Collections – Part V

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Sources:
  - https://hbfs.wordpress.com/2012/03/30/finding-collisions/
  - https://stackoverflow.com/questions/53526790/why-are-hashmaps-implemented-using-powers-of-two

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

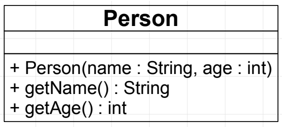
Have fun!

Associative Collections – A completely different Strategy – Part I

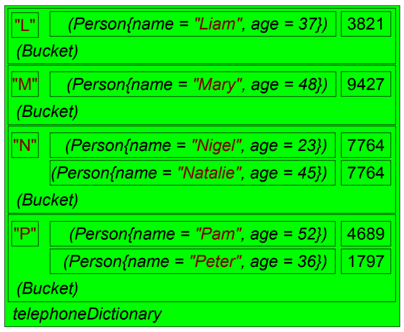
- Up to now we discussed associative collections using equivalence to manage keys.
  - We used a BST and got logarithmic cost for inserting and finding elements. – This is pretty fast!
  - An interesting point is that the organization of BSTs is concretely imaginable/understandable/"drawable".

- Now we're going to discuss associative collections using equality to manage keys, which is really cool!
  - Instead of keeping the associative collection sorted, we manage a set of collections that collect items having something in common.
  - These individual collections are then called buckets.

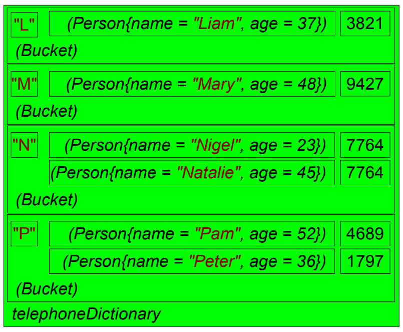
- Imagine the associative collection *telephoneDictionary*, which associates *Person*s to phone numbers.



  - This is our first approach:
    - Each bucket collects *Person*s, whose name have the same first character.
    - This is how "real" dictionaries work, where there are tabs collecting words starting with the same letter. The tabs play the role of the buckets.


"L"
(Person{name = "Liam", age = 37})
3821
(Bucket)
"M"
(Person{name = "Mary", age = 48})
9427
(Bucket)
"N"
(Person{name = "Nigel", age = 23})
7764
(Person{name = "Natalie", age = 45})
7764
(Bucket)
"P"
(Person{name = "Pam", age = 52})
4689
(Person{name = "Peter", age = 36})
1797
(Bucket)
telephoneDictionary

Associative Collections – A completely different Strategy – Part II


"L" (Person{name = "Liam", age = 37}) 3821
(Bucket)
"M" (Person{name = "Mary", age = 48}) 9427
(Bucket)
"N" (Person{name = "Nigel", age = 23}) 7764
(Person{name = "Natalie", age = 45}) 7764
(Bucket)
"P" (Person{name = "Pam", age = 52}) 4689
(Person{name = "Peter", age = 36}) 1797
(Bucket)
telephoneDictionary

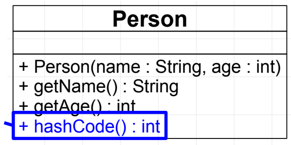
- This type of data structure is called hash table.
  - A so called hash function maps a key to a hash code.
  - In this case the hash function maps a *Person* (= the key) to the first letter of its name (= the hash code).
  - The hash function determines, in which bucket to find or into which bucket to insert a key or key-value-pair.
  - When the bucket is determined, a specific operation takes only place on the items in that bucket.

- The idea of the initial letter of the name is an oversimplification for a hash code, we'll sometimes refer back to this metaphor.

Hash Tables and Hash Codes

- OK, but how will an implementation using a hash table help? – To understand let's dissect hash table starting with the keys.
- The idea in hash tables is to use a index function that calculates a position in the hash table from the input item' hash code.
  - How can we have a function that is able to do that for an item? Item = key
  - The idea is to have the key-type, *Person* in our case, provide a method that allows to calculate its own position.
- Of course, an arbitrary object cannot know its position in an arbitrary table.
  - Instead we have to get there step by step. We begin by associating an integer code to every object (key).
  - This code can be somehow calculated from the object's data, this could be the sum of its numeric fields plus the lengths of its textual fields.
  - Such a code is called hash code. Let's implement this in the new method *hashCode()* for class *Person*:

```
public int hashCode() {
        return age + name.length();
}
```



  - Deviating from our initial example, we'll use the *age* and the length of the *name* instead of only the first letter of the *name* as hash code.
  - Now we can call *hashCode()* to get the hash codes of some *Person*s:

```
Person natalie = new Person("Natalie", 45);
int nataliesHashCode = natalie.hashCode();
// nataliesHashCode = 52
Person peter = new Person("Peter", 36);
int petersHashCode = peter.hashCode();
// petersHashCode = 41
```

Hash Codes – Part I

- Awesome! But what the hack is a hash code?

- Actually, a hash code is only a pretty short number/few data, which represents a larger set of data. Here some examples:
  - A hash code for an address like "Lisa Smith, Bakerstreet 17, K-Town" could be the ASCII-code of its first char: hash(address) = 76

```
int hash1 = "Lisa Smith, Bakerstreet 17, K-Town".charAt(0);
// hash1 = 76
```

  - Another hash code for an address could be the count of letters of the text: hash(address) = 34

```
int hash2 = "Lisa Smith, Bakerstreet 17, K-Town".length();
// hash2 = 34
```

  - Another hash code for an address could be the sum of ASCII-code of the letters in the text: hash(address) = 2922

```
int hash3 = "Lisa Smith, Bakerstreet 17, K-Town".chars().sum();
// hash3 = 2922
```

- Before we go on, let's derive some features of hash codes from the examples above:
  - A hash code is a value, that combines "all relevant" data of an object into a single number.
    - Remember, that for our initial example the first letter of a *Person*'s name was just enough, but in reality most often more data is relevant.
    - The examples of the address data above suggests, that hash code are a kind of cross sums or check sums and this is pretty correct.
  - A hash code can be calculated quickly.
  - Between the lines: the type/size of input data is different, but the resulting hash codes map down to a common size/type (32b/int).
    - In our example the length of text doesn't matter, the hash code will always fit into a 32b int!
    - It is difficult to impossible to reverse-engineer the original object from a hash code: we lost information of the original object, when coming to a hash code.

Hash Codes – Part II

- Again, awesome! But why do we need a hash code of an object?

- The dull answer is: We can use the hash code of the object to find this object quickly in a hash table.
  - <u>This closes the argumentation to the introduction of hash tables some slides ago.</u>

- And with hash codes we get another definition of hash table:
  - A hash table is a data structure <u>optimized to find a certain item quickly with O(1) complexity</u> in the best case.
    - Also a BST is a data structure optimized to find a certain item quickly, <u>but only with O(log n) complexity</u> in the best case.
  - To make this possible, items in a hash table can be accessed by an index, which is calculated with the hash code of the item.
  - Still a hash function maps a key to a hash code and an index function maps a hash code to a certain index.

- Before we go on, we have to define the type of the hash table: for now, it should be an indexed "collection" such as an array.
  - The connection of the hash code to the hash table as array is that array's index.

- For the time being we have a way to get a hash code from an object: we have the method *hashCode()*.
  - But we also need a way to get an index to the hash tables' array from that hash code...

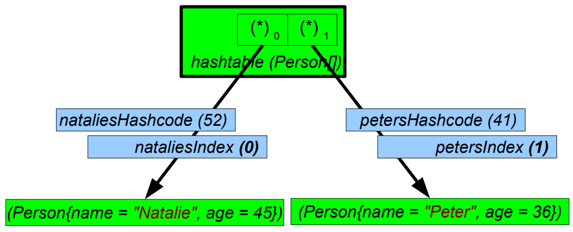
From Hash Code to Index

- A doable (but naïve) way for an index function to get valid indexes is using a simple remainder operation:

$$\boxed{index = hashcode \bmod hashtablesize}$$

- We can recapitulate the functionality of "finding an index" with a *Person*-array of size 2:
  - When we use the formula shown above, we "can come to indexes" into the hash table based on the hash code of the *Person*s.
  - As can be seen we come to indexes for *natalie* and *peter*:

```
Person[] hashtable = new Person[2];
int nataliesIndex = natalie.hashCode() % hashtable.length;
// nataliesIndex = 0
hashtable[nataliesIndex] = natalie;
int petersIndex = peter.hashCode() % hashtable.length;
// petersIndex = 1
hashtable[petersIndex] = peter;
```



- So far so good, but our idea of *hashCode()* has a little problem: the hash codes it produces are not unique for a *Person*:

```
Person natalie = new Person("Natalie", 45);
int nataliesHashCode = natalie.hashCode();
// nataliesHashCode = 52
Person peter = new Person("Peter", 36);
int petersHashCode = peter.hashCode();
// petersHashCode = 41
```

```
Person mary = new Person("Mary", 48);
int marysHashCode = mary.hashCode();
// marysHashCode = 52
Person liam = new Person("Liam", 37);
int liamsHashCode = liam.hashCode();
// liamsHashCode = 41
```

  - Is the hash code 41, *peter*'s hash code or *liam*'s? → It's the hash code of both!

9

Hash Collisions

- In case two objects have the same hash code while processing, we call this a hash collision.

```
Person natalie = new Person("Natalie", 45);
int nataliesHashCode = natalie.hashCode();
// nataliesHashCode = 52
Person peter = new Person("Peter", 36);
int petersHashCode = peter.hashCode();
// petersHashCode = 41
```

```
Person mary = new Person("Mary", 48);
int marysHashCode = mary.hashCode();
// marysHashCode = 52
Person liam = new Person("Liam", 37);
int liamsHashCode = liam.hashCode();
// liamsHashCode = 41
```

  - Hash collisions are a normal thing: somewhen objects must have the same hash code.
    - Hash codes are ints and the count of available ints is usually limited!

- Hash collisions cannot be avoided, therefor hash tables have to deal with hash collisions.
  - Hash tables can apply strategies to reduce the probability of hash collisions by re hashing, but still hash collisions are unavoidable.
  - It should also give you a feeling, why the first letter of the name would be a bad hash code: we would get many collisions!

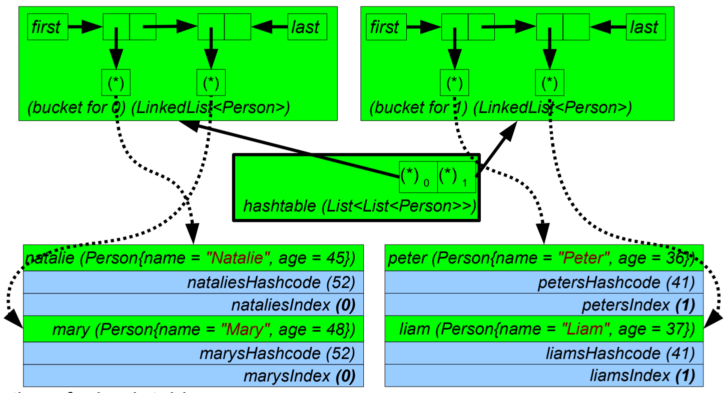
- The idea to handle hash collisions is simple: objects with the same hash code are collected in buckets.
  - So we have to get away from our idea of a plain array of *Person*s, we rather need an array of *Person*-arrays.
  - Actually, we will also get away from the idea of arrays, but rather use *List* and *List<List>*, because we require efficient resizing abilities.
    - It is needed to add and remove buckets and also to add and remove items into/from the buckets.
  - Each person "landing" at a specific index in the bucket-*List* (because of its hash code) will be stored in the *List<Person>* at that index.
    - Buckets can be understood as groups of objects having the hash code in common.
    - The tabs we found in the telephone dictionary represent the buckets: the tabs contain a list of *Person*s with common initials.

10

Buckets

- Without further ado: here our hash table with buckets expressed as *List<List<Person>>*:

```
List<List<Person>> hashtable = new ArrayList<>(2);
hashtable.add(new LinkedList<>()); // Bucket for index 0.
hashtable.add(new LinkedList<>()); // Bucket for index 1.

int nataliesIndex = natalie.hashCode() % hashtable.size();
// nataliesIndex = 0
hashtable.get(nataliesIndex).add(natalie); // natalie to the bucket for 0
int petersIndex = peter.hashCode() % hashtable.size();
// petersIndex = 1
hashtable.get(petersIndex).add(peter); // peter to the bucket for 1
int marysIndex = mary.hashCode() % hashtable.size();
// marysIndex = 0
hashtable.get(marysIndex).add(mary); // mary to the bucket for 0
int liamsIndex = liam.hashCode() % hashtable.size();
// liamsIndex = 1
hashtable.get(liamsIndex).add(liam); // liam to the bucket for 1
```


first
last
(*)
(*)
(bucket for 0) (LinkedList<Person>)
first
last
(*)
(*)
(bucket for 1) (LinkedList<Person>)
(*) 0
(*) 1
hashtable (List<List<Person>>)
natalie (Person{name = "Natalie", age = 45})
nataliesHashcode (52)
nataliesIndex (0)
mary (Person{name = "Mary", age = 48})
marysHashcode (52)
marysIndex (0)
peter (Person{name = "Peter", age = 36})
petersHashcode (41)
petersIndex (1)
liam (Person{name = "Liam", age = 37})
liamsHashcode (41)
liamsIndex (1)

- What we see here is a <u>general purpose implementation of a hash table</u>:
  - It maintains an indexed collection of buckets, usually one that can grow to accommodate many objects to collect.
  - It uses the hash codes of objects to get indexes of the hash table as slots for the objects.
  - It uses a kind of remainder operation to get an index of an object from its hash code and the length of the hash table.
  - It uses collections (usually linked lists) as buckets to handle hash collisions to put objects with the same hash into that buckets.

