UNIVERSITETI® METROPOLITAN TIRANA

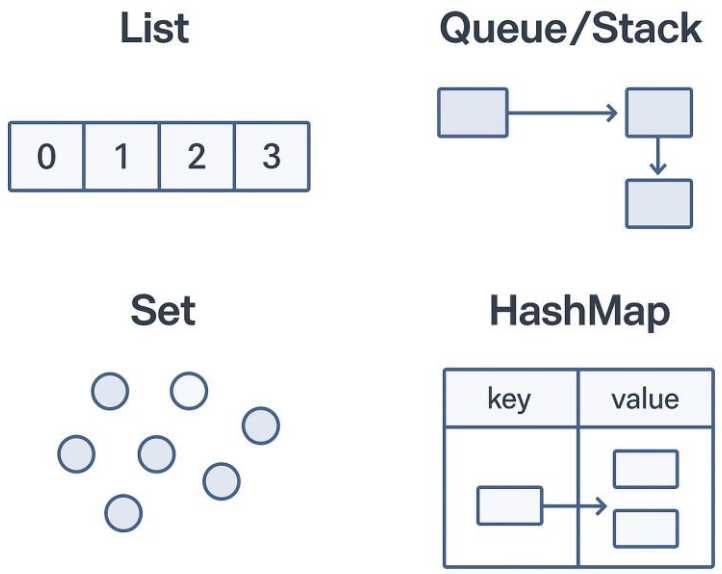
Course: Object Oriented Programming

Organizing Data
Understanding Collections in Java

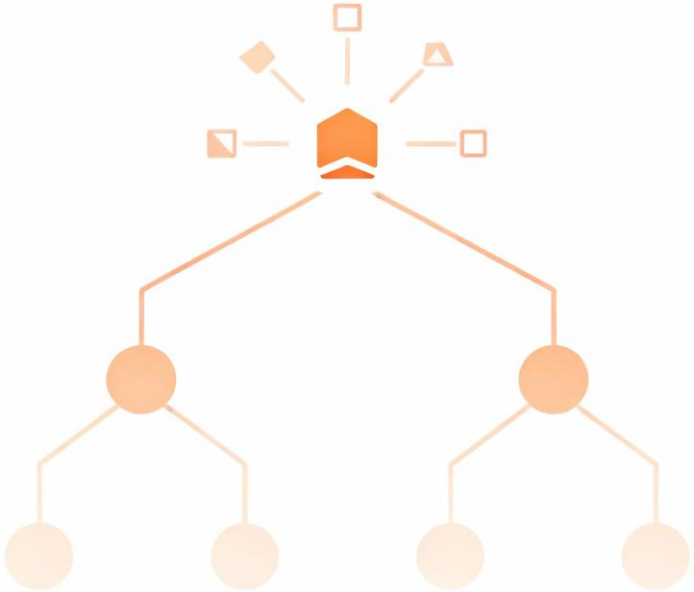
Evis Plaku

Various ways to organize data: each solves different problems

- Lists: order matters
- Sets: Uniqueness matters
- Maps: store pairs where each key instantly finds its value
- Queues & Stacks: process items in order



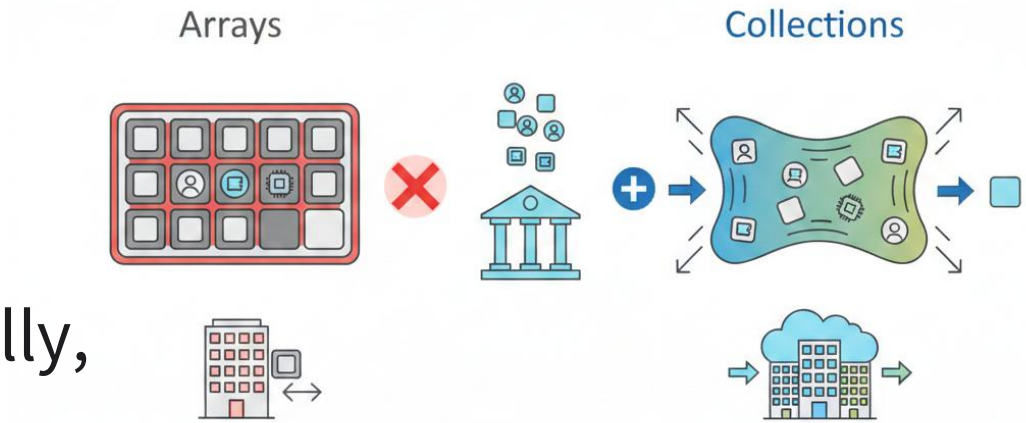
Today: Focus on core concepts and Maps. Lists covered earlier; others later.



