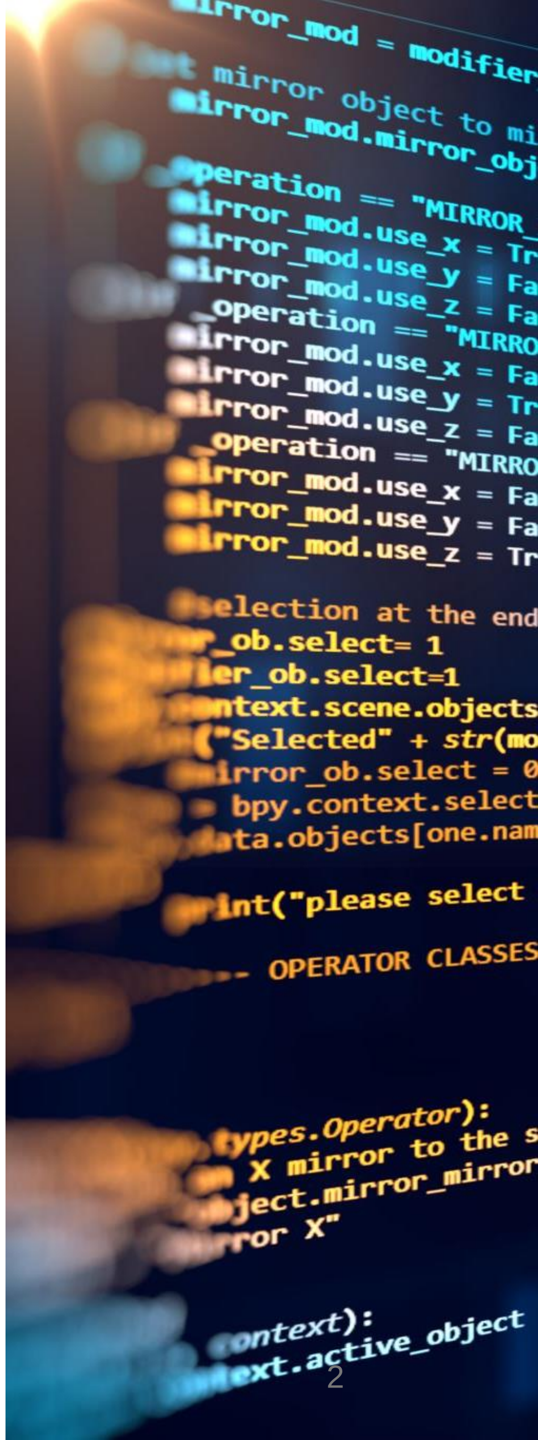
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

