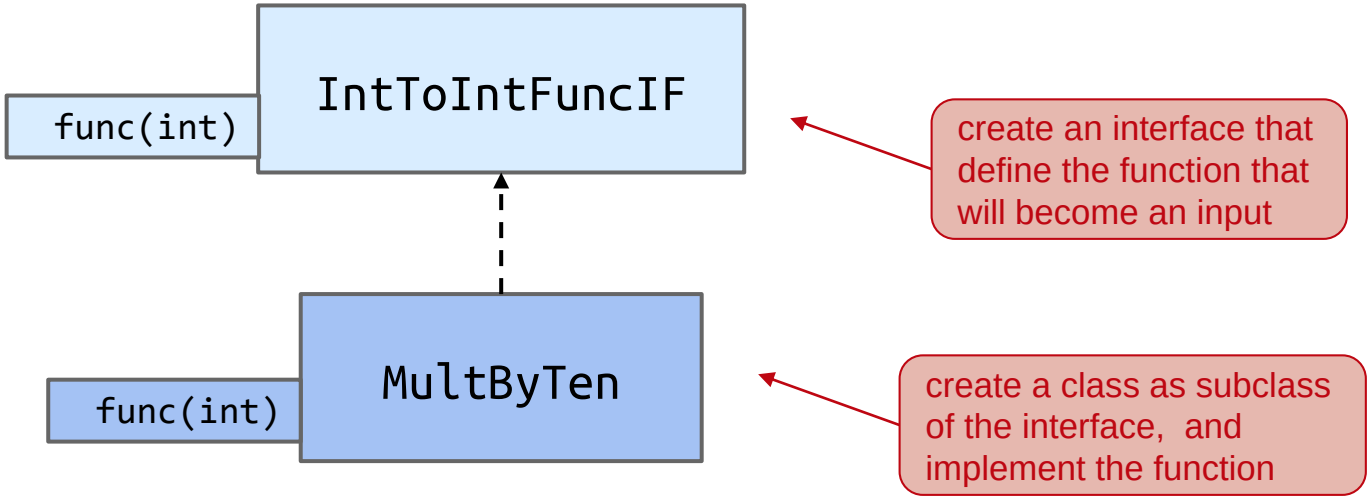
    we have that, `applyTwice(multByTen, 5)` → `500`

multByTen(multByTen(5))
= multByTen(10 * 5)
= 10 * 50
= 500

how do we do this in Java?

Higher Order Functions in Java 7

- In Java 7 and earlier, variables cannot contain pointers to functions
- So, we use an interface instead:



- We also learn this design pattern because it is going to be used in our next topic, the `Comparator`

Higher Order Functions by Interface (1)

1. Create an interface that declare the function which will be an input to another function

```
public interface IntToIntFuncIF {
    int func(int x);
}
```

represent an `int` to `int` function `func`

2. Create a class that is a subclass of that interface, and implement the function

```
public class MultByTen implements IntToIntFuncIF {

    /** @return ten times the int argument */
    public int func(int x) {
        return 10 * x;
    }

}
```

implement it as 10 * x function

Higher Order Functions by Interface (2)

3. Implement the function `applyTwice` that takes the interface and call the function declared in that interface

```
public class HigherOrderFunc {

    public static int applyTwice(IntToIntFuncIF f, int x) {
        return f.func(f.func(x));
    }

    public static void main(String[] args) {
        IntToIntFuncIF multByTen = new MultByTen();
        System.out.println(applyTwice(multByTen,  x: 5));
    }

}
```

→ 500

so instead of passing a function,
we pass an object that implements the function, and call the function (method)

3. Call the function `applyTwice` by passing an object of a specific class that implements the function as guaranteed by the interface

Higher Order Functions in Java 8

- Actually, you can do the following in the newer version:

```java
import java.util.function.Function;

public class Java8HigherOrderFunc {
    public static int multByTen(int x) {
        return 10 * x;
    }

    public static int applyTwice(Function<Integer, Integer> f, int x) {
        return f.apply(f.apply(x));
    }

    public static void main(String[] args) {
        int result = applyTwice(Java8HigherOrderFunc::multByTen, x: 5);
        System.out.println(result);
    }

}
```

must use `apply`

but the one with interface is used in `Comparator` next ...