Core Concepts
Collections

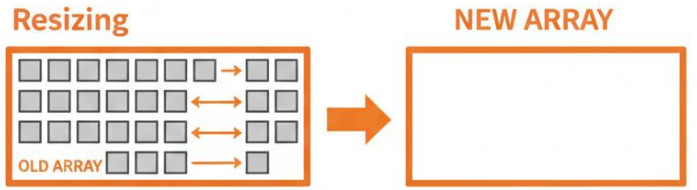
Arrays work, but collections solve real problems at scale

- Arrays are rigid: fixed size, no built-in add/remove, grows messy fast
- Collections are flexible: resize automatically, rich methods, *type-safe* with generics
- Bank scenario: thousands of accounts need fast lookup, not just storage

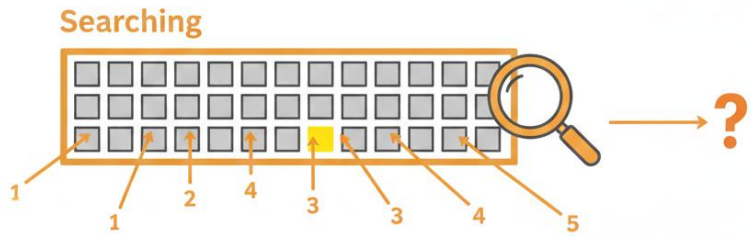


Manual array management slow, error-prone, and scales terribly

- Resizing: need more space? Create new array, copy everything manually



- Searching: loop through entire array looking for one item; slow with thousands



- Lookups: No way to ask “find account by ID”; must iterate every time

Four properties determine which collection fits your specific need

- Order: does order matter?
- Duplicates: are duplicates allowed?
- Access speed: some find instantly; others search slowly
- Memory trade-off: speed requires space