Getting an Item in a Hash Table – Part I

- We know how storing objects in hash tables works, but the picture is not complete: How do we find objects in a hash table?

```
int liamsBucketIndex = liam.hashCode() % hashtable.size();
Person liamOfTheHashtable = null;
for (Person person : hashtable.get(liamsBucketIndex)) {
        if (person.getAge() == liam.getAge() && person.getName().equals(liam.getName())) {
                liamOfTheHashtable = person;
                break;
        }
}
```

- Ok, so what we basically have to do is:
  - (1) Get the index of the bucket in which *liam*, if it is contained in the hash table, must reside. We do this with *liam*'s hash code.
  - (2) If the bucket exists, we have to iterate over potentially all items in that bucket to find *liam* specified by its *age* and *name*.
- Hm, but how does this a better job, at least complexity-/performance-wise than a *TreeMap*?
  - (We also have to mention that we didn't even associate a value with the key held in the hash table, but that is a pretty small piece…)
  - Esp. step (2) is pretty expensive: searching through the bucket is a O(n) operation, because it is just a linear search (a loop).
  - What is so thrilling about the hash table to be relevant?
- The mighty aspect of hash table is in step (1), associating an object to a bucket, which is an O(1) operation!
  - The goal: as least items as possible in the buckets. – If each bucket only contains one item, hash tables have overall access-complexity of O(1)!
  - I.e., the more we minimize the probability of hash collisions, the more access-complexity develops against O(1)!

Getting an Item in a Hash Table – Part II

- For hash table, O(1) access-complexity is in reach: Java's *HashMap* (hash-table-based implementation of *Map*) is able to get there.

- But for *HashMap* to work, it makes <u>claims</u> against the types, which are used as key-types.
  - In this aspect *HashMap* is like *TreeMap*, but *TreeMap* claims types to either implement *Comparable* or provide *Comparator*s.
  - Remember: when we underscored the idea of associative collections, we concluded that assoc. collections <u>inspect their items (keys)</u>.
  - *TreeMap*s call comparison-methods (e.g. *Comparable.compareTo()*) to analyze the relative of order of keys to find keys in a BST.

- All right, but what must a *HashMap* call or "do" to find keys in its hash table? We can identify it in our recent code:

```java
int liamsBucketIndex = liam.hashCode() % hashtable.size();
Person liamOfTheHashtable = null;
for (Person person : hashtable.get(liamsBucketIndex)) {
	if (person.getAge() == liam.getAge() && person.getName().equals(liam.getName())) {
		liamOfTheHashtable = person;
		break;
	}
}
```

  - This is the search algorithm we coded to find a certain *Person* in the hash table. Three methods are called on the key:
    - *Person.hashCode()*
    - *Person.getAge()*
    - *Person.getName()*
  - Hm, it does not yet make fully sense, esp. calling *getAge()* and *getName()* cannot really express a more <u>common pattern</u> ...

Getting an Item in a Hash Table – Part III

- However, when we analyze why *getAge()* and *getName()* are called we can recognize the bigger picture:

```
int liamsBucketIndex = liam.hashCode() % hashtable.size();
Person liamOfTheHashtable = null;
for (Person person : hashtable.get(liamsBucketIndex)) {
        if (person.getAge() == liam.getAge() && person.getName().equals(liam.getName())) {
                liamOfTheHashtable = person;
                break;
        }
}
```

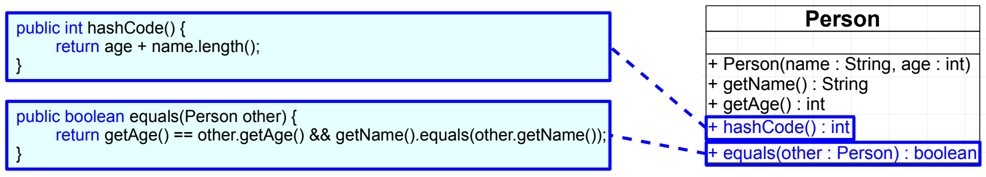
- It boils down to:
  - (1) Apply the hash function, i.e. getting the hash code via *Person.hashCode()*.
    - (1.1) Apply the index function (remainder-operation in our case) to the hash code to get the index of the belonging to bucket.
    - (1.2) Select the bucket.
  - (2) Compare two *Person*-instances "somehow" (each *Person* of a bucket with the specific *Person* to find).
    - "Somehow" in our case means compare *age*s and *name*s.

- There is a "deeper truth" behind the hash code of an object and the comparison of objects to be used in a hash table.
  - … and this does not mean equivalence-comparison as we have it with *Comparable*/*Comparator*!

- This "deeper truth" is the so-called "equals contract" (sometimes also "hash-code-contract") in Java (and also in .NET).
  - On the following slides we'll gradually get to the equals contract.

Hash Table – Equality

- As a matter of fact hash collisions can not be avoided, because any objects could have the same hash code!
  - E.g. two names could have the same initials or the length of the name and the age have the same sum. This is not under our control!
- The answer is that potentially all objects of a "hash-code-matching-bucket" need to be searched for a certain object.
  - The objects in a "hash-code-matching-bucket" are linearly searched and compared for equality to find the exactly matching object.
  - But what means "exactly matching object" in opposite to a (just) "hash-code-matching object"?
  - We have to prepare our objects' type to provide another method to check for the exact equality of two objects!
- We factor out the "somehow"-comparison of *Person*s into the method *Person.equals()*, it exists besides *Person.hashCode()*:



```
public int hashCode() {
        return age + name.length();
}
```

```
public boolean equals(Person other) {
        return getAge() == other.getAge() && getName().equals(other.getName());
}
```

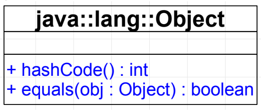
Person
+ Person(name : String, age : int)
+ getName() : String
+ getAge() : int
+ hashCode() : int
+ equals(other : Person) : boolean

Hash Table – Equals Contract – Part I

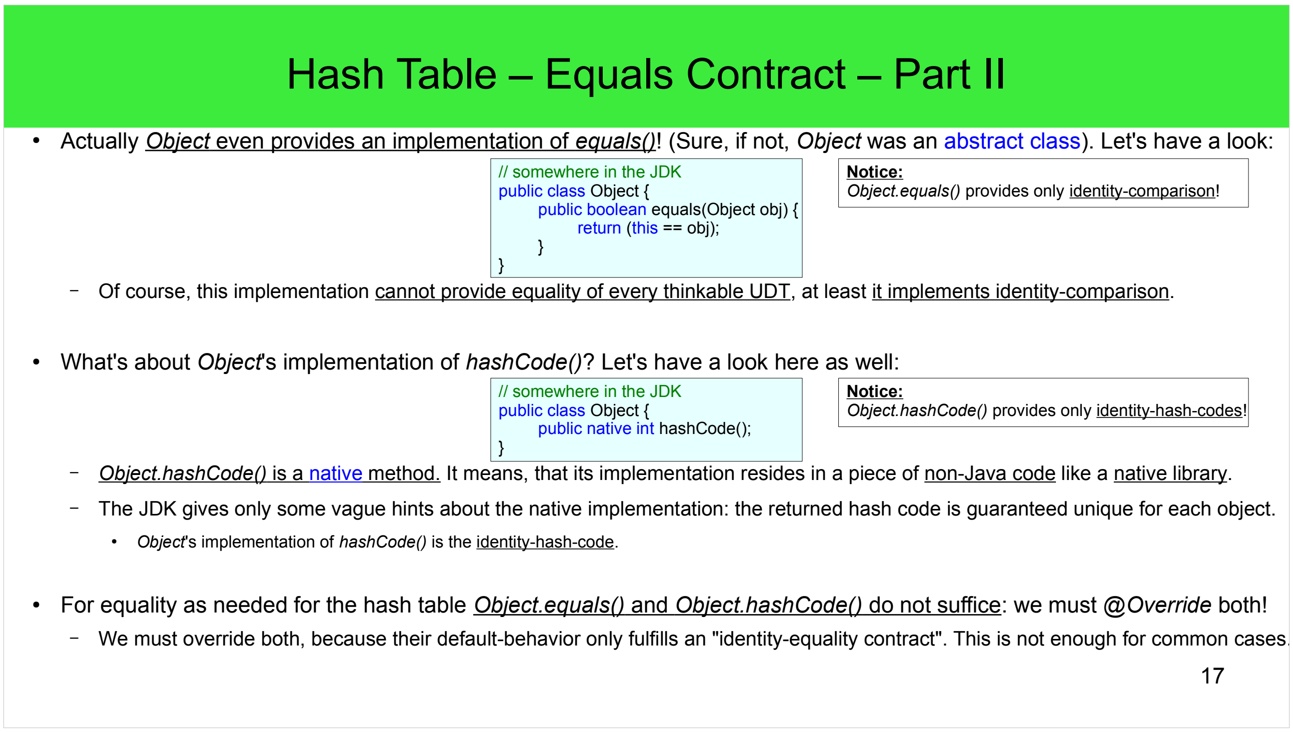
- Having *Person.equals()* factored out, we come to this code to find a certain *Person* in the hash table:

```
int liamsBucketIndex = liam.hashCode() % hashtable.size();
Person liamOfTheHashtable = null;
for (Person person : hashtable.get(liamsBucketIndex)) {
        if (person.equals(liam)) {
                liamOfTheHashtable = person;
                break;
        }
}
```

- The equals contract is important in Java, so important, that the methods *hashCode()* and *equals()* <u>are leveraged to *Object*</u>:



- Actually a type fulfilling the <u>equals contract</u> in Java (and also in .NET) <u>must @*Override equals()* and *hashCode()*</u>.
  - On the following slides, we'll learn how this works basically.



Hash Table – Equals Contract – Part II

- Actually *Object* even provides an implementation of *equals()*! (Sure, if not, *Object* was an abstract class). Let's have a look:

```
// somewhere in the JDK
public class Object {
        public boolean equals(Object obj) {
                return (this == obj);
        }
}
```

Notice:
Object.equals() provides only identity-comparison!

  - Of course, this implementation cannot provide equality of every thinkable UDT, at least it implements identity-comparison.

- What's about *Object*'s implementation of *hashCode()*? Let's have a look here as well:

```
// somewhere in the JDK
public class Object {
        public native int hashCode();
}
```

Notice:
Object.hashCode() provides only identity-hash-codes!

  - *Object.hashCode()* is a native method. It means, that its implementation resides in a piece of non-Java code like a native library.
  - The JDK gives only some vague hints about the native implementation: the returned hash code is guaranteed unique for each object.
    - *Object*'s implementation of *hashCode()* is the identity-hash-code.

- For equality as needed for the hash table *Object.equals()* and *Object.hashCode()* do not suffice: we must @*Override* both!
  - We must override both, because their default-behavior only fulfills an "identity-equality contract". This is not enough for common cases.

17

- Java's and .NET's inherited *equals()*/*Equals()* (inherited from *Object* esp. for .NET reference types) will just compare references. It means that *equals()*/*Equals()* will only return true, if the same object is compared to itself, so the inherited implementations do an identity comparison.
- Java's inherited *hashCode()* (inherited from *Object*) just returns the object's address in memory. .NET's inherited *GetHashCode()* (inherited from *Object* i.e. for .NET reference types) uses some number that is guaranteed to be unique within the .NET *AppDomain*.

Hash Table – Equals Contract – Part III

- Let's *@Override hashCode()* and *equals()* from *Object*:

```
// somewhere
public class Person {
	@Override
	public int hashCode() {
		return age + name.length();
	}
	@Override
	public boolean equals(Object other) {
		// Simple and naïve implementation of equals():
		Person otherPerson = (Person) other;
		return getAge() == otherPerson.getAge() && getName().equals(otherPerson.getName());
	}
}
```

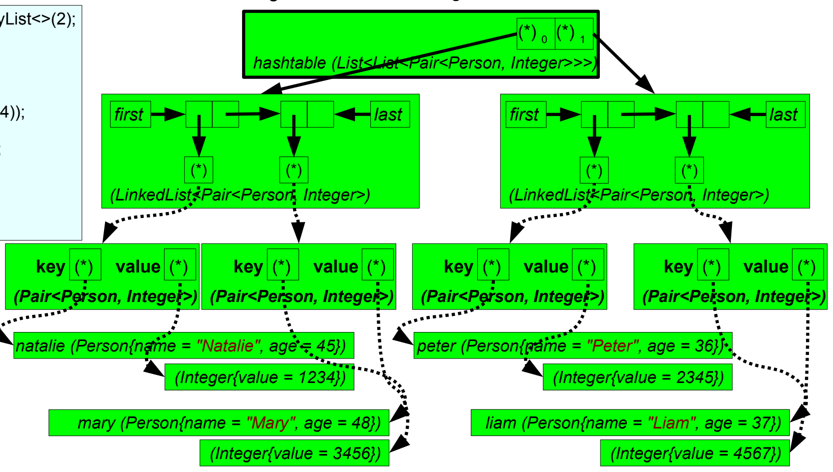
- As can be seen in the *@Override*, we leave *Person.hashCode()*'s implementation as before: it is appropriate!