Natural Order

- **Natural Order** is used to refer to the ordering implied by a `Comparable`'s `compareTo` method, for example by ***size*** in our `Dog` objects :

"XiaXue", size: 20

"FooDog", size: 200

"HellHound", size: 4444

Natural Order 2

- We don't always want to compare objects in the same way every time
  - for example, by ***name***:

"FooDog", size: 200

"HellHound", size: 4444

"XiaXue", size: 20

Additional Orders

- To do that in Java, we use the Higher Order Function technique: we create `SizeComparator` or `NameComparator` classes that implement `Comparator` interface
  - and fill in the method that is required called `compare` :

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

differentiate from `compareTo` of `Comparable` interface

same rule with `compareTo` (positive if `o1` is bigger, …)

- The methods comparing `T objects` must also take `Comparator<T>` arguments
  - for example, a comparison-based sorting method

NameComparator for Dog

- As an example, here is how implement NameComparator for Dog objects:

```
private static class NameComparator implements Comparator<Dog> {
    public int compare(Dog d1, Dog d2) {
        return d1.name.compareTo(d2.name);
    }
}

public static Comparator<Dog> getNameComparator() {
    return new NameComparator();
}
```

use private static class that implements `Comparator <Dog>`, so we need to `import java.util.Comparator`

implement `compare` by calling `compareTo` of `String`, since we compare names alphabetically

use a getter to return an object of `Comparator<Dog>`

Calling NameComparator for Dog Client

- And here is how we use the `NameComparator` for `Dog` objects:

```
import java.util.Comparator;

public class DogClient {
    public static void main(String[] args) {

        Dog d1 = new Dog( n: "XiaXue",  w: 20);
        Dog d2 = new Dog( n: "FooDog",  w: 200);
        Dog d3 = new Dog( n: "HellHound",  w: 4444);

        Comparator<Dog> nameComp = Dog.getNameComparator();
        if (nameComp.compare(d1, d2) < 0) {
            d1.bark();
        }
        else {
            d2.bark();
        }
    }
}
```

first, need to instantiate the `NameComparator` by calling the getter

call `compare`, in this case, do decide if `d1` comes before than `d2` in the alphabet

this we can compare two objects in many ways!

Part 2 : Concurrency

- In the second part of the lecture, we will
  - learn about message passing and shared memory models of concurrency
  - study concurrent processes and threads, time slicing, and the danger of race conditions

Concurrency (1)

- Concurrency means **multiple** computations are happening **at the same time**
- Concurrency is *everywhere* in modern programming, whether we like it or not:
  - Multiple computers in a network
  - Multiple applications running on one computer
  - Multiple processors in a computer
    - today, often multiple processor *cores* on a single chip

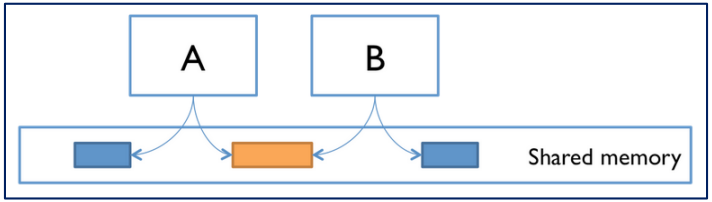
Concurrency (2)

- In fact, concurrency is essential in modern programming:
  - Web sites must handle multiple simultaneous users
  - Mobile apps need to do some of their processing on servers "*in the cloud*"
  - Graphical user interfaces almost always require background work that does not interrupt the user
    - for example, IntelliJ compiles your Java code *while* you're still editing it!
- Being able to program with concurrency will still be important in the future
  - Processor clock speeds are no longer increasing
  - Instead, we're getting more cores with each new generation of chips
  - So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces

Two Models for Concurrent Programming (1)

There are two common models for concurrent programming: shared memory and message passing

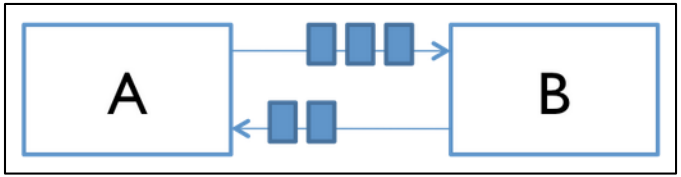
- **Shared memory**. In the shared memory model of concurrency, concurrent modules interact by *reading and writing shared objects in memory*