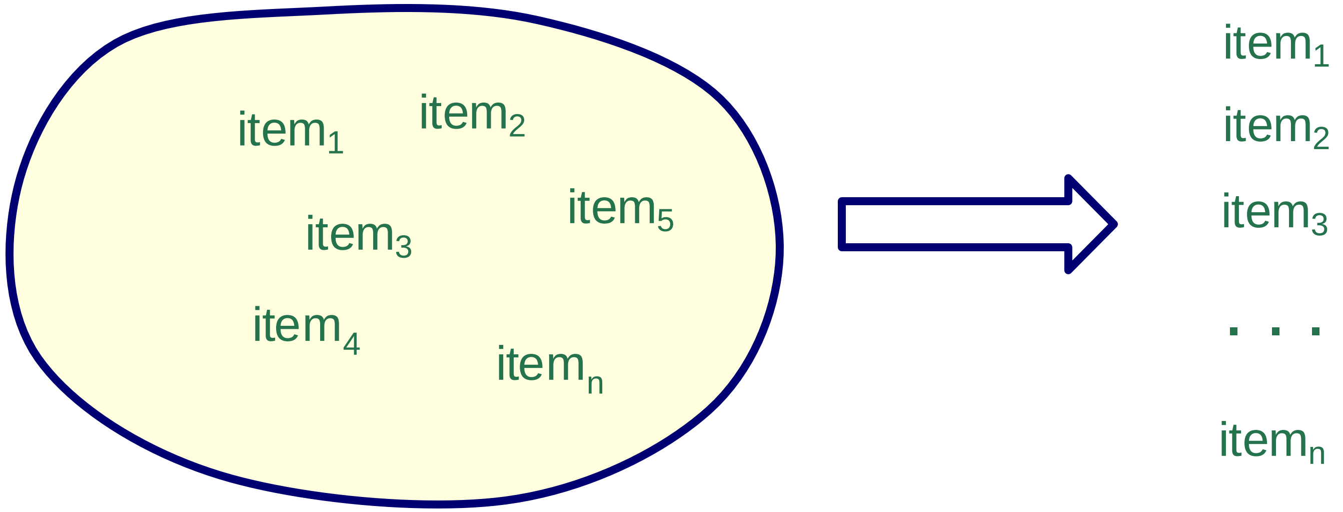
Iterator ADT

Ahmed Ibrahim



Motivation

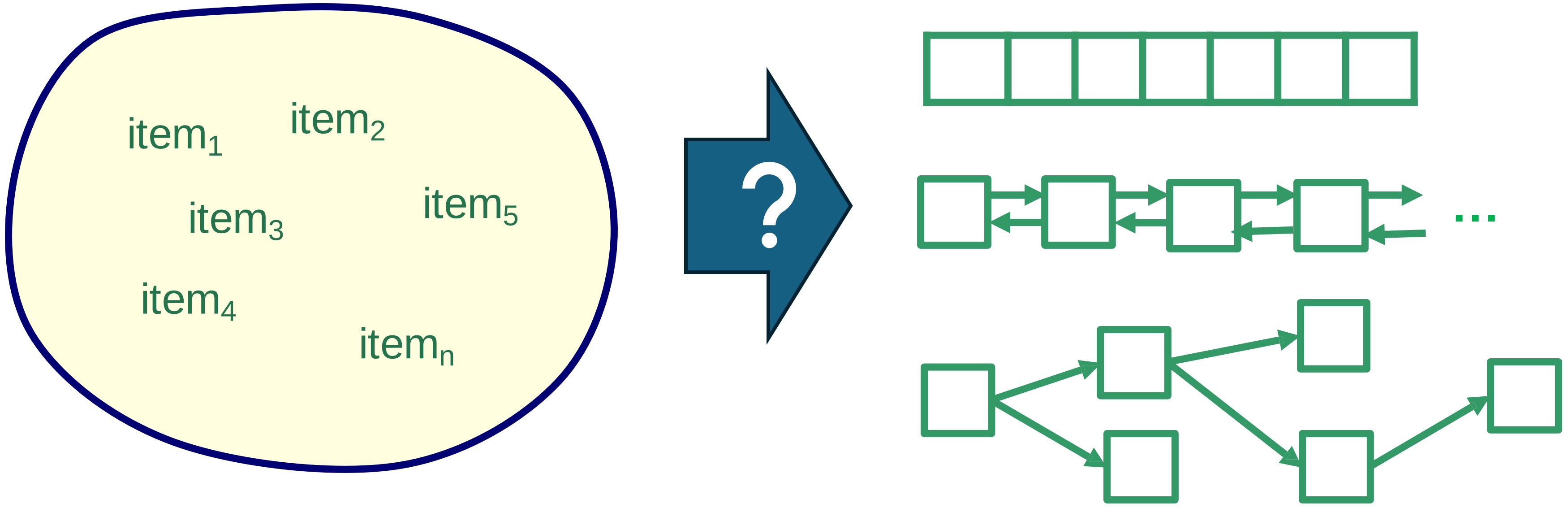
- In many applications, we need to access the items in a collection one at a time.
  - We call this *traversing* or *iterating over, stepping through,* or *visiting every item in* the collection.



Can we traverse a *collection* of objects if we do not know the underlying data structure used to store the data items?

Which data structure is used?

- How can we traverse a *collection* of objects if we do not know the underlying data structure used to store the data items?



Iterators ADT

- An iterator is an abstract data type that allows us to access the data items in a collection one by one.
- An iterator provides 3 operations:
  - next: returns the next data item in the collection. Error if the data item does not exist
  - hasNext: returns **true** if more data items have not been accessed; **false** otherwise
  - remove: removes the last data item returned by the next operation

Why use Iterators?

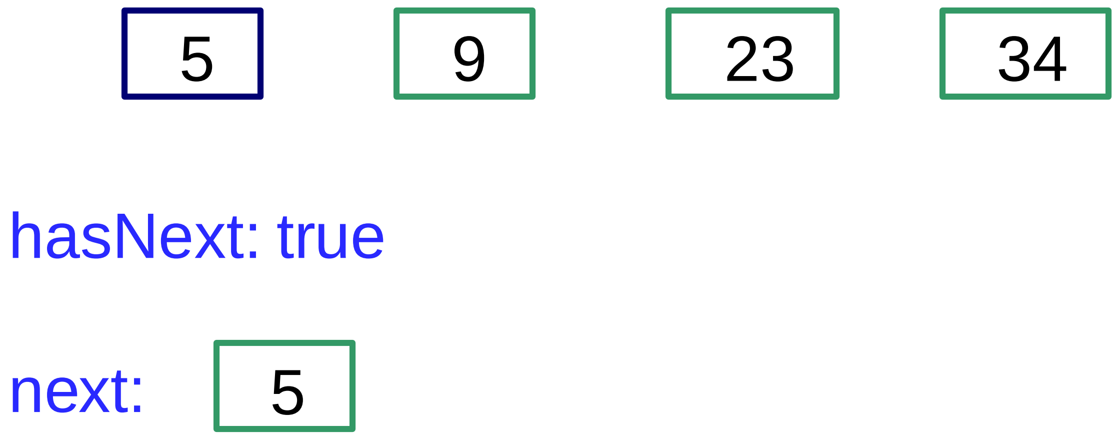
- In programming, it’s common to go through each element in a collection, and iterators offer a **consistent** way to do this across different collections.
- **Advantage**: With an iterator, we don’t need to know how the collection is structured internally!
- For example, can you tell which type of collection this code is accessing?

```
while (iter.hasNext()) {System.out.println(iter.next());} // if it's true
```

- This code works the same way for any collection, whether an **ArrayList**, **LinkedList** or other data structure.
  - An **ArrayList** is a resizable array-like data structure in Java, which is part of **Java.util package**.