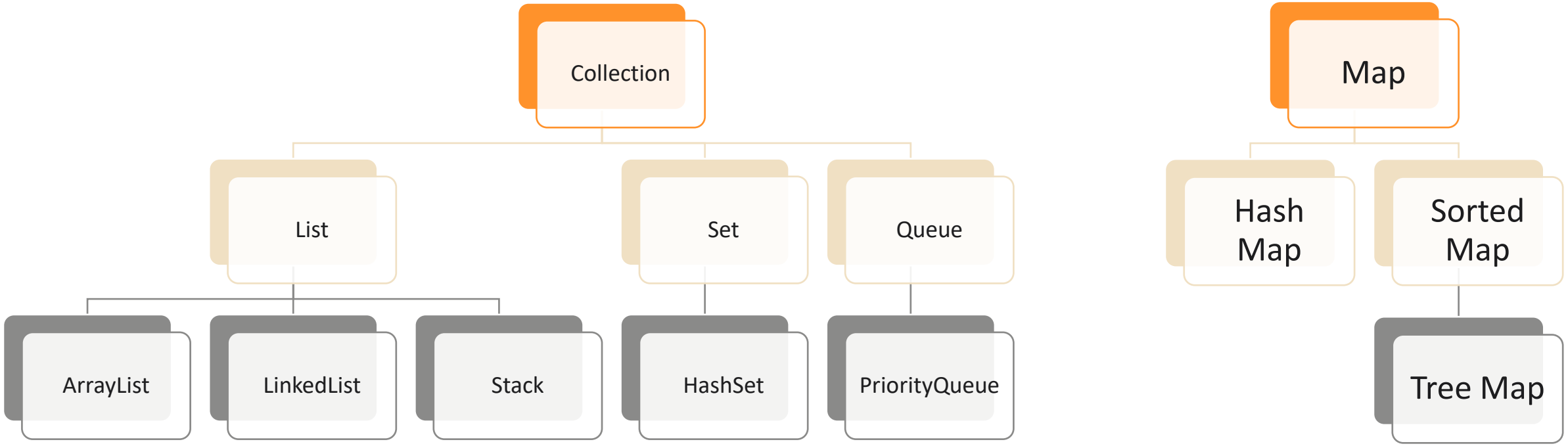
Choose

based on

what matters

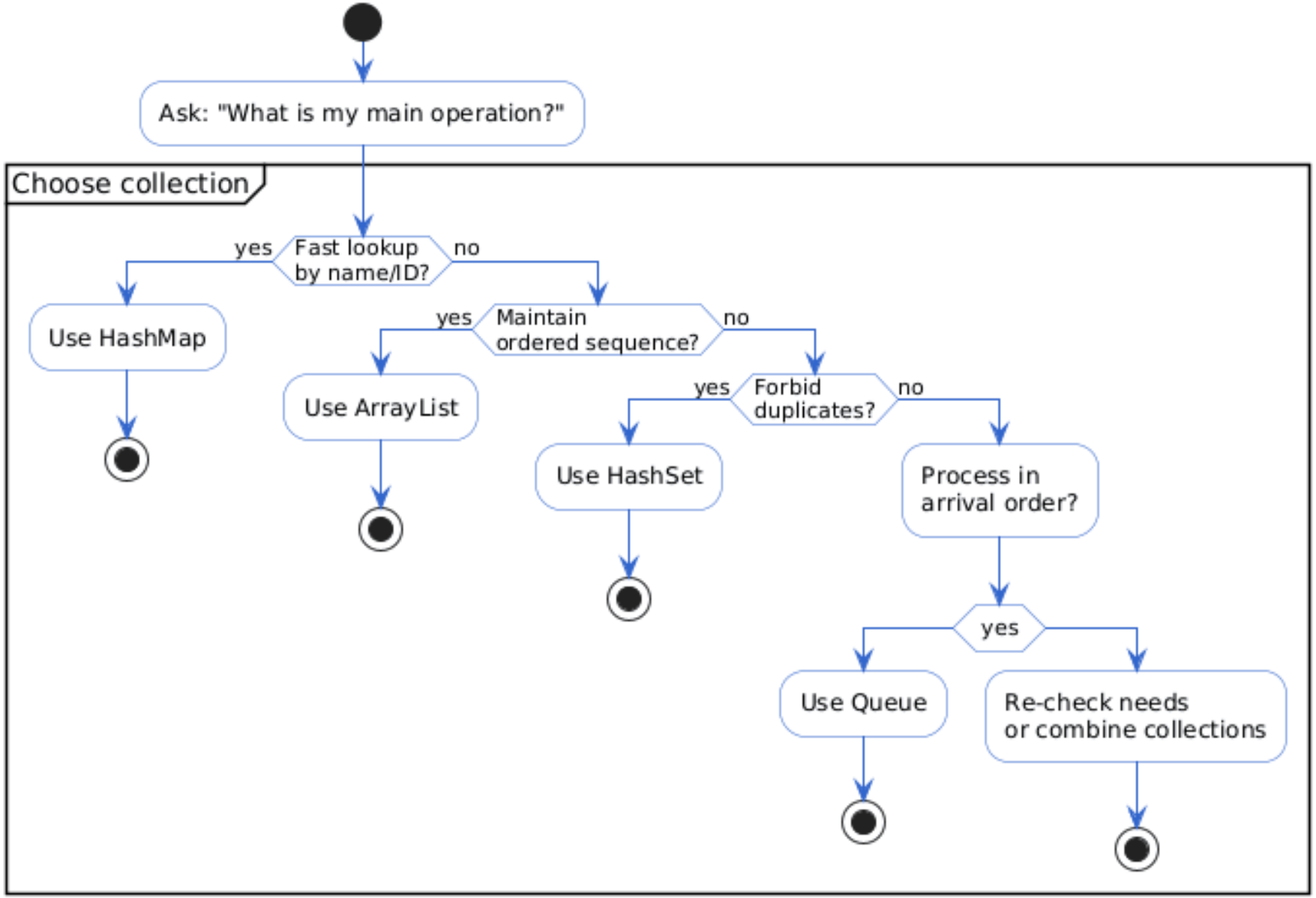
UNIVERSITETI METROPOLITAN TIRANA

One root interface, several major branches, each solving different problems



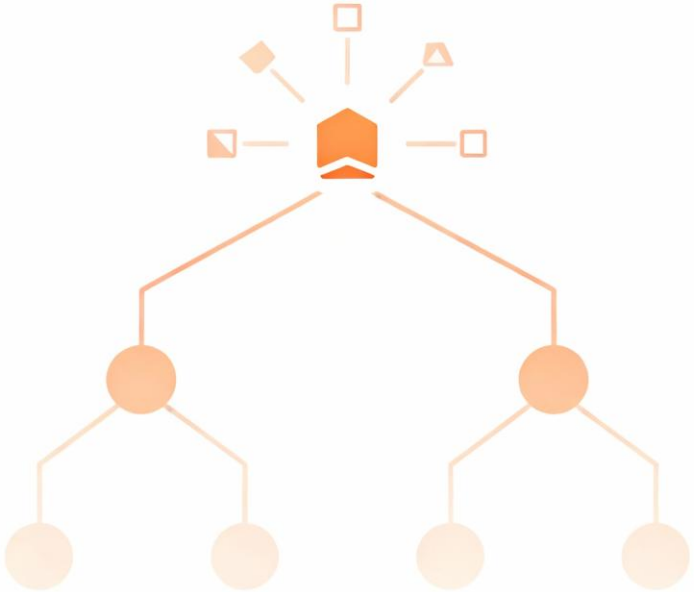
Note: the list is non-exhaustive

CHOOSING THE RIGHT ONE



Match your problem to the collection.