- Examples of the shared-memory model:
  - A and B might be two *processors* (or processor cores) in the same computer, sharing the same *physical memory*
  - A and B might be two *programs* running on the same computer, sharing a common *filesystem* with files they can read and write
  - A and B might be two *threads* in the same Java program (will explain what a thread is), sharing the same *Java objects*

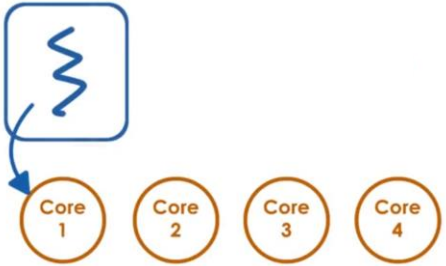
Two Models for Concurrent Programming (2)

- **Message passing**. In the message-passing model, concurrent modules interact by sending messages to each other through *a communication channel*
- Modules send off messages, and incoming messages to each module are *queued* up for handling



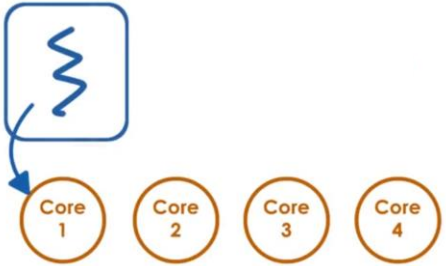
- Examples include:
  - A and B might be two *computers* in a network, communicating by *network connections*
  - A and B might be *a web browser* and *a web server* — A opens a *connection* to B and asks for a *web page*, and B sends the *web page data* back to A
  - A and B might be an instant messaging *client* and *server*
  - A and B might be two *programs* running on the same computer whose input and output have been connected by a *pipe*, like ls | grep typed into a command prompt

Processes (1)



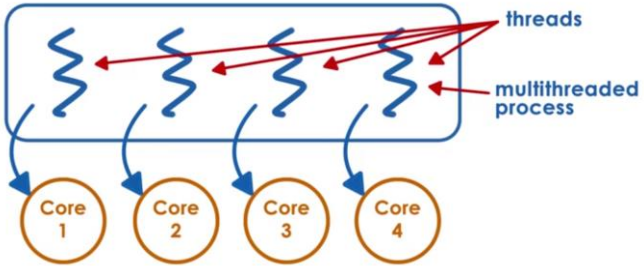
- The message-passing and shared-memory models are about *how* concurrent modules communicate
- The *concurrent modules* themselves come in two different kinds: processes and threads
- **Process**. A process is *an instance of a running program* that is isolated from other processes on the same machine
  - in particular, it has *its own private section* of the machine's memory
- The process abstraction is a *virtual computer*
  - it makes the program feel like it has the entire machine to itself — like a fresh computer has been created, with fresh memory, just to run that program

Processes (2)



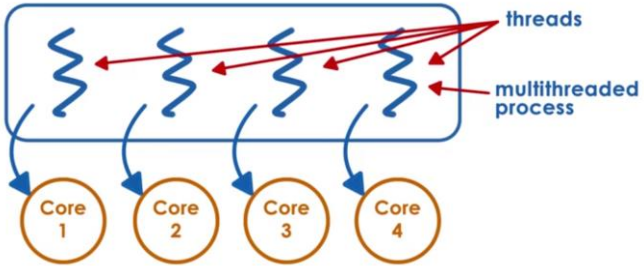
- Just like computers connected across a network, processes normally *share no memory* between them
  - A process *can't* access another process's memory or objects at all
    - sharing memory between processes is possible on most operating systems, but it needs special effort
  - By contrast, a new process is *automatically ready for message passing*, because it is created with standard input and output streams
    - which are the `System.out` and `System.in` streams you've used in Java

Threads (1)



- **Thread**. A thread is *a locus of control inside a running program*
  - think of it as a place in the program that is being run,
  - plus the stack of method calls that led to that place
    - so the thread can go back up the stack when it reaches return statements
- Just as a process represents a virtual computer, the thread abstraction represents *a virtual processor*
  - making a new thread simulates making a fresh processor inside the virtual computer represented by the process
  - this new virtual processor *runs the same program* and *shares the same memory as other threads* in the process