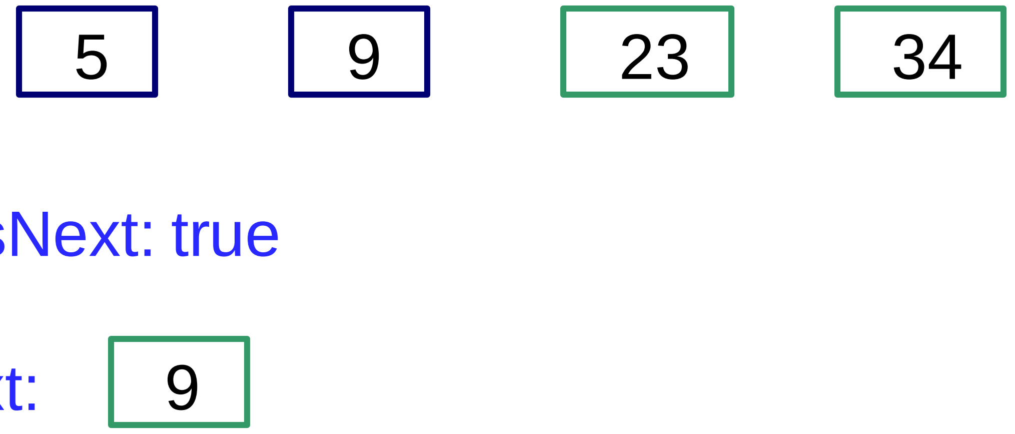
Iterators

- Consider an iterator for a collection storing the following data items:



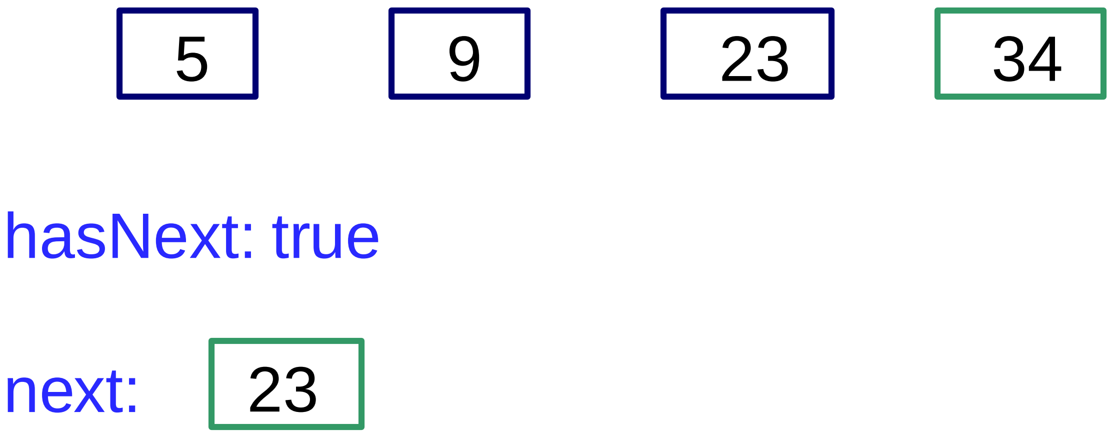
Iterators

- Consider an iterator for a collection storing the following data items:



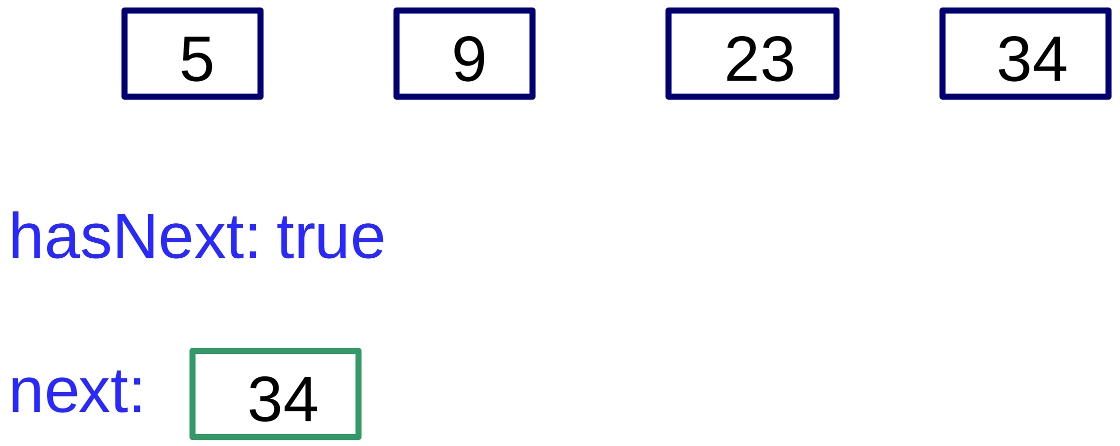
Iterators

- Consider an iterator for a collection storing the following data items:



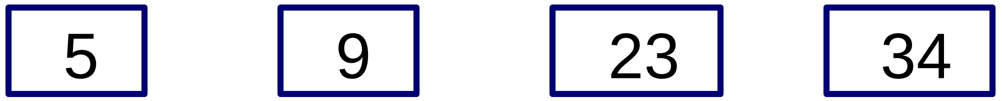
Iterators

- Consider an iterator for a collection storing the following data items:



Iterators

- Consider an iterator for a collection storing the following data items:



hasNext: false

next: NoSuchElementException thrown

Iterator Interface

- The java.util package in the Java API includes an interface called Iterator<T>, which defines the methods for implementing an iterator Abstract Data Type (ADT).
- The Iterator<T> interface provides the following key methods:

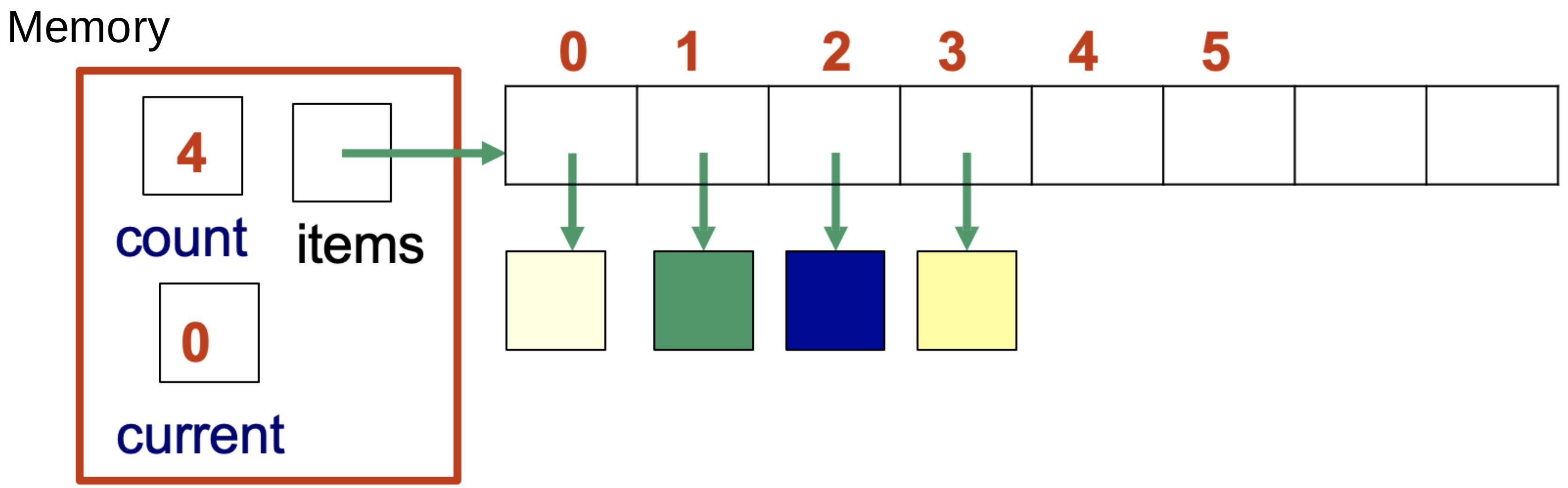
```java
public interface Iterator<T> {
  public boolean hasNext(); // Checks if there are more elements to iterate over

  // Retrieves the next element in the iteration sequence
  public T next() throws NoSuchElementException;

  // Removes the last element returned by the iterator (optional)
  public void remove() throws UnsupportedOperationException, IllegalStateException;
}
```