It's a simple decision tree



Core Functionalities
Collections

Basic operations: add, remove, check size all collections enable these

```java
Collection<Customer> customers = new ArrayList<>();

Customer c = new Customer(1, "Alan", "alan@turing.com", 41, "London")

System.out.println("Size: " + customers.size());          // 1
System.out.println("Empty? " + customers.isEmpty());     // false

customers.remove(c);
System.out.println("Size: " + customers.size());         // 0
```

- add(element): Insert one element. Returns true if successful, false if rejected
- remove(element): delete first occurrence. Returns true if found, false if not
- size() & isEmpty(): check how many or if empty; fast snapshot of collection

Ask questions: Is this element here? How fast depends on the collection

- contains(element): check if element exists.
  Returns true/false. Speed varies by collection

- HashMap: instant;
  ArrayList: must check every element

- Real scenario: bank needs to check if customer ID exists before processing

```
Customer c = new Customer(
  1, "Alan", "alan@turing.com", 41, "London")

if (customers.contains(c))
    System.out.println("Customer found!");
else
    System.out.println("Customer not found.");
```

Loop through collections safely: enhanced for-loop works with any collection

- Enhanced for-loop: for(Customer c : customers) works with any collection type
- Iterator pattern: More control. Use when modifying during iteration is needed
- Avoid modification trap: never add/remove during loop unless using iterator explicitly

```
Iterator<Customer> iter = customers.iterator();
while (iter.hasNext()) {
    Customer c = iter.next();
    if (c.getAge() < 26) {
        iter.remove();  // Safe removal
    }
}
```

Merge, filter, and combine collections efficiently with bulk operations

- addAll(other): merge two collections.
  All elements from other added to this

- retainAll(other): keep only common elements.
  Remove anything not in other

- removeAll(other): delete elements. Remove anything that appears in other collection

```
Collection<Customer> bankA = new ArrayList<>();

Collection<Customer> bankB = new ArrayList<>();


// Merge all customers from Bank B into Bank A
bankA.addAll(bankB);

// Filter: keep only customers over 28 years old
Collection<Customer> over28 = new ArrayList<>();

bankA.retainAll(over28);
```

Define how objects compare: implement Comparable to enable automatic sorting

- Comparable interface: define natural order with compareTo() method
- How it works: returns negative (this < other), zero (equal), positive (this > other)

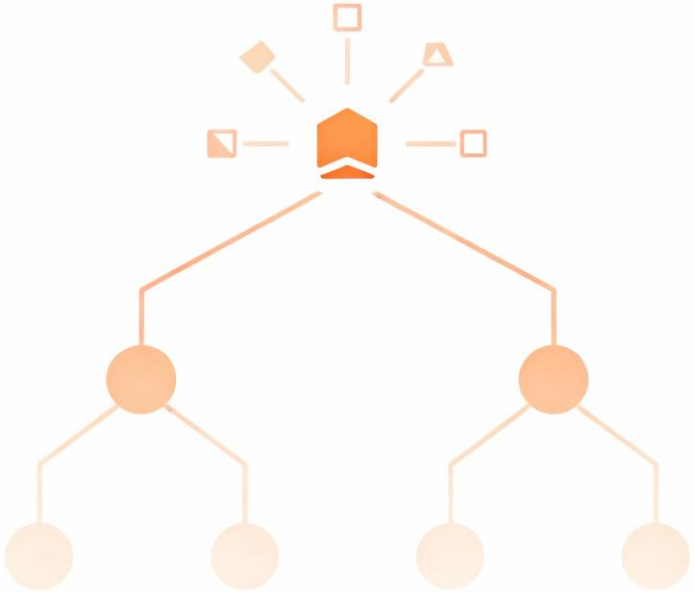
```java
public class Customer implements Comparable<Customer> {

    // Properties, constructor, getters, setters omitted...

    @Override
    public int compareTo(Customer other) {
        // Sort by age (ascending)
        return Integer.compare(this.age, other.age);

        // Or sort by name (alphabetical):
        // return this.name.compareTo(other.name);
    }
}
```

- Real scenario: sort customers by age, name, or any field using Collections.sort()