- The situation is more involved when overriding *Object.equals()*.
  - *Object.equals()* accepts an *Object* in its signature and in *@Override*s we cannot change the type of parameters.
    - => In Java, parameter types are invariant in overriding methods.
  - Therefor, we have to accept an *Object* also in *Person.equals()* and to downcast this parameter to access *other*'s interface as *Person*.
  - As stated in the comment this implementation of *Object.equals()* is naïve. Why is is naïve will be discussed in another episode.

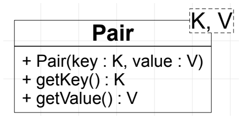
Don't forget the Pairs – Part I

- Our implementation of a hash table as associative collection is not yet complete!
  - We don't have to forget, that we want to associate a key to a value. Up to here we only discussed how to store and find the keys.
  - We can add this aspect pretty simply: instead of *Person*s we will manage *Pair<Person, Integer>*-instances in the *List*:

```
List<List<Pair<Person, Integer>>> hashtable = new ArrayList<>(2);
hashtable.add(new LinkedList<>());
hashtable.add(new LinkedList<>());

int nataliesIndex = natalie.hashCode() % hashtable.size();
hashtable.get(nataliesIndex).add(new Pair<>(natalie, 1234));
int petersIndex = peter.hashCode() % hashtable.size();
hashtable.get(petersIndex).add(new Pair<>(peter, 2345));
int marysIndex = mary.hashCode() % hashtable.size();
hashtable.get(marysIndex).add(new Pair<>(mary, 3456));
int liamsIndex = liam.hashCode() % hashtable.size();
hashtable.get(liamsIndex).add(new Pair<>(liam, 4567));
```



- The code shows, how the association must be done.



Don't forget the Pairs – Part II

- We know how to associate values to keys with our *List<List<Pair<Person, Integer>>>*-based implementation.
  - Now we'll see how we get back an associated value, i.e. doing the lookup, e.g. looking up *liam*'s phone number:

```
int liamsPhoneNumber = 0;
for (Pair<Person, Integer> personAndPhoneNumber : hashtable.get(liamsIndex)) {
        if (liam.equals(personAndPhoneNumber.getKey())) {
                liamsPhoneNumber = personAndPhoneNumber.getValue();
                break;
        }
}
System.out.println(liamsPhoneNumber);
```

- Sure, this usage of a hash table is pretty "explicit" and not easy to use.
  - We understand, that what we basically have is a hash-table-based associative collection.
  - "The" interface in Java, which supports associative collections in Java is *Map* …
  - … and actually Java offers a hash-table-based implementation of *Map*, the *HashMap*, as alternative to *TreeMap*.

- Now we have a deeper understanding of how hash tables work, we can officially introduce *HashMap*.

Sneak Preview to HashMap

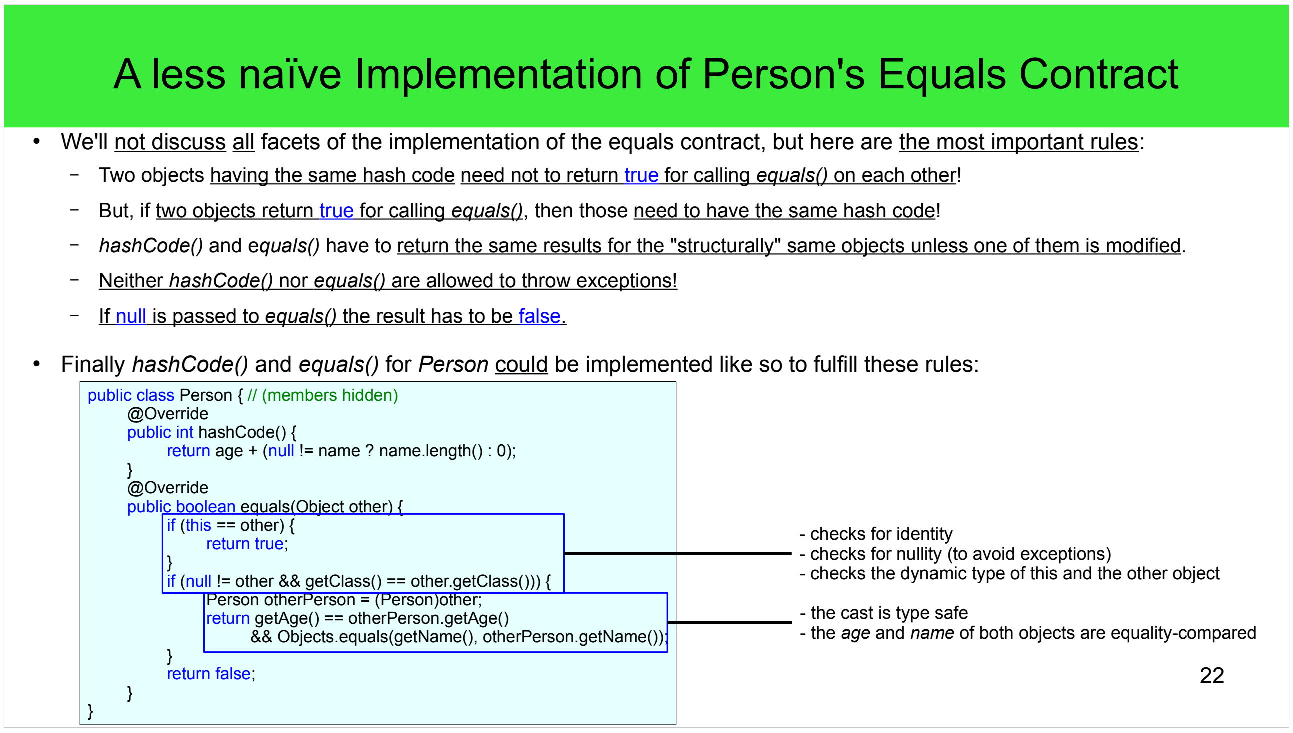
- In Java we can use *Person* as key type in a *Map* that is implemented in terms of a hash table.
  - So, Java's hash-table-based implementation of *Map* is *HashMap*! It can (of course) be used like any other *Map*:

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
```

  - When *hashCode()* and *equals()* are overridden appropriately in the key type *Person HashMap* will work automatically:

```
telephoneDictionary.put(liam, 7764);
telephoneDictionary.put(natalie, 3821);
```

- Java's *HashMap* principally works like the hash table we have just implemented in terms of a *List<LinkedList<Pair<>>>*.
  - Commonalities:
    - *HashMap* uses buckets of keys to deal with hash-collisions.
  - Differences:
    - *HashMap* is pre-allocated with 16 slots in the hash table.
    - *HashMap* processes the hash codes so that hash collisions are minimized.
    - *HashMap* automatically switches from a linked list to a BST-based collection, if a bucket grows over 8 items and the key type implements *Comparable<T>*.



A less naïve Implementation of Person's Equals Contract

- We'll not discuss all facets of the implementation of the equals contract, but here are the most important rules:
  - Two objects having the same hash code need not to return true for calling *equals()* on each other!
  - But, if two objects return true for calling *equals()*, then those need to have the same hash code!
  - *hashCode()* and *equals()* have to return the same results for the "structurally" same objects unless one of them is modified.
  - Neither *hashCode()* nor *equals()* are allowed to throw exceptions!
  - If null is passed to *equals()* the result has to be false.
- Finally *hashCode()* and *equals()* for *Person* could be implemented like so to fulfill these rules:

```
public class Person { // (members hidden)
        @Override
        public int hashCode() {
                return age + (null != name ? name.length() : 0);
        }
        @Override
        public boolean equals(Object other) {
                if (this == other) {
                        return true;
                }
                if (null != other && getClass() == other.getClass()) {
                        Person otherPerson = (Person)other;
                        return getAge() == otherPerson.getAge()
                                && Objects.equals(getName(), otherPerson.getName());
                }
                return false;
        }
}
```

- checks for identity
- checks for nullity (to avoid exceptions)
- checks the dynamic type of this and the other object

- the cast is type safe
- the *age* and *name* of both objects are equality-compared

22

- *hashCode()* and *equals()* should be implemented to work very fast.

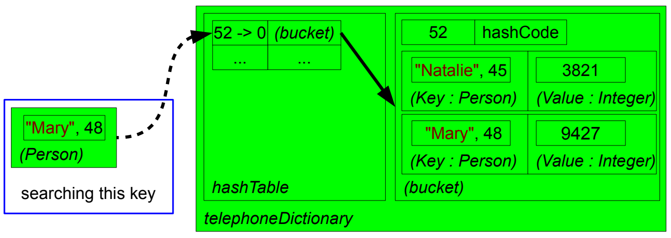
The Lookup-Algorithm of HashMap

- All right! Now we have *Person*'s equals contract in place. But how do *HashMaps* use these tools to work?
- Let's assume following content in the *HashMap telephoneDictionary*:

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
telephoneDictionary.put(new Person("Natalie", 45), 3821);
telephoneDictionary.put(new Person("Mary" , 48), 9427);
```

  - Then we'll lookup/search the phone number of *mary*:

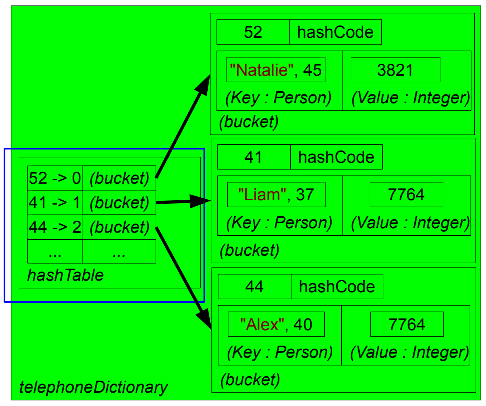
```
int no = telephoneDictionary.get(mary);
```



- The lookup will initiate following algorithm basically:
  - *HashMap* will call *hashCode()* on the *HashMap.get()*'s argument *mary* and the result is 52.
  - *HashMap* calculates the index of the internal hash table from the hash code, e.g. 0 in this example.
  - *HashMap* will get the bucket at the hash table's index 0. This bucket has two entries: *Person("Natalie", 45)* has the same hash code!
  - *HashMap* will call '*equals(mary)*' against each key of the key-value-pairs in the just returned bucket.
  - The value of the key-value-pair for which the key was equal to *mary* will be returned.
- Read these steps for multiple times and make sure you understood those! Now we have to discuss some details...
  - The algorithms to insert or update a key-value-pair work the same way as for lookups!