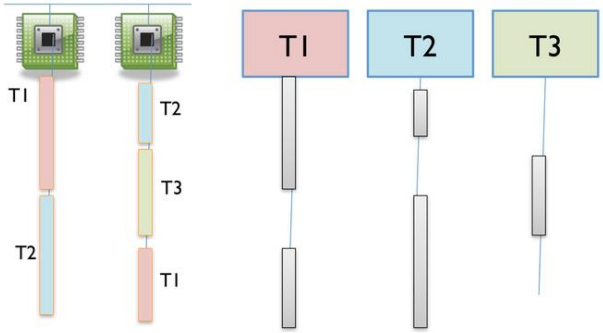
Threads (2)



- Threads are *automatically ready for shared memory*, because threads share all the memory in the process
  - it takes special effort to get "thread-local" memory that's private to a single thread
  - it's also necessary to set up message-passing explicitly, by creating and using queue data structures
    - will discuss about that in future lecture

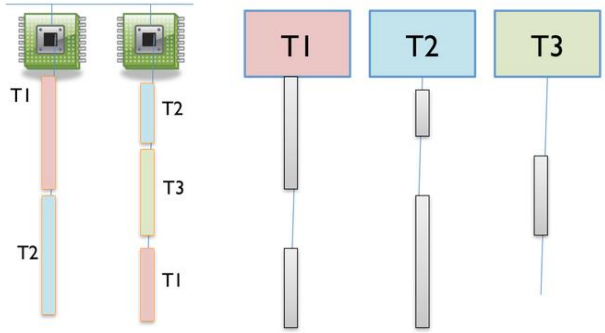
- Whenever you run a Java program, the program starts with one thread, which calls `main()` as its first step
  - this thread is referred to as the **main thread**

Time-slicing (1)



- How can you have many concurrent threads with only one or two processors in my computer?
- When there are more threads than processors, concurrency is *simulated* by **time slicing**, which means that the processor switches between threads
- The figure above shows how *three* threads T1, T2, and T3 might be time-sliced on a machine that has only *two* actual processors

Time-slicing (2)



- In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3
- Thread T2 simply pauses, until its next time slice on the same processor or another processor
- On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time

Starting a thread in Java (1)

- You can start new threads by making an instance of `Thread` and telling it to `start()`
  - You provide code for the new thread to run by creating a class implementing `Runnable`
  - The first thing the new thread will do is call the `run()` method in this class
  - For example:

```
// ... in the main method:
new Thread(new HelloRunnable()).start();

// elsewhere in the code
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello! It's me, a thread!");
    }
}
```

create an instance of `HelloRunnable` class, and pass that instance to the `Thread` constructor

Starting a thread in Java (2)

- But a very common idiom is starting a thread with an ***anonymous*** `Runnable`, which eliminates the need to name the `HelloRunnable` class at all:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello! It's me, a thread!");
    }
}).start();
```

Anonymous Class (1)

- An **anonymous class** is an implementation of an interface that has ***no class name*** of its own
- Usually when we implement an interface, we do so by declaring a class
  - For example, in the first part of lecture, the interface `Comparator` in the Java API:

```
/** A comparison function that imposes a total ordering on some objects.
 *  ... */
public interface Comparator<T> {
    /** Compares its two arguments for order.
    *  ...
    *  @return a negative integer, zero, or a positive integer if the first
    *         argument is less than, equal to, or greater than the second */
    public int compare(T o1, T o2);
}
```

Anonymous Class (2)

- We might declare:

```
/** Orders Strings by length (shorter first) and then lexicographically. */
public class StringLengthComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }
}
```

Anonymous Class (3)

- One purpose of `Comparator` is for sorting, for example `SortedSet` keeps its items in a total order
- Without a `Comparator`, the `SortedSet` implementation uses the `compareTo` method provided by the objects in the set:

```
SortedSet<String> strings = new TreeSet<>();
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "alice", "bob", "yolanda", "zach" }
```

lexicographically

With a `Comparator`:

```
// uses StringLengthComparator declared in previous slide
Comparator<String> compareByLength = new StringLengthComparator();
SortedSet<String> strings = new TreeSet<>(compareByLength);
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "bob", "zach", "alice", "yolanda" }
```

by length, then lexicographically

Anonymous Class (4)

- If we only intend to use this comparator in this one place,
  we already know how to eliminate the variable:

```
// uses StringLengthComparator declared in previous slide
SortedSet<String> strings = new TreeSet<>(new StringLengthComparator());
```

Anonymous Class (5)

- An anonymous class declares an *unnamed* class that implements an interface and immediately creates the *one and only* instance of that class