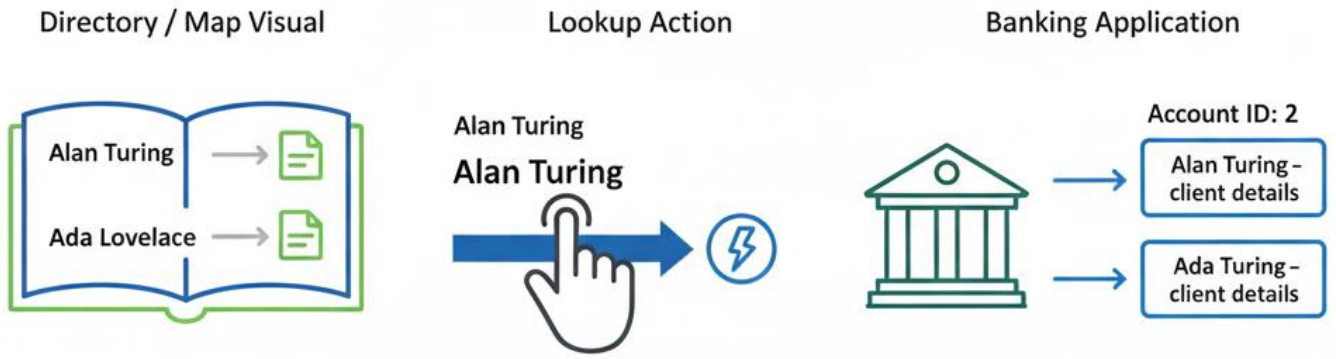
Core Concepts & Functionalities

Hash Maps

UNIVERSITETI METROPOLITAN TIRANA

Store pairs: key finds value instantly. Like a phone directory lookup

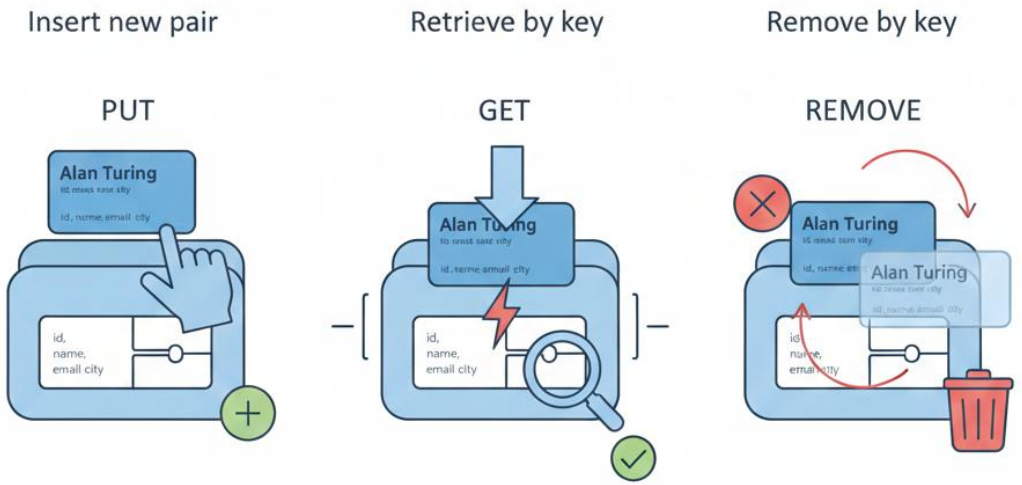
- Key-value pairs: ask for key, get value back instantly. Not index-based
- High speed: nearly instant lookup, regardless of how many pairs stored



- Real banking: Account ID → Account Details.
  Customer name → Customer object. Perfect fit