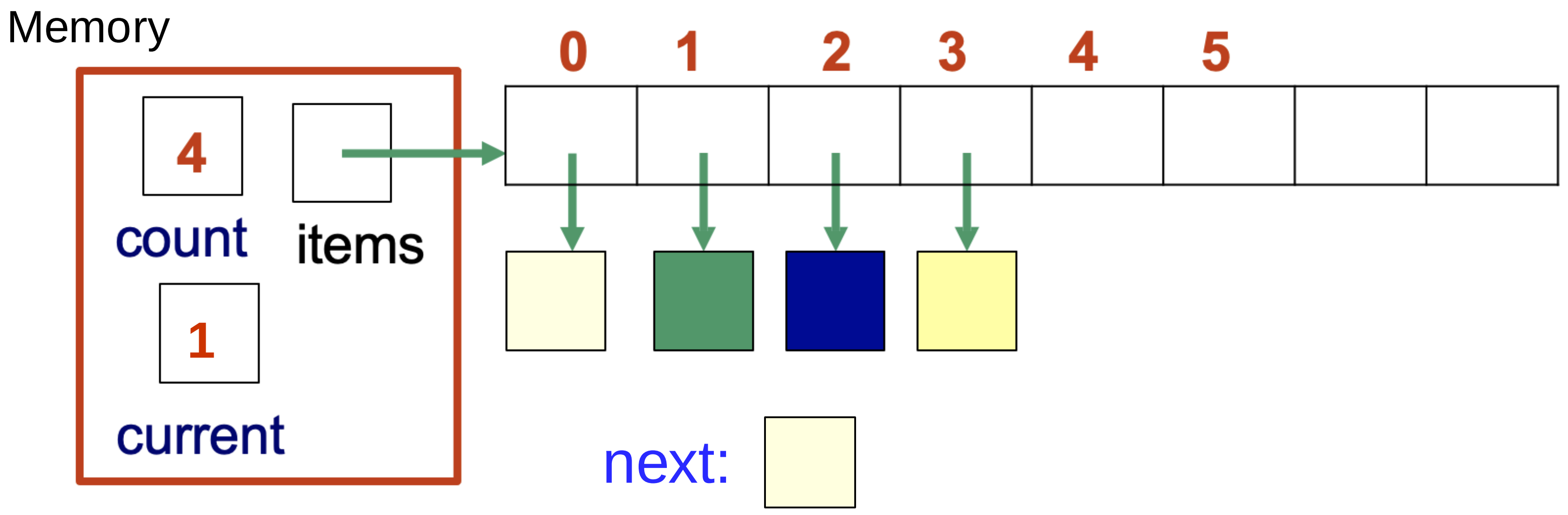
Array Implementation of an Iterator

- Store the data items in an array
- Fix first data item of the list at index 0
- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)



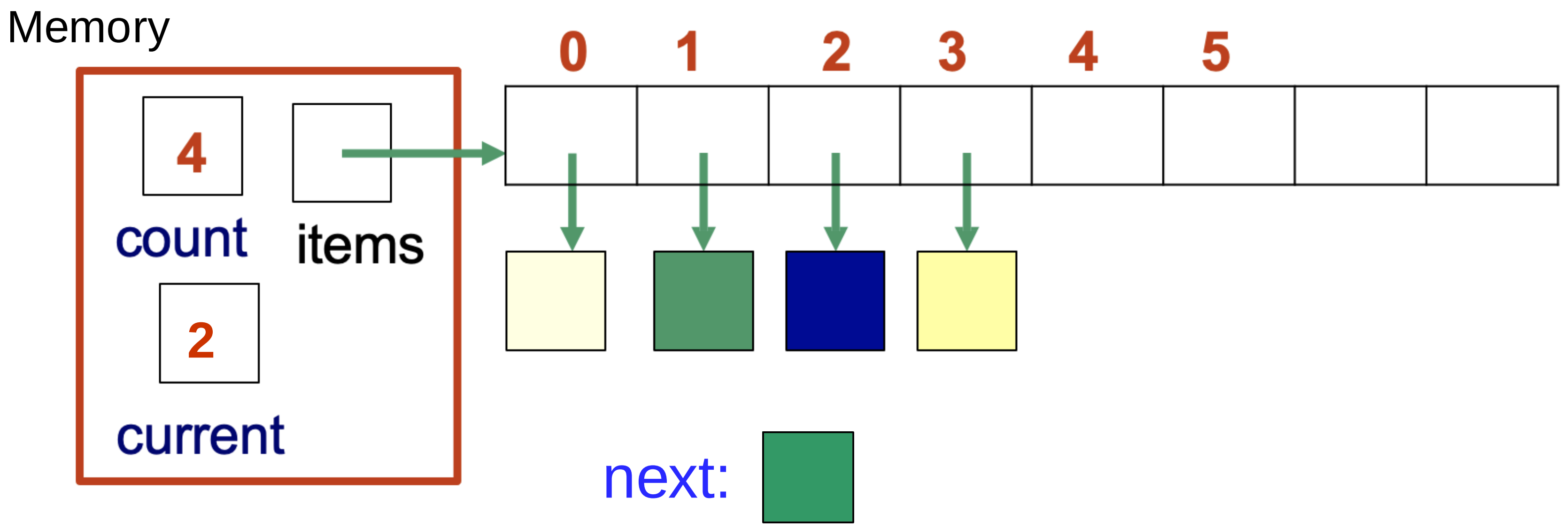
Array Implementation of an Iterator

- Store the data items in an array
- Fix first data item of the list at index 0
- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)