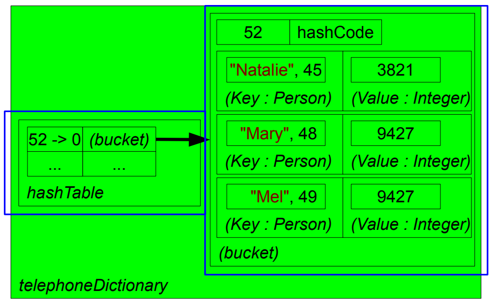
HashMap's best Case Complexity

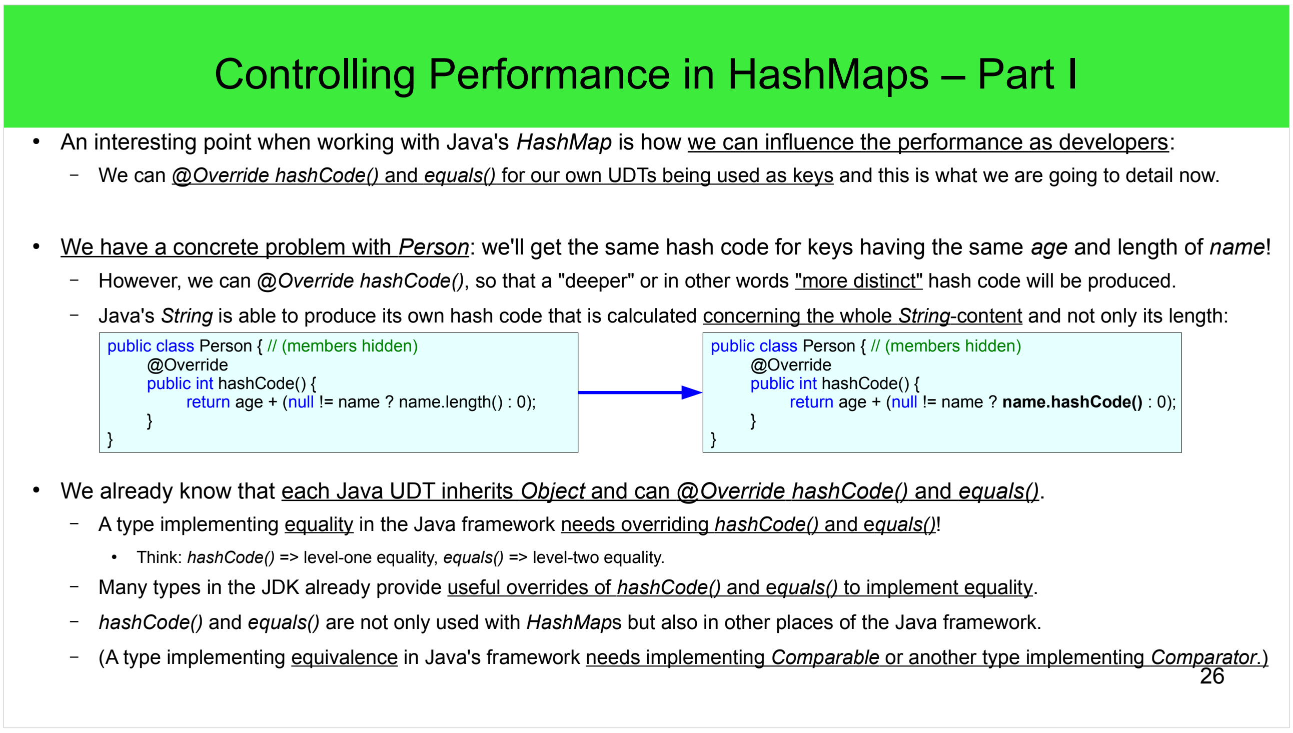
- Although the algorithm to lookup keys is complex, the analysis of a hash table's complexity is really simple!
- The best case yields a complexity of O(1)! – Wow!
  - If every item can be associated to exactly one distinct bucket each, we have one item in every bucket: an 1 : 1 association between items and buckets.
  - This means that every item has a distinct hash code and thus a distinct index in the hash table.
  - As the hash table is an indexed collection can access their items by index with O(1) complexity, we have the best case for hash table!
- In the best case, accessing a hash table, means accessing an indexed collection by index with constant complexity.
  - This is better than O(log n) for BST-based associative collections!
  - This is the best we can get for collections at all!
- But there is also a worst case!



HashMap's worst Case Complexity

- The worst case yields a complexity of O(n)! – Oh no!
  - If every item can be associated to the same bucket, we have all items in only one single bucket: an n : 1 association between items and buckets.
  - This means that every item has the same hash code and thus the same index in the hash table.
  - As the only one bucket needs to be search linearly to find the equal key, the worst complexity boils down to the linear complexity O(n)!
- In the worst case the workload is moved to a single bucket that must be searched in a linear manner.





Controlling Performance in HashMaps – Part I

- An interesting point when working with Java's *HashMap* is how we can influence the performance as developers:
  - We can *@Override hashCode()* and *equals()* for our own UDTs being used as keys and this is what we are going to detail now.
- We have a concrete problem with *Person*: we'll get the same hash code for keys having the same *age* and length of *name*!
  - However, we can *@Override hashCode()*, so that a "deeper" or in other words "more distinct" hash code will be produced.
  - Java's *String* is able to produce its own hash code that is calculated concerning the whole *String*-content and not only its length:

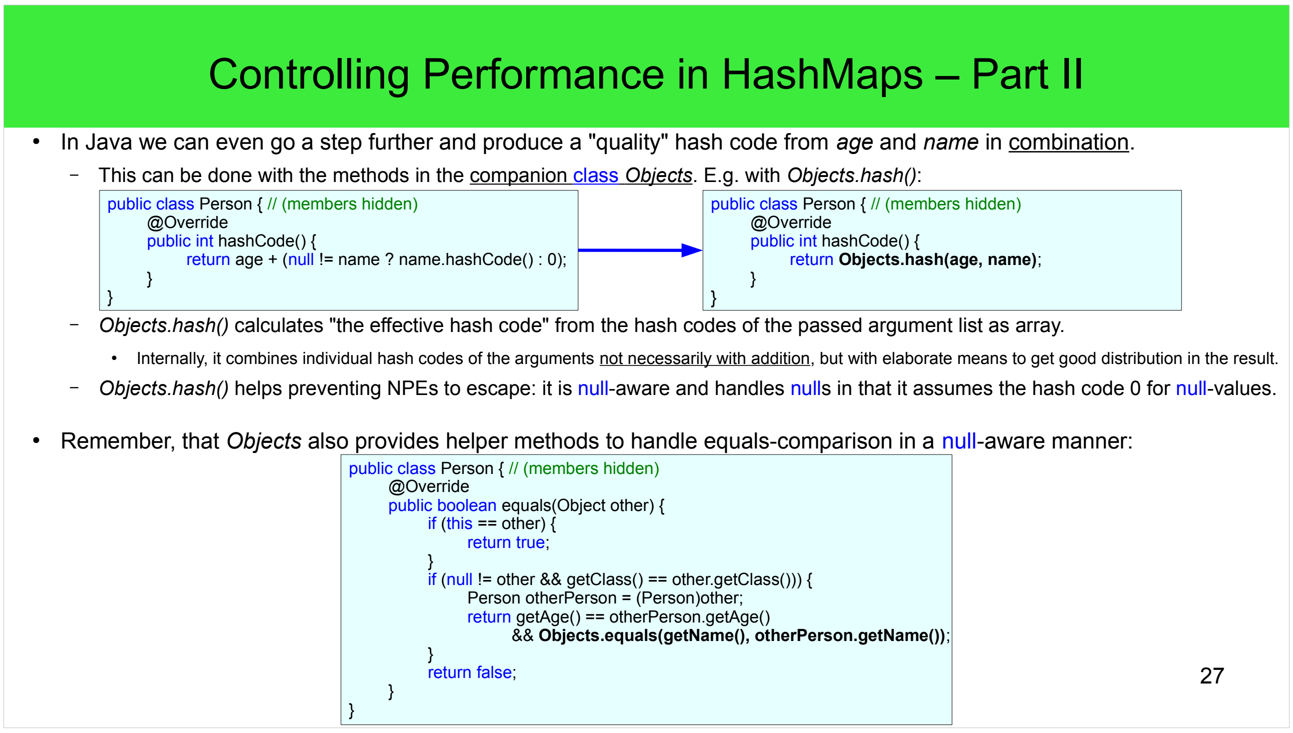
```java
public class Person { // (members hidden)
	@Override
	public int hashCode() {
		return age + (null != name ? name.length() : 0);
	}
}
```

```java
public class Person { // (members hidden)
	@Override
	public int hashCode() {
		return age + (null != name ? name.hashCode() : 0);
	}
}
```

- We already know that each Java UDT inherits *Object* and can *@Override hashCode()* and *equals()*.
  - A type implementing equality in the Java framework needs overriding *hashCode()* and *equals()*!
    - Think: *hashCode()* => level-one equality, *equals()* => level-two equality.
  - Many types in the JDK already provide useful overrides of *hashCode()* and *equals()* to implement equality.
  - *hashCode()* and *equals()* are not only used with *HashMap*s but also in other places of the Java framework.
  - (A type implementing equivalence in Java's framework needs implementing *Comparable* or another type implementing *Comparator*.)

26

- switch-case with strings in C# and Java uses the strings' hash code to do the comparisons.



Controlling Performance in HashMaps – Part II

- In Java we can even go a step further and produce a "quality" hash code from *age* and *name* in combination.
  - This can be done with the methods in the companion class *Objects*. E.g. with *Objects.hash()*:

```
public class Person { // (members hidden)
	@Override
	public int hashCode() {
		return age + (null != name ? name.hashCode() : 0);
	}
}
```

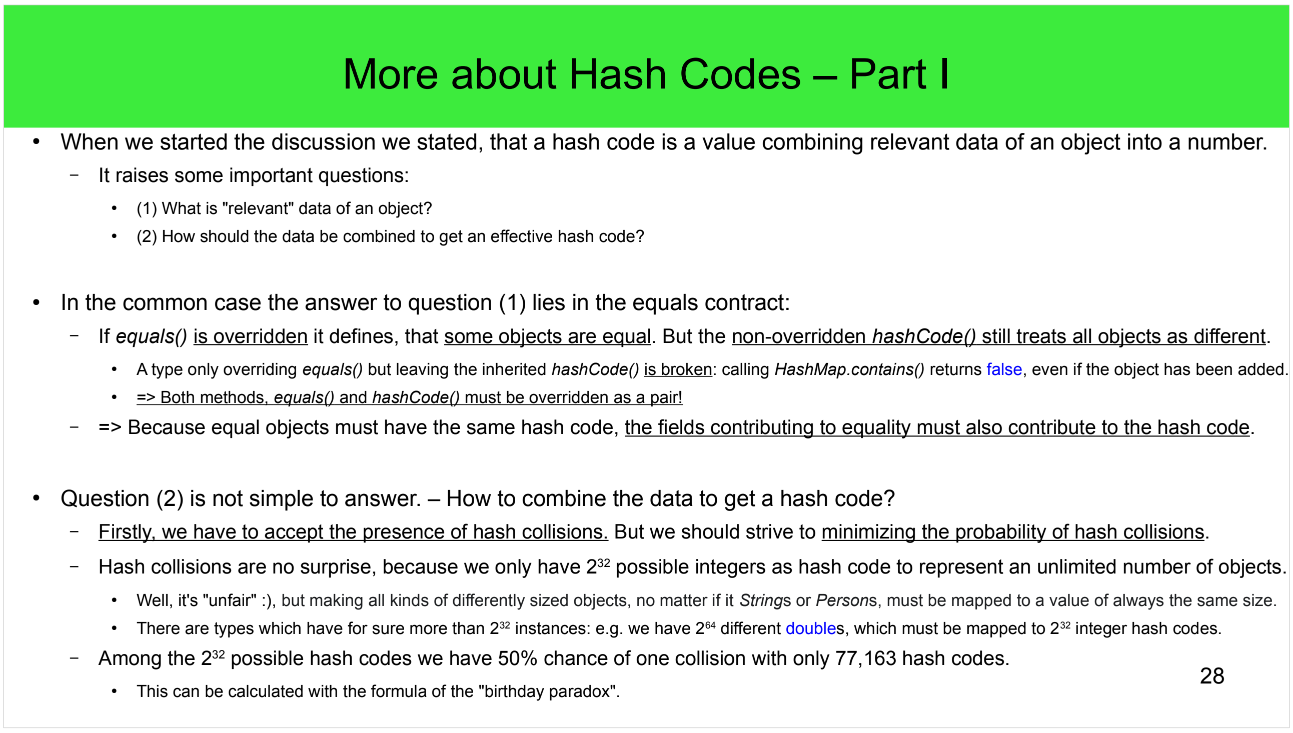
```
public class Person { // (members hidden)
	@Override
	public int hashCode() {
		return Objects.hash(age, name);
	}
}
```

  - *Objects.hash()* calculates "the effective hash code" from the hash codes of the passed argument list as array.
    - Internally, it combines individual hash codes of the arguments not necessarily with addition, but with elaborate means to get good distribution in the result.
  - *Objects.hash()* helps preventing NPEs to escape: it is null-aware and handles nulls in that it assumes the hash code 0 for null-values.
- Remember, that *Objects* also provides helper methods to handle equals-comparison in a null-aware manner:

```
public class Person { // (members hidden)
	@Override
	public boolean equals(Object other) {
		if (this == other) {
			return true;
		}
		if (null != other && getClass() == other.getClass()) {
			Person otherPerson = (Person)other;
			return getAge() == otherPerson.getAge()
				&& Objects.equals(getName(), otherPerson.getName());
		}
		return false;
	}
}
```

27

- *Objects.hash()* has an important downside: if value-types are passed, they are getting boxed, which does not come for free.