```
// no StringLengthComparator class!
SortedSet<String> strings = new TreeSet<>(new Comparator<String>() {
    @Override public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }} );
```

Anonymous Class (6)

- Advantages of anonymous class over named class:
  - if we're only using the comparator in this one piece of code, we've reduced its scope by using an anonymous class; with a named class, any other code could start using and depending on `StringLengthComparator`
  - a reader no longer has to search elsewhere for the details of the comparator; everything is right here
- Disadvantages:
  - If we need the same comparator more than once, we might be tempted to copy-and-paste; a named class is more DRY
  - If the implementation of the comparator is long, it interrupts the surrounding code, making it harder to understand; a named class is separated out as a modular piece
- So anonymous classes are good for ***short one-off*** implementations of a method

the Runnable we use to create new threads often meet these criteria perfectly

Using an anonymous Runnable to start a thread

- To start a new thread, we create an anonymous implementation of `Runnable` which represents the code we want the new thread to start running:

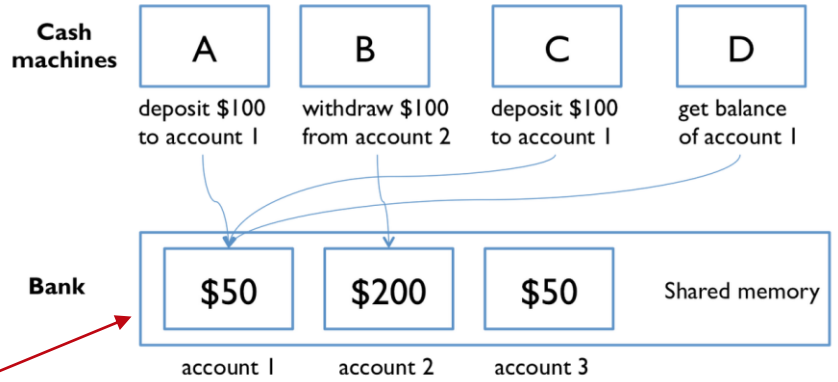
```
public static void main(String[] args) {

    new Thread(new Runnable() {
        public void run() {
            System.out.println("Hello! It's me, a thread!");
        }
    }).start();

}
```

Shared Memory System (1)

- Let's look at an example of a shared memory system
- The point of this example is to show that concurrent programming is *hard*, because it can have subtle bugs
- Imagine that a bank has cash machines that use *a shared memory model*, so all the cash machines can *read* and *write* **the same account objects in memory**



Shared Memory System (2)

- To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

- Customers use the cash machines to do transactions like this:

```
deposit();  // put a dollar in
withdraw(); // take it back out
```

Shared Memory System (3)

- Each transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account *unchanged*
- Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions:

```
// each ATM does a bunch of transactions that modify balance,
// but leave it unchanged afterward
public static void cashMachine() {
    new Thread(new Runnable() {
        public void run() {
            for (int i = 0; i < TRANSACTIONS_PER_MACHINE; i++) {
                deposit();  // put a dollar in
                withdraw(); // take it back out
            }
        }
    }).start();
}
```

Shared Memory System (4)

- So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed,
  - we should *expect* the account balance to **still be 0**

- But if we run this code, we discover frequently that the balance at the end of the day is not 0
  - if more than one thread *created by* `cashMachine()` is running at the same time – say, on separate processors in the same computer – then `balance` **may not** be zero at the end of the day
  - *why not*?

Interleaving (1)

- Here's one thing that can happen
- Suppose two cash machine threads, A and B, are *both working on a deposit at the same time*
- Here's how the `deposit()` step typically breaks down into *low-level processor instructions*:

get balance (balance=0)
add 1
write back the result (balance=1)

Interleaving (2)

- When A and B are running concurrently, these low-level instructions can **interleave** with each other, meaning that the actual sequence of execution can intersperse A's operations and B's operations arbitrarily
- Here is one possible interleaving:

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

- This interleaving is fine — we end up with balance 2, so both A and B successfully put in a dollar

In-Class Quiz 3

- But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result	
	B write back the result

- What is the value of `balance` now ?
  - 0
  - 1
  - 2
  - 3

Interleaving (3)

- But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result	
	B write back the result

- The `balance` is now — A's dollar was **lost**!
- A and B *both read the balance at the same time*, computed separate final balances, and then **raced** to store back the new balance — which failed to take the other's deposit into account

Race Condition

- That is an example of **a race condition**
- A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the *relative timing* of events in concurrent computations A and B
  - when this happens, we say **"A is in a race with B"**

- Some interleavings of events may be OK, in the sense that they are consistent with what a single, non-concurrent process would produce, but other interleavings produce wrong answers — violating postconditions or invariants

Tweaking the Code Won't Help (1)

- All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
// version 2
private static void deposit() {
    balance += 1;
}
private static void withdraw() {
    balance -= 1;
}
```