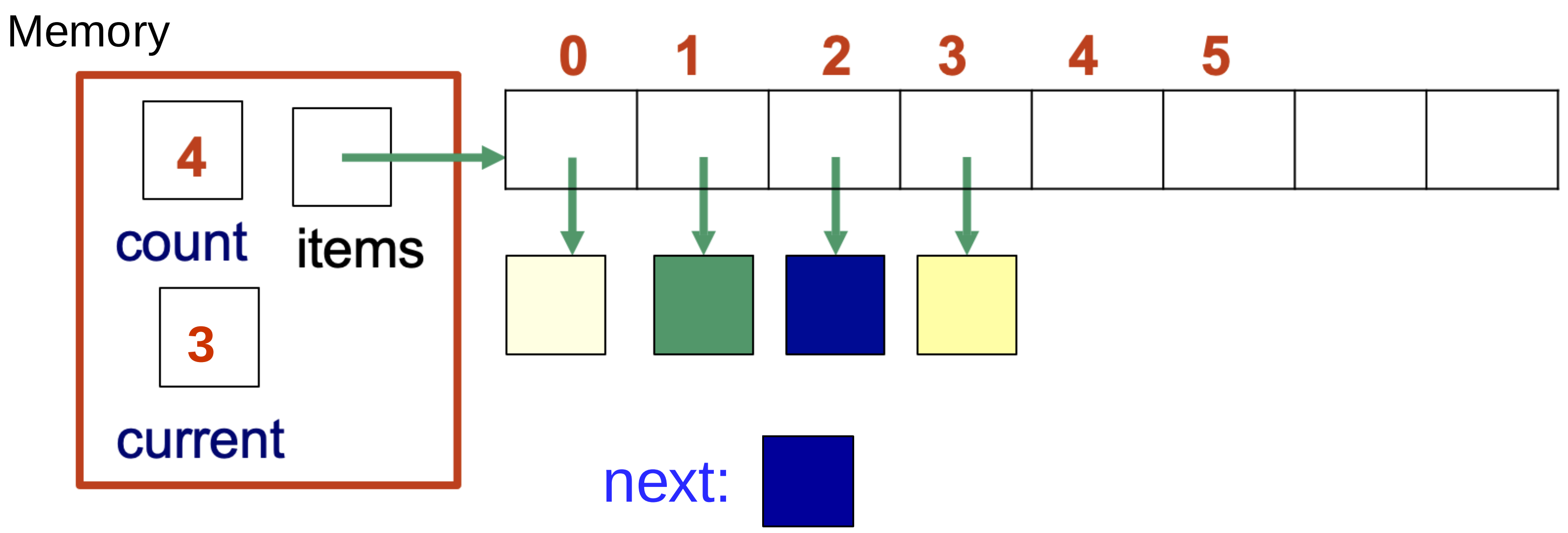
Array Implementation of an Iterator

- Store the data items in an array
- Fix first data item of the list at index 0
- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)