More about Hash Codes – Part I

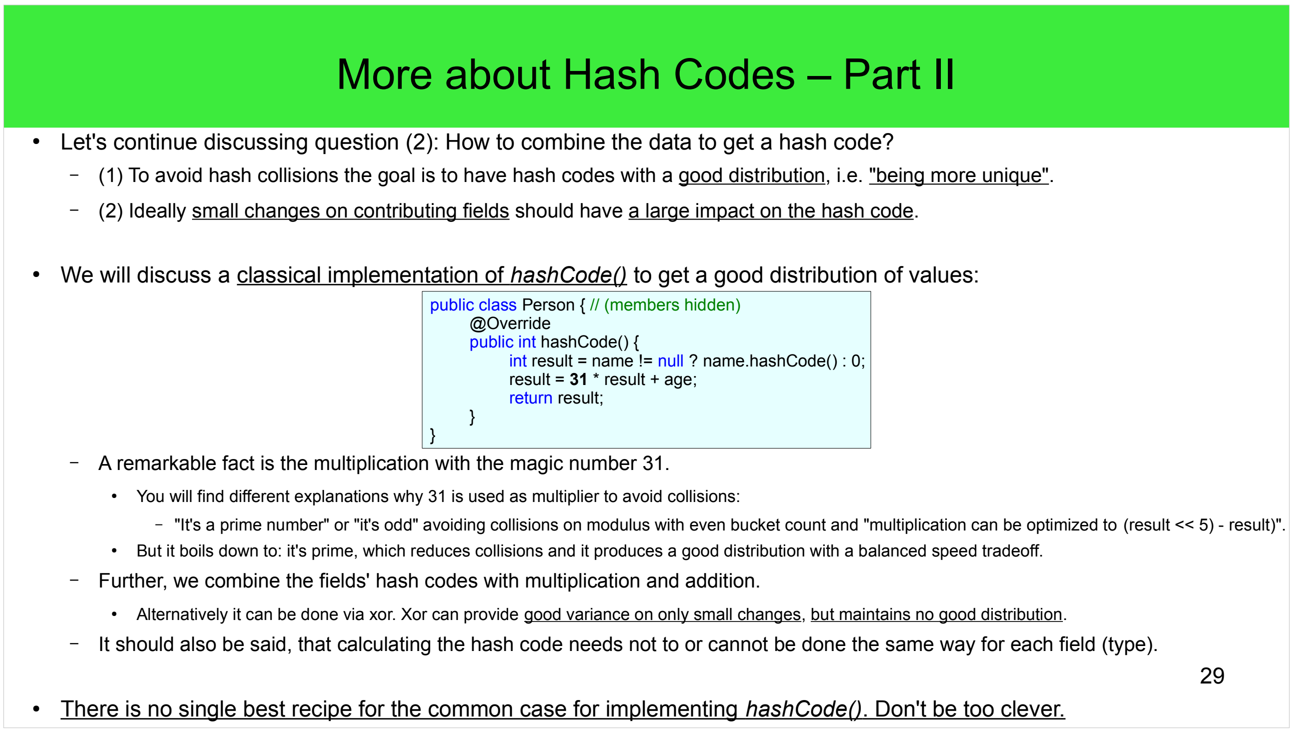
- When we started the discussion we stated, that a hash code is a value combining relevant data of an object into a number.
  - It raises some important questions:
    - (1) What is "relevant" data of an object?
    - (2) How should the data be combined to get an effective hash code?
- In the common case the answer to question (1) lies in the equals contract:
  - If *equals()* is overridden it defines, that some objects are equal. But the non-overridden *hashCode()* still treats all objects as different.
    - A type only overriding *equals()* but leaving the inherited *hashCode()* is broken: calling *HashMap.contains()* returns false, even if the object has been added.
    - => Both methods, *equals()* and *hashCode()* must be overridden as a pair!
  - => Because equal objects must have the same hash code, the fields contributing to equality must also contribute to the hash code.
- Question (2) is not simple to answer. – How to combine the data to get a hash code?
  - Firstly, we have to accept the presence of hash collisions. But we should strive to minimizing the probability of hash collisions.
  - Hash collisions are no surprise, because we only have 2^{32} possible integers as hash code to represent an unlimited number of objects.
    - Well, it's "unfair" :), but making all kinds of differently sized objects, no matter if it *String*s or *Person*s, must be mapped to a value of always the same size.
    - There are types which have for sure more than 2^{32} instances: e.g. we have 2^{64} different doubles, which must be mapped to 2^{32} integer hash codes.
  - Among the 2^{32} possible hash codes we have 50% chance of one collision with only 77,163 hash codes.
    - This can be calculated with the formula of the "birthday paradox".

28

- The birthday paradox describes, that with 23 persons the chance to find at least 2 persons with the same birthday is greater than 50%! When we apply this to 32b hash codes, it means that with 77,163 different objects, there is a 50% chance for a collision. The formula to find this:

$$n=\frac{1}{2}\left(\sqrt{1-8\,d\ln(1-\alpha)}\right)$$

  - *d*: count of differing hash codes, *α*: probability of collision
  - With *d* = 2^{32} and *α* = 0.5 (50%) we get:
    0.5 * sqrt(1 - 8 * 2^{32} * ln(0.5)) = 77,163



More about Hash Codes – Part II

- Let's continue discussing question (2): How to combine the data to get a hash code?
  - (1) To avoid hash collisions the goal is to have hash codes with a good distribution, i.e. "being more unique".
  - (2) Ideally small changes on contributing fields should have a large impact on the hash code.

- We will discuss a classical implementation of *hashCode()* to get a good distribution of values:

```
public class Person { // (members hidden)
	@Override
	public int hashCode() {
		int result = name != null ? name.hashCode() : 0;
		result = 31 * result + age;
		return result;
	}
}
```

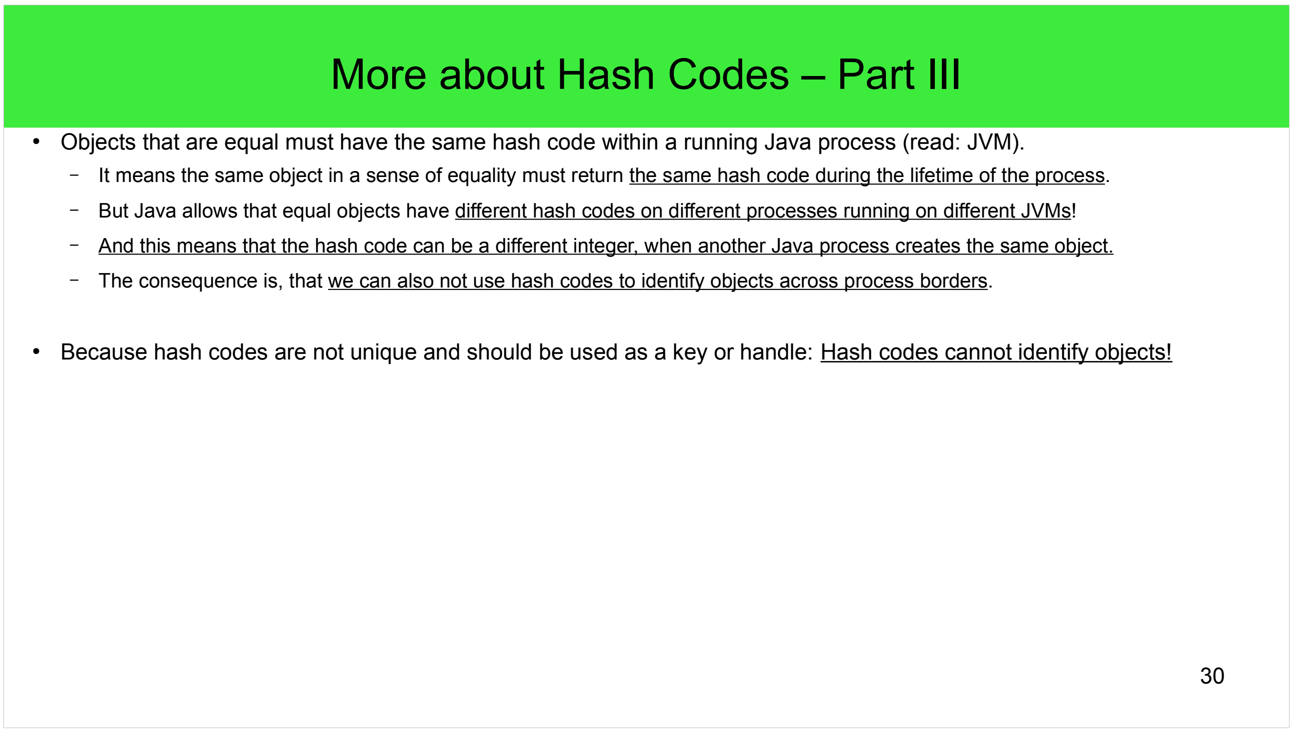
  - A remarkable fact is the multiplication with the magic number 31.
    - You will find different explanations why 31 is used as multiplier to avoid collisions:
      - "It's a prime number" or "it's odd" avoiding collisions on modulus with even bucket count and "multiplication can be optimized to (result << 5) - result)".
    - But it boils down to: it's prime, which reduces collisions and it produces a good distribution with a balanced speed tradeoff.
  - Further, we combine the fields' hash codes with multiplication and addition.
    - Alternatively it can be done via xor. Xor can provide good variance on only small changes, but maintains no good distribution.
  - It should also be said, that calculating the hash code needs not to or cannot be done the same way for each field (type).

29

- There is no single best recipe for the common case for implementing *hashCode()*. Don't be too clever.

- The idea behind many suggestions we can find in books and the WWW do often rely on special assumptions of the platform and the implementation. E.g. modern JVMs are able to optimize multiplications on their own, which elides any performance benefits using 31 as magic factor.
- The number 31 is found in many implementations, incl. those found in Joshua Bloch's Book "Effective Java".
  - The 31 was also used in *String.hashCode()*-implementations of the first Java-versions. It is said, that 31 yielded the best distribution of hashes of terms of the English Merriam-Webster dictionary. – This should give a hint how specific the "distributive quality" of the factor 31 might be (English terms).
  - Multiplication with 31 with bit-shifting:

```
int number = 15;
int resultA = 31 * number;
int resultB = (number << 5) - number;
assert resultA == resultB;
```



More about Hash Codes – Part III

- Objects that are equal must have the same hash code within a running Java process (read: JVM).
  - It means the same object in a sense of equality must return the same hash code during the lifetime of the process.
  - But Java allows that equal objects have different hash codes on different processes running on different JVMs!
  - And this means that the hash code can be a different integer, when another Java process creates the same object.
  - The consequence is, that we can also not use hash codes to identify objects across process borders.

- Because hash codes are not unique and should be used as a key or handle: Hash codes cannot identify objects!

30

- Java's inherited *hashCode()* (inherited from *Object*) just returns the object's address in memory. .NET's inherited *GetHashCode()* (inherited from *Object* i.e. for .NET reference types) uses some number that is guaranteed to be unique within the .NET *AppDomain*.

Be aware of mutable Key-Types! – Part I

- So, the key-types of a *HashMap* are better off implementing the equals contract to be useful, but that's not all!

- We pondered about the fields, which should take part in getting the hash code and in equality.
  - The thing which seems important is that fields contributing to the hash code (*hashCode()*) should also contribute to equality (*equals()*).

- But there is another aspect: What happens to an object if fields "bound to the equality contract" are modified?
  - Sure, the object will no longer be equal to its version before the modification.
  - And also the hash code should be another one.

- But if an object modified in such a way is used as key in a *HashMap* we have a big problem! Consider this:

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
Person mary = new Person("Mary", 48);
telephoneDictionary.put(mary, 3456);
Integer marysNumber = telephoneDictionary.get(mary);
// marysNumber = 3456
```

  - We can rely on *HashMap* to find *mary*'s phone number right after we had added it. – No Problem!
  - After we added *mary*, we will modify it's age and then try to find it again in the *HashMap*:

```
mary.setAge(mary.getAge() + 1);
Integer againMarysNumber = telephoneDictionary.get(mary);
// againMarysNumber = null Oups!
```

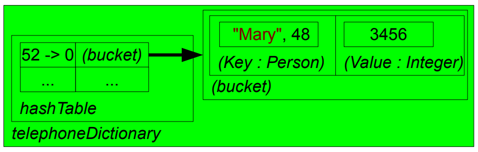
  - But it doesn't work this time! – We can't find the *Integer* associated to *mary*! What went wrong?

Be aware of mutable Key-Types! – Part II

- In the moment we add *mary* to the *HashMap*, its hash code will be requested and used to select a bucket to put *mary* in.
  - E.g. *Person{name="Mary", age=48}* would yield the hash code 52, which is associated to the bucket at index 0:

```
public class Person { // (members hidden)
        @Override
        public int hashCode() {
                return age + (null != name ? name.length() : 0);
        }
}
```

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
Person mary = new Person("Mary", 48);
telephoneDictionary.put(mary, 3456);
Integer marysNumber = telephoneDictionary.get(mary);
// marysNumber = 3456
```



- But when we modify *mary*'s age to 49, *mary* will yield another hash code, which is not associated to the bucket at index 0!
  - The effect is that *HashMap.get()* won't find a bucket for *mary*, because it is no longer equal to the version of *mary* that was put!

```
mary.setAge(mary.getAge() + 1);
Integer againMarysNumber = telephoneDictionary.get(mary);
// againMarysNumber = null Oups!
```

- What is the problem? The core problem is that we changed the structural equality of the key after it was added!
  - When we allow changing key-type that way, it will result in objects that can no longer be found in *HashMaps*!

- How can we fix this? The fix is simple, but maybe not attractive: Key-types should be immutable!