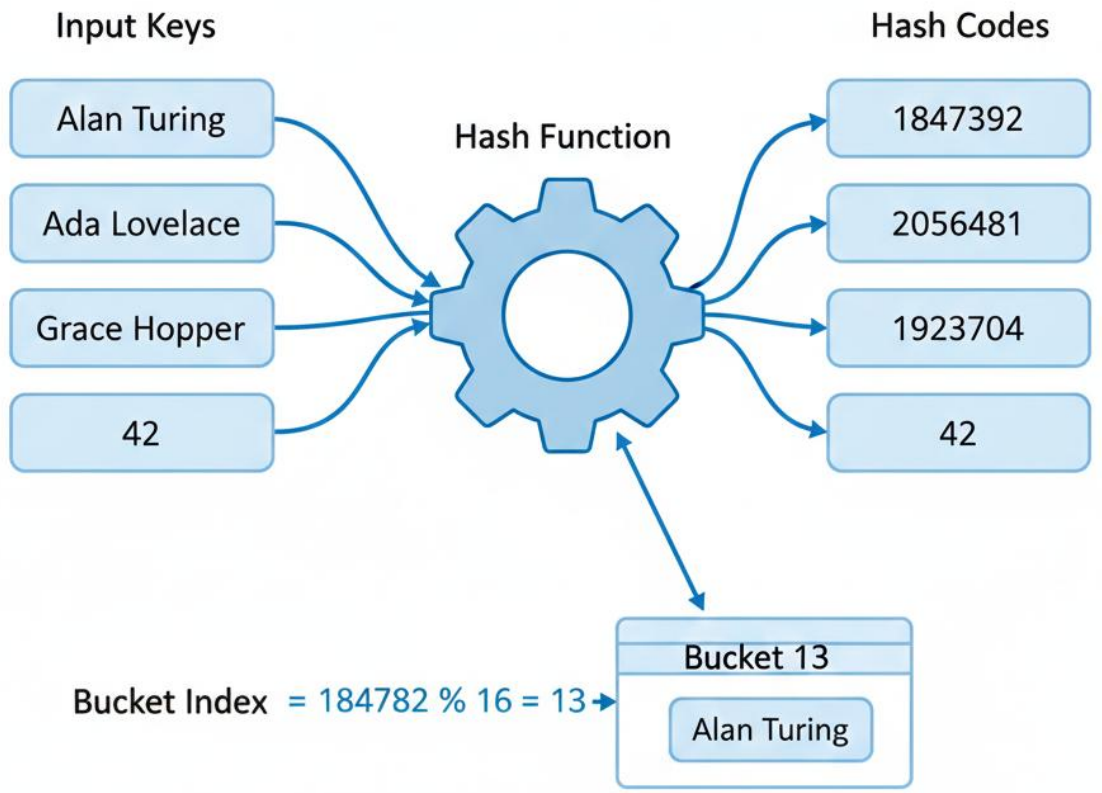
Put pairs in, get them out by key. Four main operations to know

- put(key, value): store a pair. If key exists, update value. If new, insert pair
- get(key): retrieve value by key. Returns null if key doesn't exist. Check first
- remove(key) & size(): delete a pair, or count how many pairs exist in map



Hash function: key → number.
That number tells HashMap where to look

- Hash function: converts any key (String, Integer, etc.) into a number (hash code)
- Bucket location: hash code determines which internal bucket holds that key-value pair
- Why fast: No searching through all pairs hash code goes directly to the bucket

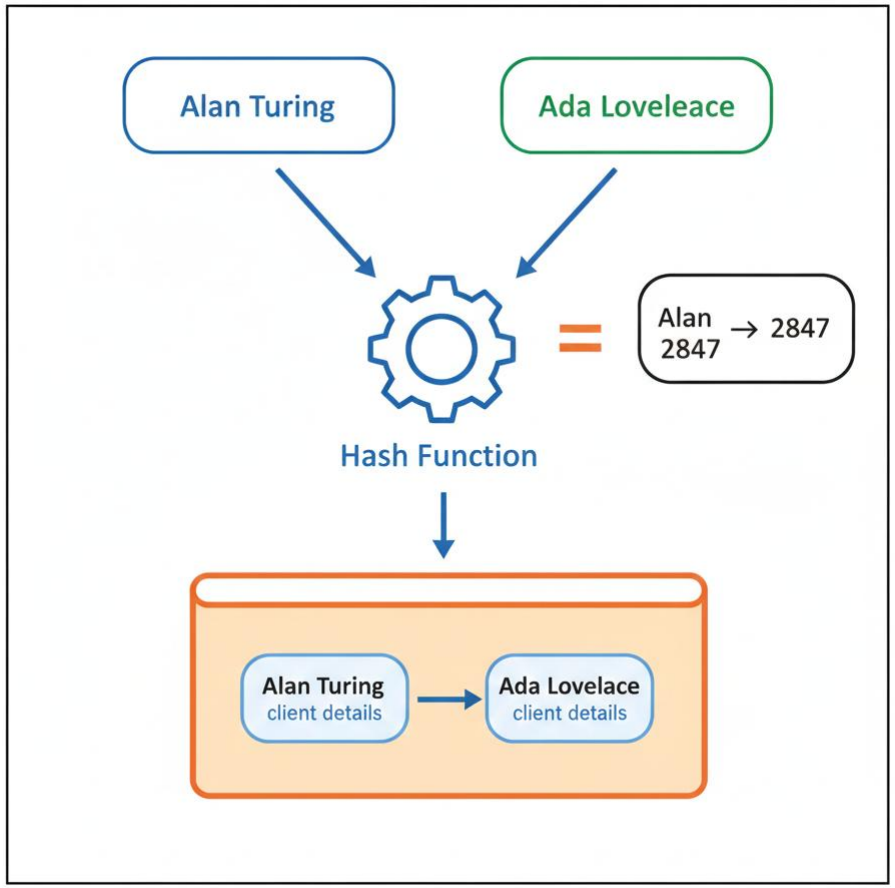


Good hash functions distribute keys evenly.
Fewer collisions mean faster lookups

- Consistency: same key always produces same hash code. Essential for HashMap to work
- Distribution: good hash functions spread keys across all buckets evenly, avoiding clusters
- Index calculation: index = hash % arraySize. This maps any hash to a valid bucket