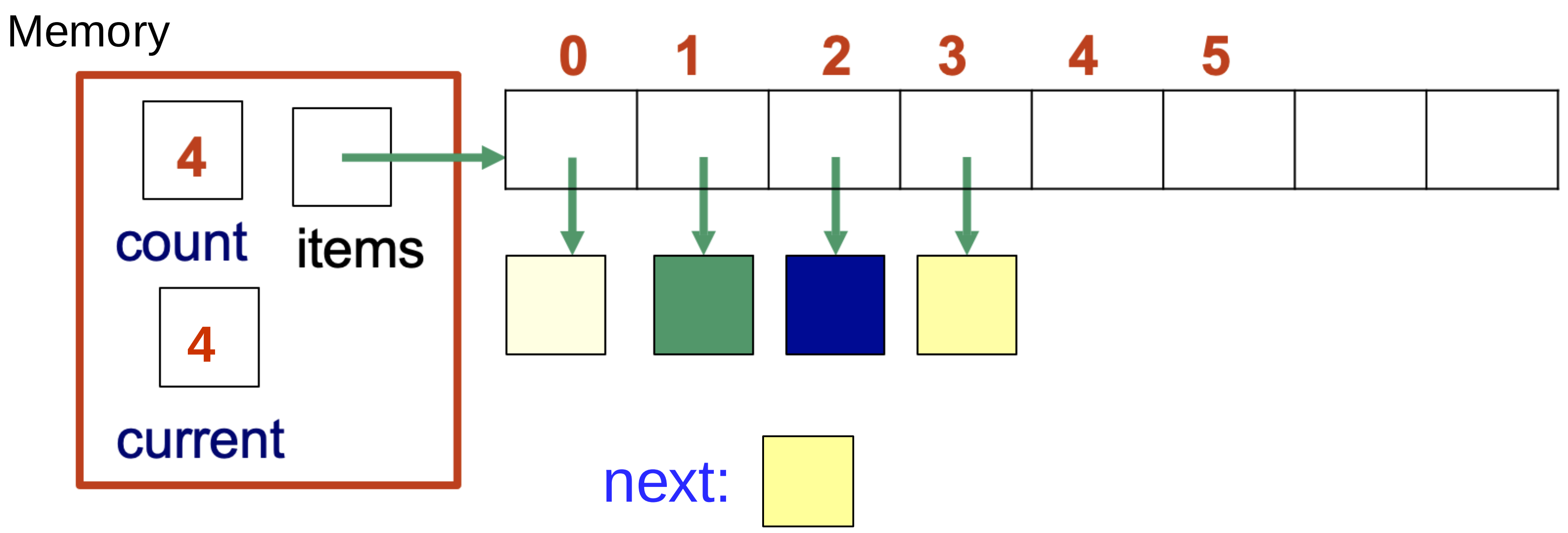
Array Implementation of an Iterator

- Store the data items in an array
- Fix first data item of the list at index 0
- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)



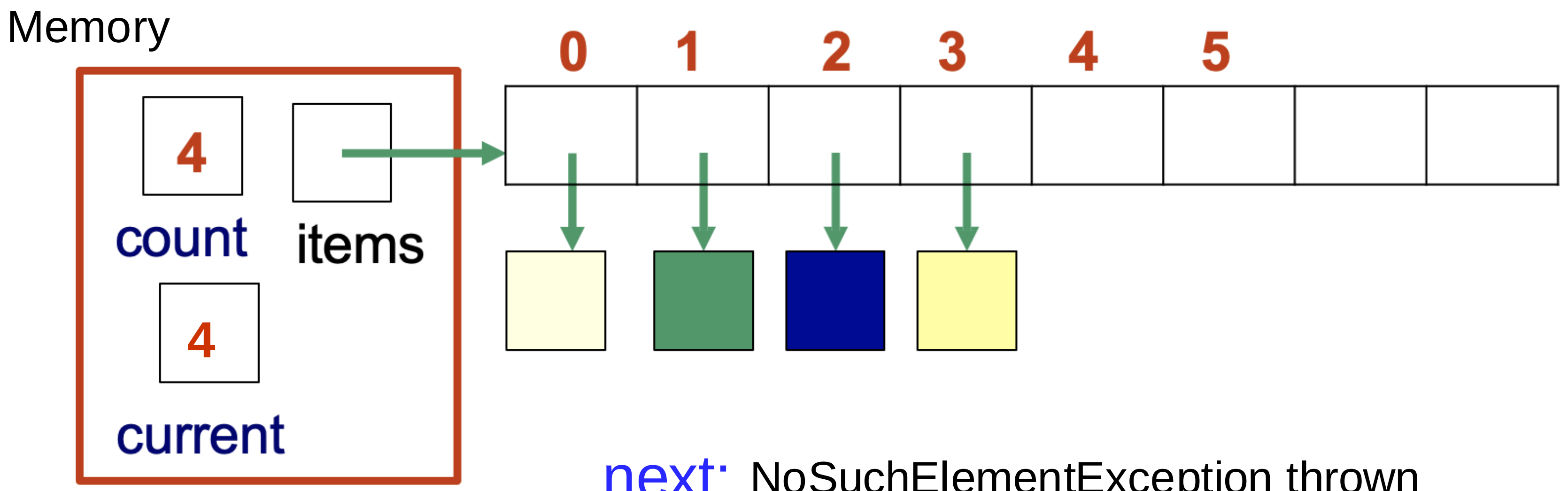
Array Implementation of an Iterator

- Store the data items in an array
- Fix first data item of the list at index 0
- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)



Array Implementation of an Iterator

- Variable *count* indicates the number of data items
- Variable *current* indicates the current position of the iteration (next item to access)
- When *current* = *count*, a NoSuchElementException should be indicated.



next: NoSuchElementException thrown

Array Implementation of an Iterator

```
import java.util.*;

public class ArrayIterator<T> implements Iterator<T> {
private int count; // Total number of elements in the collection
private int current; // Current position in the iteration
private T[] items; // Array storing the items in the collection

// Constructor to initialize the iterator with a specified collection of items and its size
public ArrayIterator(T[] collection, int size) {
this.items = collection;
this.count = size;
this.current = 0;}
}
```

Array Implementation of an Iterator

```
// Checks if there is at least one more element in the iteration
@Override
public boolean hasNext() {
return current < count;
}

// Retrieves the next element in the iteration.
// Throws NoSuchElementException if no more elements are available.
@Override
public T next() {
if (!hasNext()) {throw new NoSuchElementException();}
T result = items[current];
current++;
return result;}
```

Question!

Given an iterator for the array {7, 14, 21, 28, 35}, you perform these operations in order:

1. hasNext()
2. next()
3. next()
4. hasNext()
5. next()
6. remove()
7. next()
8. hasNext()
9. next()
10. hasNext()

What will be the final output or behaviour of the last operation (hasNext())?