Tweaking the Code Won't Help (2)

```
// version 3
private static void deposit() {
     ++balance;
}
private static void withdraw() {
     --balance;
}
```

Tweaking the Code Won't Help (3)

- You can't tell just from looking at Java code how the processor is going to execute it
- You can’t tell what the **atomic operations** (the indivisible steps of the computation) will be
  - it isn't atomic just because it's one line of Java
  - it doesn't touch `balance` only once just because the balance identifier occurs only once in the line
- The Java compiler, and in fact the processor itself, makes *no commitments* about what low-level operations it will generate from your code
  - in fact, a typical modern Java compiler produces exactly the same code for **all three** of those versions!
- The key lesson is that you *can't tell* by looking at an expression whether it will be safe from race conditions

Reordering (1)

- It's even worse than that, in fact
- The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors
- But in fact, when you're using multiple variables and multiple processors, you ***can't even count on*** changes to those variables *appearing in the same order*

Reordering (2)

- Here's an example of a broken code
- Note that it uses a loop that continuously checks for a concurrent condition; this is called **busy waiting** and it is ***not*** a good pattern

```
private boolean ready = false;
private int answer = 0;
// computeAnswer runs in one thread
private void computeAnswer() {
    // ... calculate for a long time ...
    answer = 42;
    ready = true;
}
// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

Reordering (3)

- We have two methods that are being run in different threads
  - `computeAnswer` does a long calculation, finally coming up with the answer 42, which it puts in the `answer` variable
  - then it sets the `ready` variable to `true`, in order to signal to the method running in the other thread, `useAnswer`, that the answer is ready for it to use
- Looking at the code, `answer` is set before `ready` is set,
  - so once `useAnswer` sees `ready` as `true`,
  - then it seems reasonable that it can assume that the `answer` will be 42, *right*?
    - *not* so...

Reordering (4)

- The problem is that modern compilers and processors do a lot of things to make the code fast
  - one of those things is *making temporary copies of variables* like `answer` and `ready` in faster storage (processor registers or processor caches), and
  - *working with them temporarily* before eventually storing them back to their official location in memory
  - the storeback **may occur in a different order** than the variables were manipulated in your code

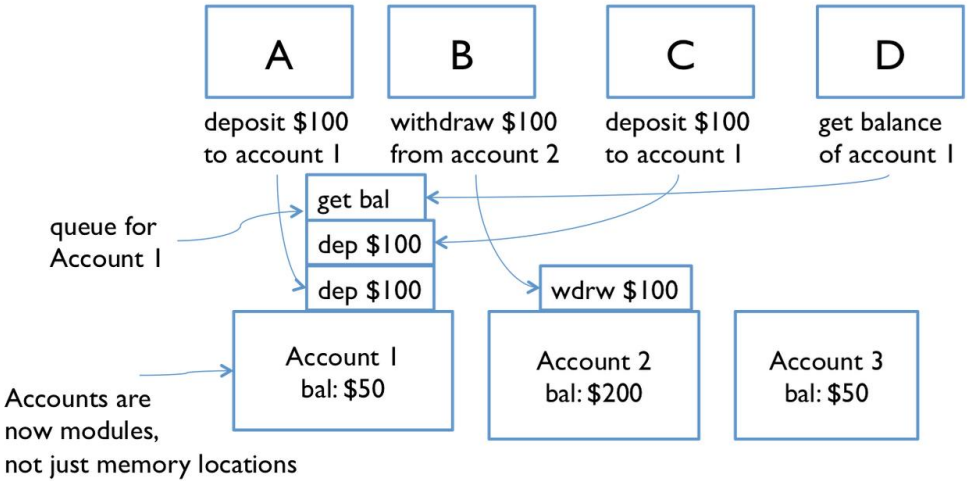
Reordering (5)

- Here's what might be going on under the covers (but expressed in Java syntax to make it clear)
- The processor is effectively creating two temporary variables, `tmpr` and `tmpa`, t o manipulate the fields `ready` and `answer`:

```
private void computeAnswer() {
    boolean tmpr = ready;
    int tmpa = answer;
    tmpa = 42;
    tmpr = true;
    ready = tmpr;
                        // <-- what happens if useAnswer() interleaves here?
                        // ready is set, but answer isn't!
    answer = tmpa;
}
```

Message Passing Example (1)

- Now let's look at the *message-passing approach* to our bank account example
- Now not only are the cash machine modules, but the accounts are modules, too
- Modules interact by sending messages to each other
- Incoming requests are placed in a queue to be handled one at a time

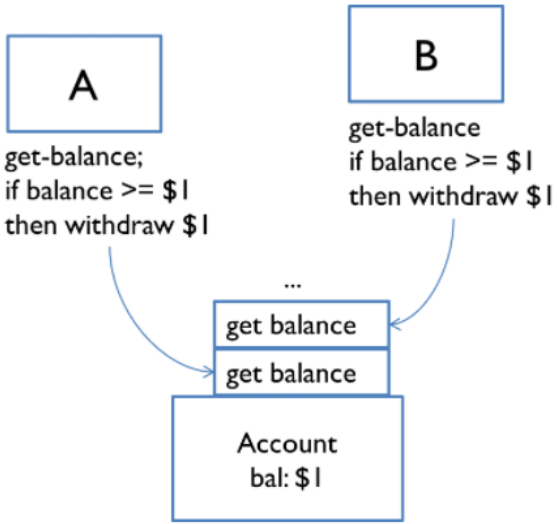


- The sender *doesn't stop working* while waiting for an answer to its request
  - it handles more requests from its own queue
  - the reply to its request eventually comes back as another message

Message Passing Example (2)

- Unfortunately, message passing *doesn't* eliminate the possibility of *race conditions*
- Suppose each account supports `get-balance` and `withdraw` operations, with corresponding messages
  - Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account
  - They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties:

```
get-balance
if balance >= 1 then withdraw 1
```



Message Passing Example (3)

- The problem is again interleaving, but this time *interleaving of the **messages*** sent to the bank account, rather than the *instructions* executed by A and B
- If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account
- One lesson here is that you need to carefully choose the operations of a message-passing model
  - `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`

Concurrency is Hard to Test and Debug (1)

- As we clearly see from these examples, concurrency is tricky, and here's the worst of it
  - it's ***very hard to discover*** race conditions using testing
  - and even once a test has found a bug, it may be ***very hard to localize*** it to the part of the program causing it
- Concurrency bugs exhibit ***very poor reproducibility***
  - it's hard to make them happen the same way twice
  - interleaving of instructions or messages depends on the *relative timing of events* that are strongly influenced by the environment
  - delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc
  - each time you run a program containing a race condition, you may get different behavior

Concurrency is Hard to Test and Debug (2)

- These kinds of bugs are **heisenbugs**, which are nondeterministic and hard to reproduce,
  - as opposed to a **bohrbug**, which shows up repeatedly whenever you look at it
  - almost all bugs in sequential programming are bohrbugs

- A heisenbug may even *disappear* when you try to look at it with println or debugger!
  - the reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving

Concurrency is Hard to Test and Debug (3)

- Inserting a simple print statement into the `cashMachine()`:

```
private static void cashMachine() {
     for (int i = 0; i < TRANSACTIONS_PER_MACHINE; i++) {
         deposit();  // put a dollar in
         withdraw(); // take it back out
         System.out.println(balance); // makes the bug disappear!
     }
}
```

  … and suddenly the balance is always 0, as desired, and the bug appears to disappear
  - but it's only *masked*, ***not*** truly fixed: a change in timing somewhere else in the program may suddenly make the bug come back

- Concurrency is hard to get right
  - but, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs *next week* !

Thank you for your attention !

- In this lecture, you have learned:
  - to make your data structure can be compared in multiple ways
  - to create a resizing hash table that enables you to add and search data in constant average time
  - to create good hash codes
  - about concurrency, shared-memory and message-passing paradigms, processes and threads, and race conditions

- Please continue to Lecture Quiz 12 and Lab 12:
  - to do Lab Exercise 12.1 - 12.4, and
  - to do Exercise 12.1 - 12.3