Be aware of mutable Key-Types! – Part III

- Let' make *Person* a key-type, that is really immutable.
  - The fields contributing to the equals contract (*hashCode()*/ *equals()*) are declared private final.
  - Those fields can only be initialized in ctors.
  - Those fields cannot be set, but getters can be provided.
    - The exposed fields should either be value-types or themselves be immutable types (like *String*) following these guidelines.
- Takeaway: key-types should be immutable!
  - There exist different ideas of immutability, but it should be clear, that fields contributing to the equals contract should be specifically guarded.
- (When key-types are immutable as discussed here, it could be an option to intern their hash codes.)

```
public class Person {
        private final String name; // Immutable fields
        private final int age;
        public Person(String name, int age) { // Only initialization
                this.name = name;
                this.age = age;
        }
        public String getName() { // Only getters, no setters
                return name; // Returns a String that is also immutable
        }
        public int getAge() {
                return age; // int is a value-type that is copied when returned
        }
        @Override
        public int hashCode() {
                return age + (null != name ? name.length() : 0);
        }
        @Override
        public boolean equals(Object other) {
                if (this == other) {
                        return true;
                }
                if (null != other && getClass() == other.getClass()) {
                        Person otherPerson = (Person)other;
                        return getAge() == otherPerson.getAge()
                                && Objects.equals(getName(), otherPerson.getName());
                }
                return false;
        }
}
```

33

More Details about Java's HashMap

- The actual index function used in *HashMap* is not just a modulus-operation:
  - Aspect 1: performance
    - (1) The size is always a power of 2. The initial capacity is 16, which is the nearest power of 2 to the number 10.
    - (2) The index in the hash table is: **index = hash-code % hash-table-size**. Or expressed with bitand: **index = hash-code & (hash-table-size - 1)**.
    - => Calculating the modulus with bitand is very fast when hash-table-size is a power of two!
  - Aspect 2: negative values
    - If a hash code is a negative number, hash-table-size is necessarily positive, modulus will result in a negative index! But we cannot have negative indexes.
    - => To avoid invalid indexes, *HashMap* confirms to calculating the index via bitand: **index = hash-code & (hash-table-size - 1)**.
  - Aspect 3: bad hashes: The implementation performs re-hashing operations on the keys' hash-code to give them a wider spread.

- If the key-type implements *Comparable* and a bucket grows over 8 items buckets are converted from linked lists to BSTs!
  - This means, that for more than 8 keys a binary search will be done in a bucket, which has only O(logn) complexity!
  - Mind that with the linked list and iteration and *equals()*-based comparison it would be a linear search of O(n) complexity.

- *HashMap*'s buckets are internally called "bins".

34

Java: Equality-based Comparison apart from HashMap

- Let me loose some words, why the equality contract is very important in Java.
  - Put simple: if our types implement the equality contract, a lot of mighty functionality is enabled.

- Many methods regarding finding certain elements in collections require implementing the equals contract.
  - There are even voices, who consider types without implementing the equals contract cannot be used in collections.
  - Some Java-algorithms are "based" on items' *equals()*, but this virtually means that the equality contract must be fully implemented.
  - Notice, the equals contract is not sufficient to implement sorting, because it doesn't define a relative order of elements!

- Actually, there exist some methods in collections, which look into the contained elements, esp. to do some comparison.
  - Remember: usually methods of non-associative collections do not call methods on contained elements following ones are exceptions:
    - *Collection.contains()*, *Collection.containsAll()*, *Collection.remove()*, *Collection.removeAll()* and *Collection.retainAll()*. *List.indexOf()* and *List.lastIndexOf()*.
  - Contrary to associative collections, which need to call methods on contained elements/keys for their internal management.

- *Stream.distinct()* does its processing also based on the equality contract.

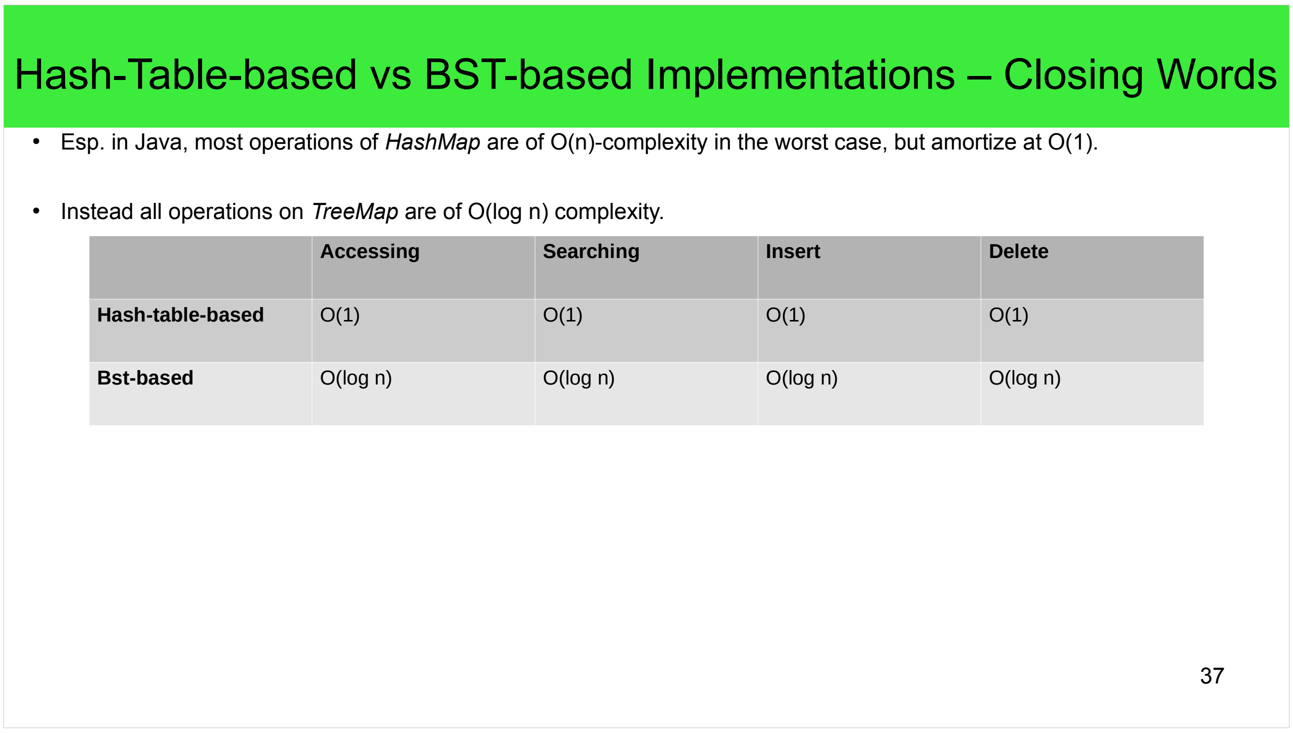
Looking up Values

- Keep in mind, that associative collections have only requirements on the key-type.
  - In BST-based associative collections like *TreeMap*s, key-types must implement *Comparable* or *Comparator*s must be provided.
  - In hash-table-based associative collections like *HashMap*s, key-types must implement the equals contract.

- But associative collections also provide some methods, which require the value-type implementing the equals contract.
  - There is for example *Map.containsValue()*. – You got it, in associative collections we can also lookup values! Let's try it:

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
telephoneDictionary.put(liam, 7764);
telephoneDictionary.put(natalie, 3821);

// Ok, and now let's lookup a phone number:
boolean containsNataliesNumber = telephoneDictionary.containsValue(3821);
// containsNataliesNumber = true
```

  - Yes, we can lookup values, but this operation is not that efficient. All implementations will probably do a linear search over all values.
    - Btw. looking up a value in an associative collections is known as "inverse search", because associative collections are rather there to lookup keys.
  - This linear search would equality-compare the values, so it makes sense that the value-types implement the equals contract as well.
  - (Adhering to the equals contract can be a low-hanging-fruit, but for *Map.containsValue()* identity-comparison might also be useful.)



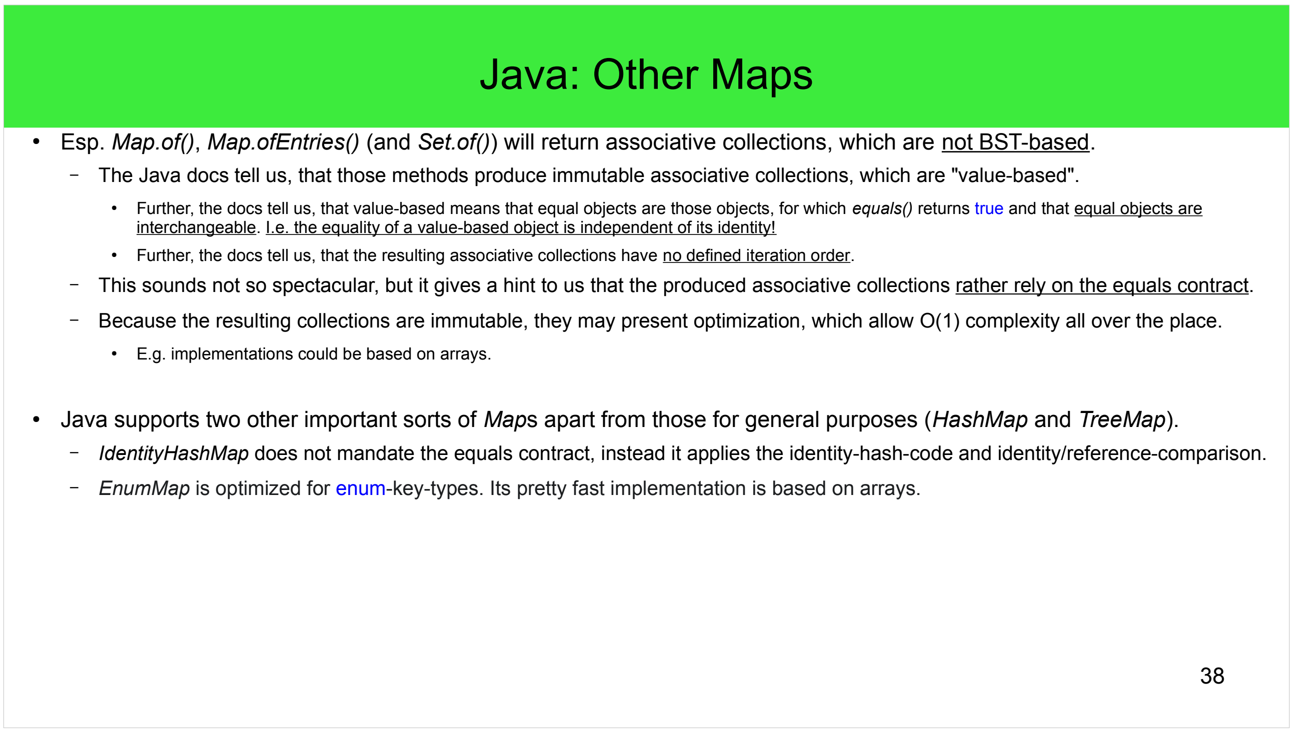
Hash-Table-based vs BST-based Implementations – Closing Words

- Esp. in Java, most operations of *HashMap* are of O(n)-complexity in the worst case, but amortize at O(1).
- Instead all operations on *TreeMap* are of O(log n) complexity.

	Accessing	**Searching**	**Insert**	**Delete**
Hash-table-based	O(1)	O(1)	O(1)	O(1)
Bst-based	O(log n)	O(log n)	O(log n)	O(log n)

37

- These statements also hold true for *HashSet* and *TreeSet* respectively.



Java: Other Maps

- Esp. *Map.of()*, *Map.ofEntries()* (and *Set.of()*) will return associative collections, which are not BST-based.
  - The Java docs tell us, that those methods produce immutable associative collections, which are "value-based".
    - Further, the docs tell us, that value-based means that equal objects are those objects, for which *equals()* returns true and that equal objects are interchangeable. I.e. the equality of a value-based object is independent of its identity!
    - Further, the docs tell us, that the resulting associative collections have no defined iteration order.
  - This sounds not so spectacular, but it gives a hint to us that the produced associative collections rather rely on the equals contract.
  - Because the resulting collections are immutable, they may present optimization, which allow O(1) complexity all over the place.
    - E.g. implementations could be based on arrays.

- Java supports two other important sorts of *Map*s apart from those for general purposes (*HashMap* and *TreeMap*).
  - *IdentityHashMap* does not mandate the equals contract, instead it applies the identity-hash-code and identity/reference-comparison.
  - *EnumMap* is optimized for enum-key-types. Its pretty fast implementation is based on arrays.

38

- These statements also hold true for *Set.of()* and *Set.copyOf()* respectively.

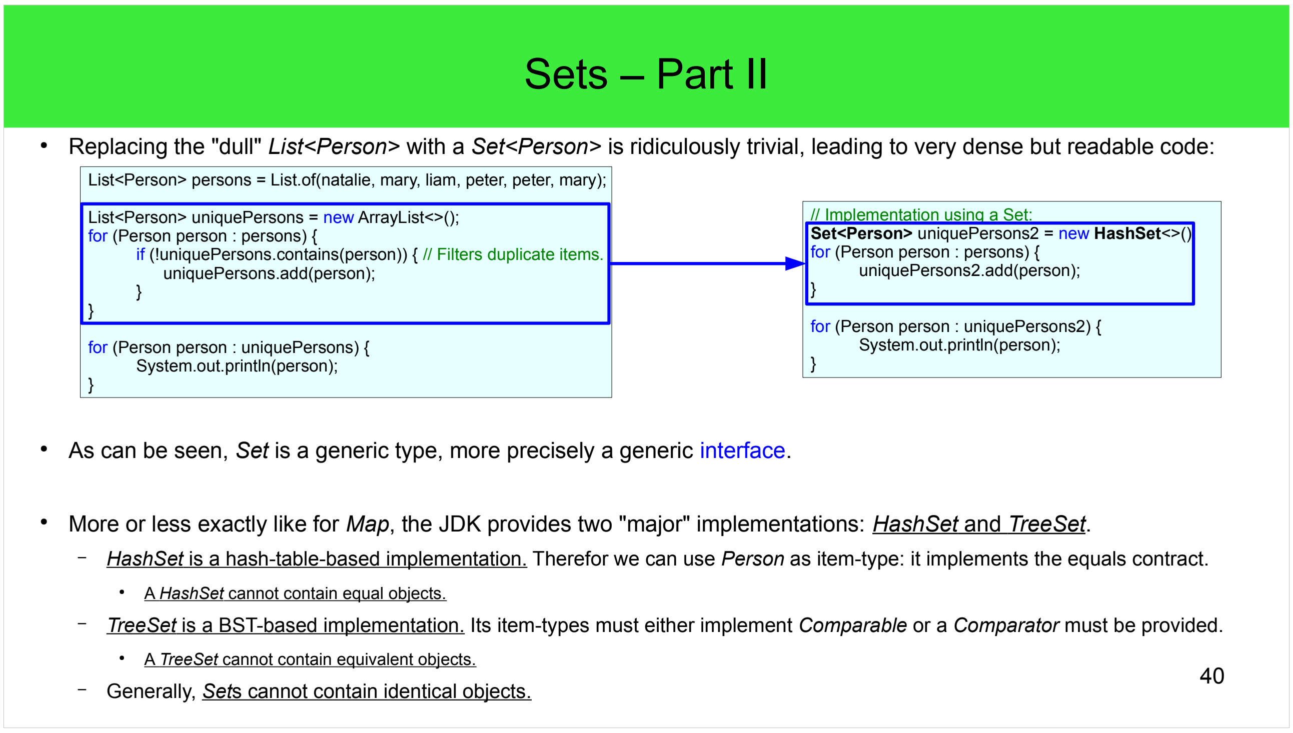
Sets – Part I

- Very often, we have to care for collections of data, which only contain unique items.
  - E.g. collecting each *Person* only once:

```
List<Person> persons = List.of(natalie, mary, liam, peter, peter, mary);

List<Person> uniquePersons = new ArrayList<>();
for (Person person : persons) {
        if (!uniquePersons.contains(person)) { // Filters duplicate items.
            uniquePersons.add(person);
        }
}

for (Person person : uniquePersons) {
        System.out.println(person);
}
// > name = Natalie, age = 45
// > name = Mary, age = 48
// > name = Liam, age = 37
// > name = Peter, age = 36
```

- This can be done simpler, esp. with less code using a specific type of collection: a *Set*.

- *Set*s reject adding items, which are already contained. This is only possible, because *Set*s look into the contained items.
  - We already know, that associative collections like *Map*s also have this specific feature in common: they inspect the contained keys.
  - *Map*s inspect the keys to (quickly) find associated values.
  - *Set*s do the same, but in a *Set* the key is the value! A *Set* associates a key with itself, therefor it is also an associative collection!



Sets – Part II

- Replacing the "dull" *List<Person>* with a *Set<Person>* is ridiculously trivial, leading to very dense but readable code:

```
List<Person> persons = List.of(natalie, mary, liam, peter, peter, mary);

List<Person> uniquePersons = new ArrayList<>();
for (Person person : persons) {
        if (!uniquePersons.contains(person)) { // Filters duplicate items.
            uniquePersons.add(person);
        }
}

for (Person person : uniquePersons) {
        System.out.println(person);
}
```

```
// Implementation using a Set:
Set<Person> uniquePersons2 = new HashSet<>()
for (Person person : persons) {
        uniquePersons2.add(person);
}

for (Person person : uniquePersons2) {
        System.out.println(person);
}
```

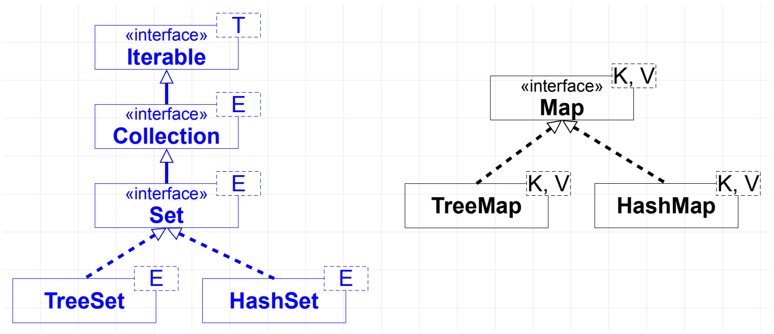
- As can be seen, *Set* is a generic type, more precisely a generic interface.

- More or less exactly like for *Map*, the JDK provides two "major" implementations: *HashSet* and *TreeSet*.
  - *HashSet* is a hash-table-based implementation. Therefor we can use *Person* as item-type: it implements the equals contract.
    - A *HashSet* cannot contain equal objects.
  - *TreeSet* is a BST-based implementation. Its item-types must either implement *Comparable* or a *Comparator* must be provided.
    - A *TreeSet* cannot contain equivalent objects.
  - Generally, *Set*s cannot contain identical objects.

40

- As for *Map*s (and all associative collections) the cause using *Set*s is to exploit its self-organization (keeping items unique), instead of organizing it ourselves. But for *Set*s the key-type is the item-type!

Sets – Part III



- In opposite to *Map*, *Set* implements *Collection*, i.e. a *Set* is a *Collection*.
  - This is useful, it means that *Sets* can be iterated with for(:) (it implements *Iterable*) and can provide a *Stream* via *Collection.stream()*:

```
// Sets can be iterated via for(:) featuring Iterable:
for (Person person : uniquePersons2) {
        System.out.println(person);
}
```

```
// Sets can directly provide a Stream, offering a plethora of mighty functionality:
uniquePersons2
        .stream()
        .filter(person -> person.getAge() > 40)
        .forEach( person -> System.out.println(person));
```

  - Remarkable fact: *Set* "refines" the method *Collection.add()*: *Set.add()* can reject to add items!

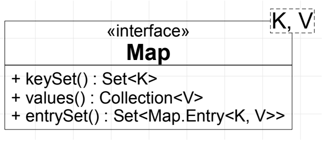
41

Connections between Maps and Sets in Java

- Actually, *Set*s are implemented in terms of *Map*s in Java:
  - *TreeSet*s are backed by BST-based *Map*s (e.g. *NavigateMap*).
  - *HashSet*s are backed by hash-table-based *Map*s (e.g. just a *HashMap*).
  - The backing *Map*s just use a single dummy-object as value for every association. – The values are not of interest for the *Set*s!

- You may remember, that *Map*s provide *Set*s of their keys via *Map.keySet()* or a *Set* of the entries via *Map.entrySet()*.
  - An important point here is, that *Map.keySet()* and *Map.entrySet()* only provide views of the underlying *Map*.
  - It means, that the *Set* returned by *Map.keySet()* reflects the changes done to the original *Map*:

```
Map<Person, Integer> telephoneDictionary = new HashMap<>();
telephoneDictionary.put(natalie, 1234);
telephoneDictionary.put(peter, 2345);
telephoneDictionary.put(mary, 3456);
telephoneDictionary.put(liam, 4567);
Set<Person> persons = telephoneDictionary.keySet();
System.out.printf("# of contained persons: %d%n", persons.size());
// > # of contained persons: 4
telephoneDictionary.remove(peter);
System.out.printf("New # of contained persons: %d%n", persons.size());
// > New # of contained persons: 3
```


K, V
«interface»
Map
+ keySet() : Set<K>
+ values() : Collection<V>
+ entrySet() : Set<Map.Entry<K, V>>

  - Also *Map.values()* only provides a view (of type *Collection<V>*) to the original *Map* and may change.
  - The results of *Map.keySet()*, *Map.entrySet()* and *Map.values()* are only views and should not be handled as durable data.

Simple Factories for Sets

- Like *Map*, *Set* provides the simple factory *Set.of()*, which creates an immutable *Set* from an arbitrary count of arguments:
  - The arguments of *Set.of()* must not be null and they must be distinct!

```
Set<Person> persons = Set.of(natalie, peter, mary, liam);
```

```
Set<Person> emptySet = Set.of();
```

- Another useful simple factory is *Set.copyOf()*. It creates an immutable *Set* from the passed *Collection*.
  - We can use *Set.copyOf()* to simplify our introductory *Set*-example:

```
List<Person> persons = List.of(natalie, mary, liam, peter, peter, mary);
Set<Person> uniquePersons = new HashSet<>();
for (Person person : persons) {
        uniquePersons.add(person);
}

for (Person person : uniquePersons) {
        System.out.println(person);
}
```

```
List<Person> persons2 = List.of(natalie, mary, liam, peter, peter, mary);
Set<Person> uniquePersons2 = Set.copyOf(persons2);

for (Person person : uniquePersons2) {
        System.out.println(person);
}
```

  - The passed *Collection* must not contain nulls.
  - If the *Collection* has duplicates, it is undefined, which unique item will be present in the resulting *Set*.
  - The created *Set* bases on a copy of the passed *Collection*: changes on the original *Collection* won't be reflected in the *Set* afterwards.
  - The type of returned *Set* is not specified, but it requires the *Collection*'s items implementing the equals contract.

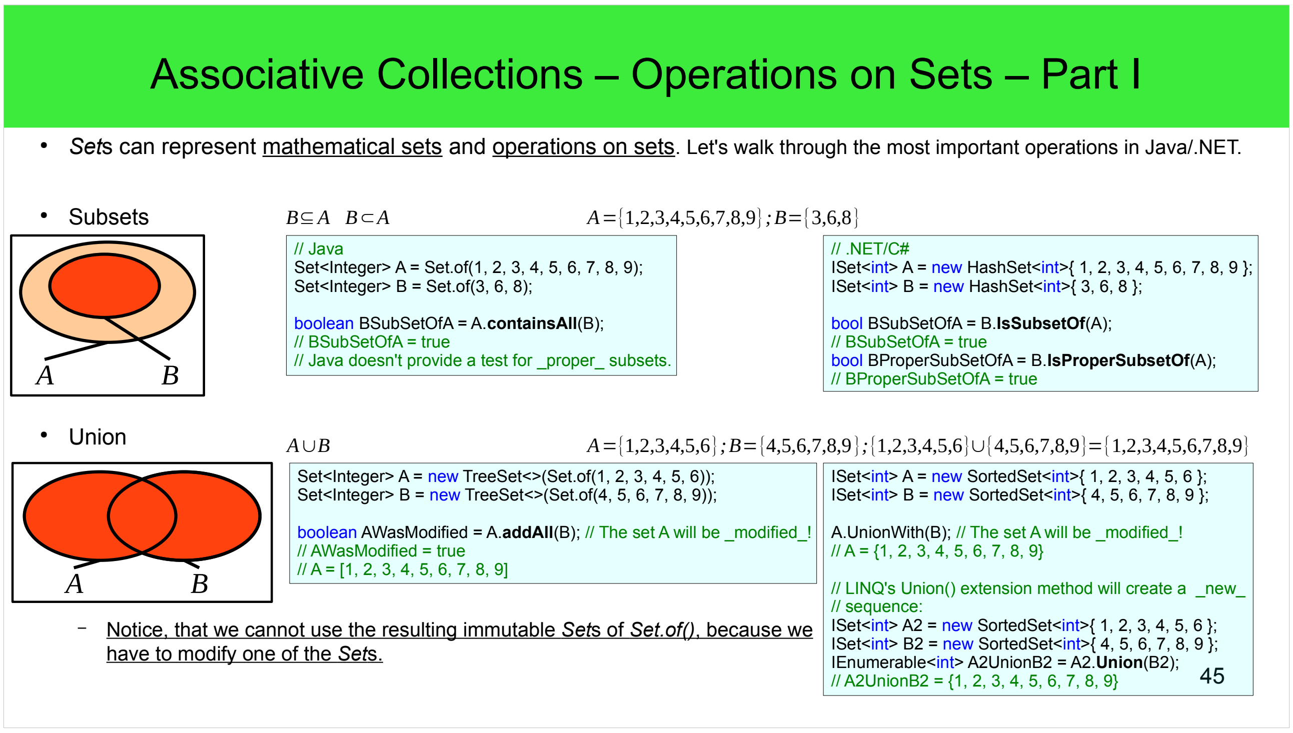
Sets in other Languages

- In .NET all sets implement the generic interface *ISet*.
  - The BST-based *ISet* is implemented in the class *SortedSet*. Key-types must implement *IComparable* or *Comparers* must be provided.
  - The hash-table-based *ISet* in implemented in the class *HashSet*. Key-types must implement .NET's equals contract.