Two different keys same hash: collision.
HashMap chains them in one bucket

- Collision: two different keys produce same hash code, hash to same bucket index
- Solution: **Chain multiple entries in one bucket** (linked list). HashMap handles automatically
- Impact: if hash function is good, collisions are few

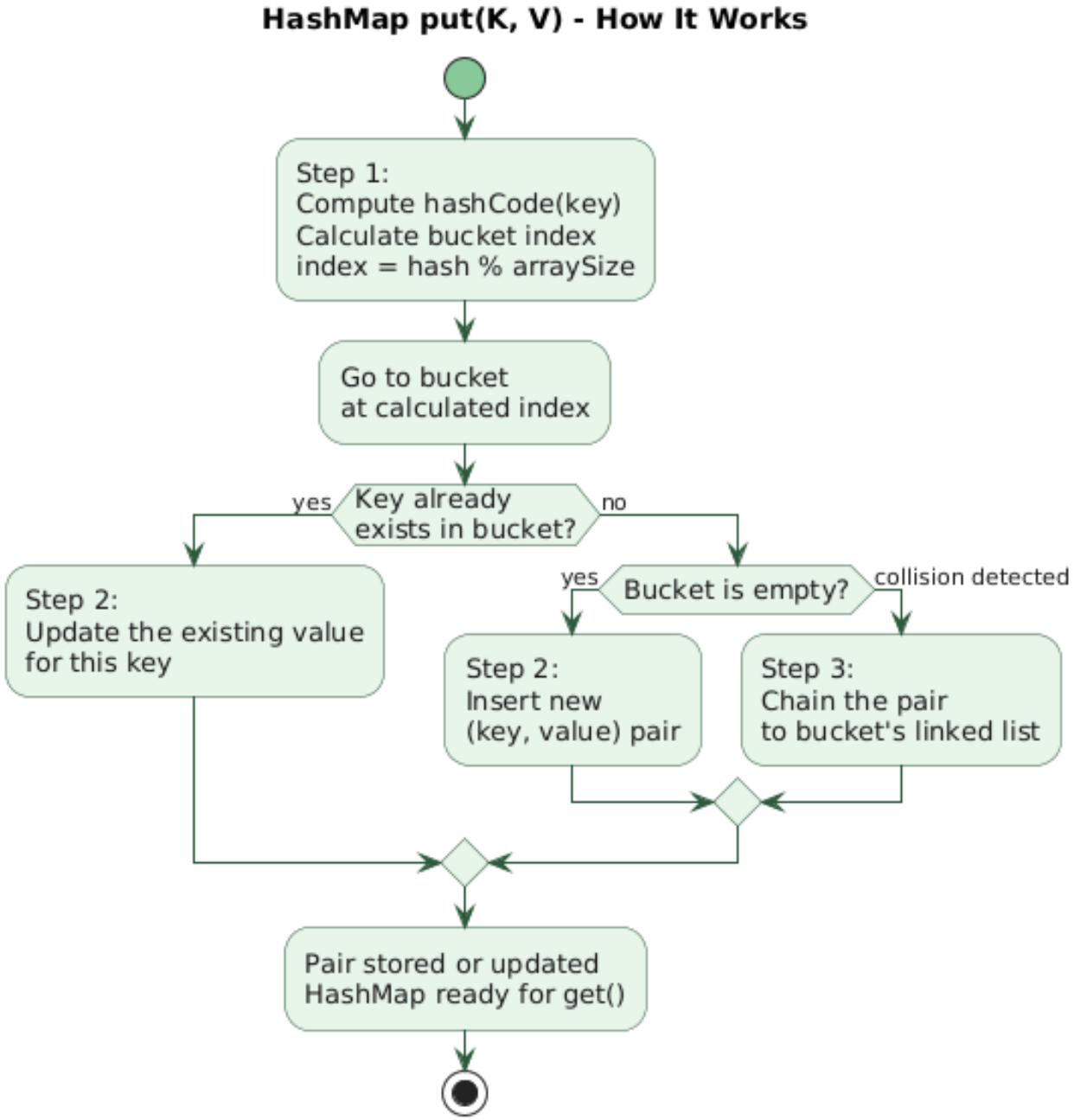


Store a pair

hash key, find bucket,

insert or update.