```
// Implementation using an ISet/HashSet:
ISet<Person> uniquePersons = new HashSet<Person>();
foreach (Person person in persons) {
	uniquePersons.Add(person);
}

foreach (Person person in uniquePersons2) {
	Console.WriteLine(person);
}
```

- In C++ sets are implemented as template classes of the STL.
  - The BST-based set is implemented in *std::set* (<set>). The equivalence of types is expressed by overloading operator<.
  - The hash-table-based set is implemented in *std::unordered_set* (<unordered_set>). Equality is check via hashers and key-equality.

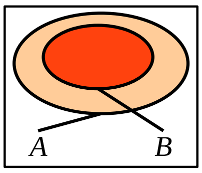
```
// Implementation using an std::set:
#include <set>
std::set<Person> uniquePersons;
for (Person person : persons) {
	uniquePersons.insert(person);
}

for (Person person : uniquePersons) {
	std::cout<<person<<std::endl;
}
```



Associative Collections – Operations on Sets – Part I

- *Set*s can represent <u>mathematical sets</u> and <u>operations on sets</u>. Let's walk through the most important operations in Java/.NET.

- Subsets

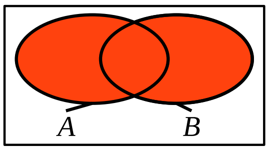


$B \subseteq A \quad B \subset A$ $\qquad A=\{1,2,3,4,5,6,7,8,9\}; B=\{3,6,8\}$

```
// Java
Set<Integer> A = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
Set<Integer> B = Set.of(3, 6, 8);

boolean BSubSetOfA = A.containsAll(B);
// BSubSetOfA = true
// Java doesn't provide a test for _proper_ subsets.
```

```
// .NET/C#
ISet<int> A = new HashSet<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
ISet<int> B = new HashSet<int>{ 3, 6, 8 };

bool BSubSetOfA = B.IsSubsetOf(A);
// BSubSetOfA = true
bool BProperSubSetOfA = B.IsProperSubsetOf(A);
// BProperSubSetOfA = true
```

- Union



$A \cup B$ $\qquad A=\{1,2,3,4,5,6\}; B=\{4,5,6,7,8,9\}; \{1,2,3,4,5,6\} \cup \{4,5,6,7,8,9\} = \{1,2,3,4,5,6,7,8,9\}$

```
Set<Integer> A = new TreeSet<>(Set.of(1, 2, 3, 4, 5, 6));
Set<Integer> B = new TreeSet<>(Set.of(4, 5, 6, 7, 8, 9));

boolean AWasModified = A.addAll(B); // The set A will be _modified_!
// AWasModified = true
// A = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

  - <u>Notice, that we cannot use the resulting immutable *Set*s of *Set.of()*, because we have to modify one of the *Set*s.</u>

```
ISet<int> A = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };

A.UnionWith(B); // The set A will be _modified_!
// A = {1, 2, 3, 4, 5, 6, 7, 8, 9}

// LINQ's Union() extension method will create a  _new_
// sequence:
ISet<int> A2 = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B2 = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };
IEnumerable<int> A2UnionB2 = A2.Union(B2);
// A2UnionB2 = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

45

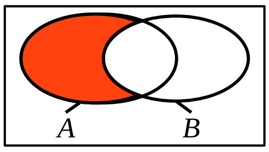
- The shown diagrams are called Venn-diagrams. It is a widely-used because it nicely shows relations between sets. It was at least popularized by John Venn, a British mathematician in the 1880s.
- A proper subset means, that a set is a subset of another set, but both sets are <u>not</u> equal!
- Subsets can also be expressed with LINQ, but usually the type *ISet* and its implementors should be used, because it leads to more expressive code:

```
// C#/.NET/LINQ
ISet<int> A = new HashSet<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
ISet<int> B = new HashSet<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
ISet<int> C = new HashSet<int>{ 3, 6, 8 };

// Subset:
bool BIsSubsetOfA = !B.Except(A).Any();
// BIsSubsetOfA = true
bool CIsSubsetOfA = !C.Except(A).Any();
// CIsSubsetOfA = true

// Proper subset:
bool BIsProperSubSetOfA = A.Except(B).Any();
// BIsProperSubSetOfA = false
bool CIsProperSubSetOfA = A.Except(C).Any();
// CIsProperSubSetOfA = true
```

Associative Collections – Operations on Sets – Part II

- Difference $A \setminus B (:= B^C \cap A)$ $A = \{1,2,3,4,5,6\}; B = \{4,5,6,7,8,9\}; \{1,2,3,4,5,6\} \setminus \{4,5,6,7,8,9\} = \{1,2,3\}$

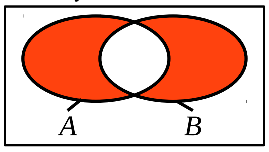
  (relative complement)



```
// Java
Set<Integer> A = new TreeSet<>(Set.of(1, 2, 3, 4, 5, 6));
Set<Integer> B = new TreeSet<>(Set.of(4, 5, 6, 7, 8, 9));

A.removeAll(B); // The set A will be _modified_!
// A = [1, 2, 3]
```

```
// .NET/C#
ISet<int> A = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };

A.ExceptWith(B); // The set A will be _modified_!
// A = {1, 2, 3}

// LINQ's Except() method will create a _new_ sequence:
ISet<int> A2 = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B2 = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };
IEnumerable<int> A2ExceptB2 = A2.Except(B2);
// A2ExceptB2 = {1, 2, 3}
```

- Symmetric difference $A \triangle B (:= (A \setminus B) \cup (B \setminus A))$ $A = \{1,2,3,4,5,6\}; B = \{4,5,6,7,8,9\}; \{1,2,3,4,5,6\} \triangle \{4,5,6,7,8,9\} = \{1,2,3,7,8,9\}$



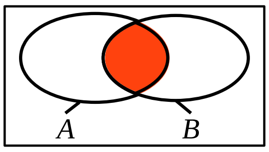
```
Set<Integer> A = new TreeSet<>(Set.of(1, 2, 3, 4, 5, 6));
Set<Integer> B = new TreeSet<>(Set.of(4, 5, 6, 7, 8, 9));
Set<Integer> A2 = new TreeSet<>(Set.of(1, 2, 3, 4, 5, 6));

A.removeAll(B); // The sets A and B will be _modified_!
B.removeAll(A2);
A.addAll(B);
// A = [1, 2, 3, 7, 8, 9]
```

```
ISet<int> A = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };

A.SymmetricExceptWith(B); // The set A will be _modified_!
// >{1, 2, 3, 7, 8, 9}

// Using LINQ we can create a _new_ sequence:
ISet<int> A2 = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B2 = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };
IEnumerable<int> A2SymmetricExceptB2 =
        A2.Except(B2).Union(B2.Except(A2));
// A2SymmetricExceptB2 = {1, 2, 3, 7, 8, 9}
```

Associative Collections – Operations on Sets – Part III

- Intersection

$A \cap B$ $A=\{1,2,3,4,5,6\}; B=\{4,5,6,7,8,9\}; \{1,2,3,4,5,6\} \cap \{4,5,6,7,8,9\} = \{4,5,6\}$



```
// Java
Set<Integer> A = new TreeSet<>(Set.of(1, 2, 3, 4, 5, 6));
Set<Integer> B = new TreeSet<>(Set.of(4, 5, 6, 7, 8, 9));

A.retainAll(B); // The set A will be _modified_!
// A = [4, 5, 6]
```

```
// .NET/C#
ISet<int> A = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };

A.IntersectWith(B); // The set A will be _modified_!
// A = {4, 5, 6}

// LINQ's Intersect() method will create a _new_ sequence:
ISet<int> A2 = new SortedSet<int>{ 1, 2, 3, 4, 5, 6 };
ISet<int> B2 = new SortedSet<int>{ 4, 5, 6, 7, 8, 9 };
IEnumerable<int> A2IntersectionWithB2 = A2.Intersect(B2);
// A2IntersectionWithB2 = {4, 5, 6}
```

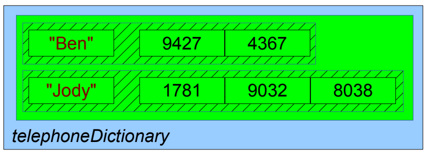
47

Special associative Collections – Multiplicity of Keys

- Inserting values having already present keys in an associative collection overwrites or updates present values.
  - Sometimes this is not desired. E.g. mind a *telephoneDictionary*, in which a name can have more than one phone number!

- Some collection frameworks provide associative collections that can handle multiplicity.
  - In C++ we can use *std::multimap* (<map>) and *std::multiset* (<set>).

```
// C++11
std::multimap<std::string, int> telephoneDictionary {
        { "Ben", 9427 }, // Mind: two values with the same key are added here.
        { "Ben", 4367 },
        { "Jody", 1781 }, // Mind: three values with the same key are added here.
        { "Jody", 9032 },
        { "Jody", 8038 }
};

// It is required to use STL iterators, because std::multimap provides no subscript operator:
for (auto item = telephoneDictionary.begin(); item != telephoneDictionary.end(); ++item) {
        std::cout<<"Name: "<<item->first<<", Phone number: "<<item->second<<std::endl;
}
```



- In other frameworks (Java, .NET etc.) multi-associative collections need to be explicitly implemented or taken from 3rd party.
  - An own implementation could use a key-type *T*, but a *List<V>* as value-type.
  - 3rd party sources: Apache Commons and Spring (Java)

Things to ponder about Associative Collections – Part I

- Key
  - Refrain from modifying key objects managed in associative collections.

- BST-based/sorted/equivalence-based associative collections:
  - *TreeMap*/*TreeSet* (Java), *SortedDictionary*/*SortedSet* (.NET), *std::set*/*std::map* (C++),
  - The internal organization of keys is equivalence-based:
    - Equivalence-based means that key objects are organized due to their relative/natural order. This is often implemented with a BST.
    - The key-type needs implementing *Comparable* (Java) or *IComparable* (.NET) or operator< (C++ STL) for semantically correct "natural order".
    - Or a *Comparator* (Java) or *IComparer* (.NET) or a comparison functor (C++ STL) needs to be specified for the associative collection that implements "natural ordering" for the key-type.
  - Searching/inserting/removing items can be done very fast (O(logn)), because binary searches are used (BST).
  - Sorted associative collections are unordered!
    - The iteration will yield the items in sorted order (BST in-order). The order, which items have been added/inserted/removed doesn't matter!
  - => Use these associative collections, if sorting of keys or the control of key-comparison (e.g. reverse order) is needed!

49

Things to ponder about Associative Collections – Part II

- Hash-table-based/equality-based associative collections:
  - *HashMap*/*HashSet* (Java), *Dictionary*/*HashSet* (.NET), *NSDictionary*/*NSSet* (Cocoa), *std::unordered_map*/*std::unordered_set* (C++), associative arrays (JavaScript)
  - The internal organization of keys is equality-based:
    - By equality. I.e. by hash codes and the result of the equality check (methods *hashCode()/equals()* (Java) or *GetHashCode()/Equals()* (.NET)).
    - The key-type needs implementing *hashCode()/equals()* (Java) or *GetHashCode()/Equals()* (.NET) for correct equality.
    - Or an *IHashCodeProvider/IEqualityComparer* (.NET) needs to be specified for the associative collection that implements equality for the key-type.
    - In C++ STL a hasher functor should be specified for the associative collection that provides hash codes for the key-type.
  - Searching/inserting/removing items can be done O(1) no extra search operation is required, the hash code is used as an index.
  - These associative collections are unordered!
    - The iteration order makes no guarantees about how items are yielded. It can change completely when items are added.

- In most cases <u>equality-based associative collection should be our first choice</u>, those are potentially most efficient.
  - <u>=> Use these associative collections, if sorting of keys or the control of key-comparison doesn't matter!</u>

- Ordered associative collections:
  - *LinkedHashMap* (Java) and *OrderedDictionary* (.NET) iterate in the order, in which items were put into the collection (insertion order).
    - *LinkedHashMap* is the type Groovy uses to implement map literals.

Things to ponder about Associative Collections – Part III

- Old and new collections:
  - Java:
    - The old *Hashtable* is not null-aware on keys, adding null will throw an NPE. Better use *HashMap*/*HashSet* (Java 1.2 or newer).
    - *Hashtable* and *HashMap* return null, if a requested key has no value (i.e. key is not present).
    - Instead of *HashMap*, *Hashtable* is synchronized, which degrades performance (similar to *Vector*, which is a synchronized form of *ArrayList*).
  - .NET:
    - The old object-based *Hashtable* should not be used for new code using .NET 1 or newer. Better use *Dictionary*/*SortedDictionary* (.NET 2 or newer).
    - Be aware that *Hashtable* returns null, if a key has no value (i.e. key is not present). *Dictionary*/*SortedDictionary* will throw a *KeyNotFoundException*.

- Keys with the value null:
  - Java:
    - *TreeMap*/*TreeSet* don't allow keys having the value null, then an NPE will be thrown.
    - *HashMap*/*HashSet* can digest keys having the value null.
  - .NET:
    - *Dictionary*/*SortedDictionary* don't allow keys having the value null, then an *ArgumentNullException* will be thrown.
    - *HashSet*/*SortedSet* can digest keys having the value null.

Thank you!