Automatic



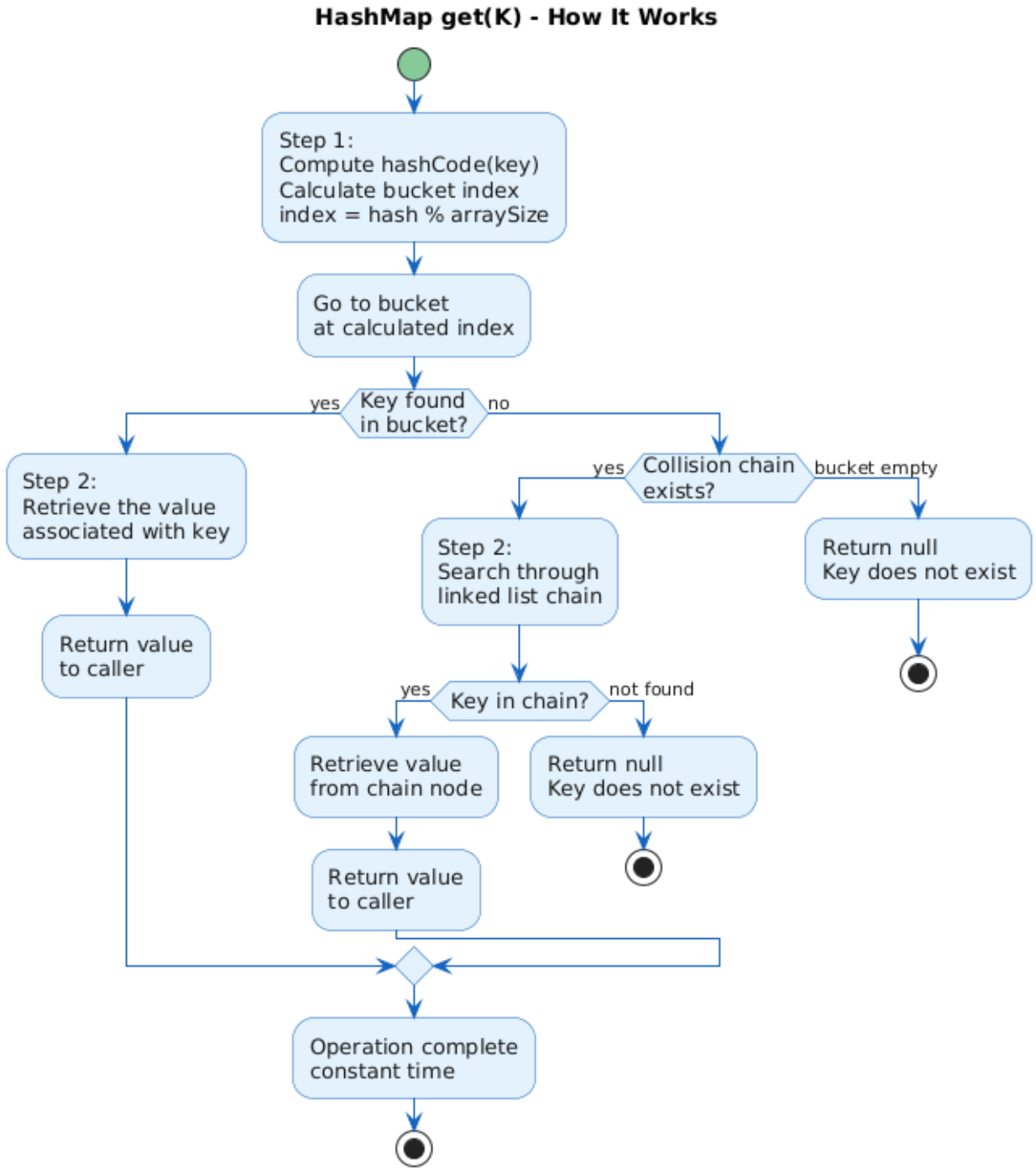
Find value by key:

hash key,

search bucket,

return or null.

Nearly instant



Seven essential methods. Master these

- Core: put(K,V), get(K), remove(K), containsKey(K).
  Fundamental operations for pairs.

- View collections: keySet(), values(), entrySet().
  Access all keys, values, or both

- Iteration: Loop through keySet, values, or entrySet
  Safe with for-each loop

Collections organize data efficiently. Choose the right one. HashMap is your fast friend

- Collections beat arrays: Dynamic sizing, rich methods, type safety with generics matter
- HashMap magic: Hash function finds values instantly. Collisions handled automatically
- Choose wisely: Fast lookup? HashMap. Ordered? ArrayList. Unique? HashSet. Match problem to tool.

UNIVERSITETI METROPOLITAN TIRANA

You can’t sort out life.

But you can at least sort your data