01:00

A) false – No elements remaining.

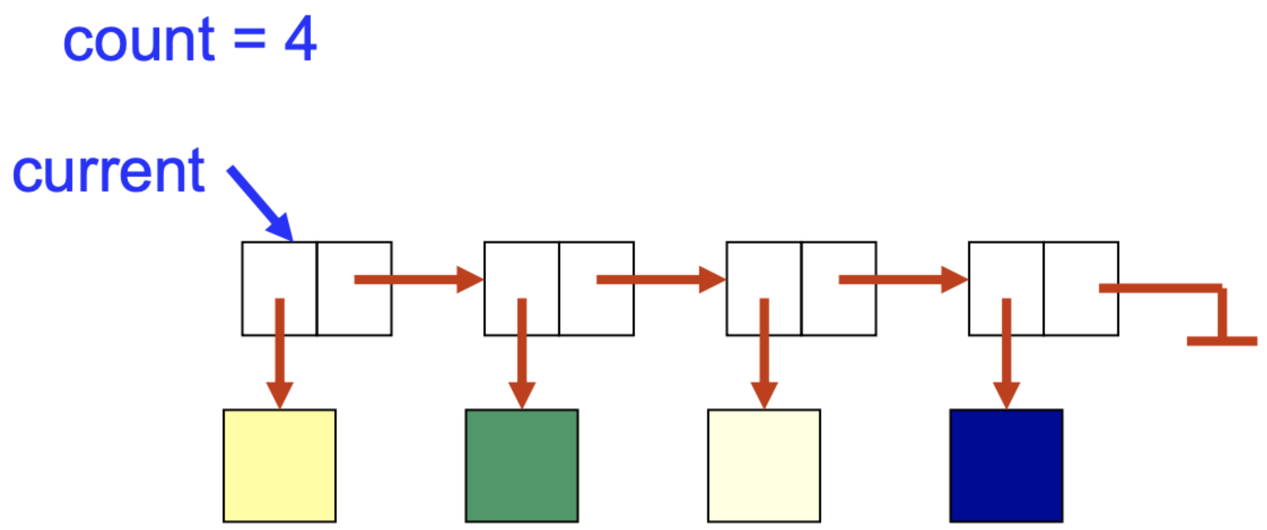
B) true – One element remaining.

C) false – Exception is thrown earlier.

D) true – Exception on last next() call.

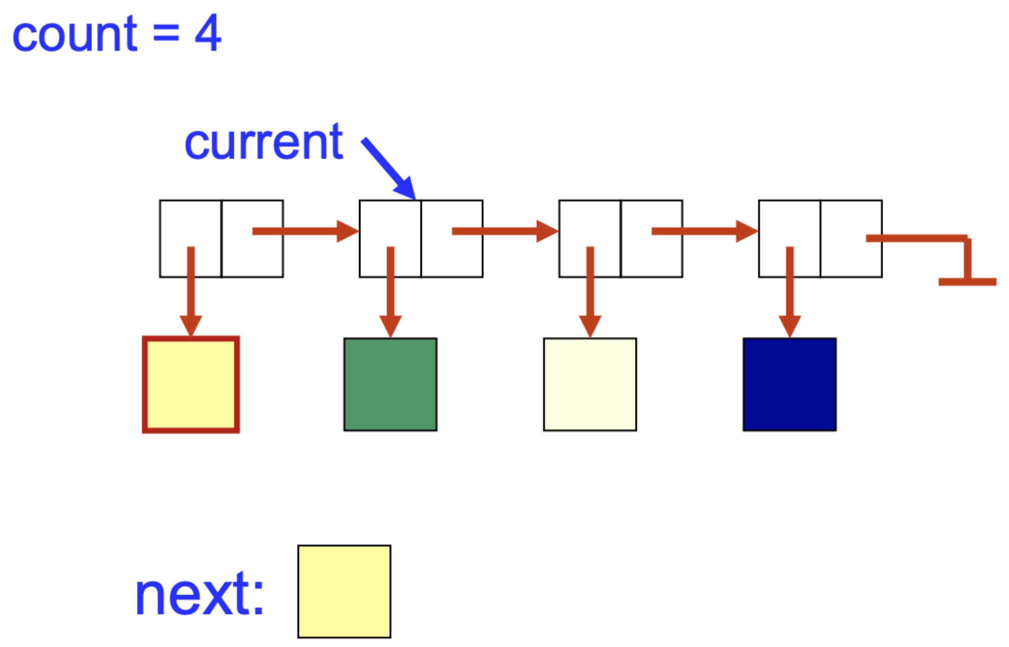
Linked Implementation of an Iterator

- Data items are stored in a linked list
- Variable *count* specifies the number of data items
- Variable *current* points to the node storing the *next* data item to be returned by the iterator



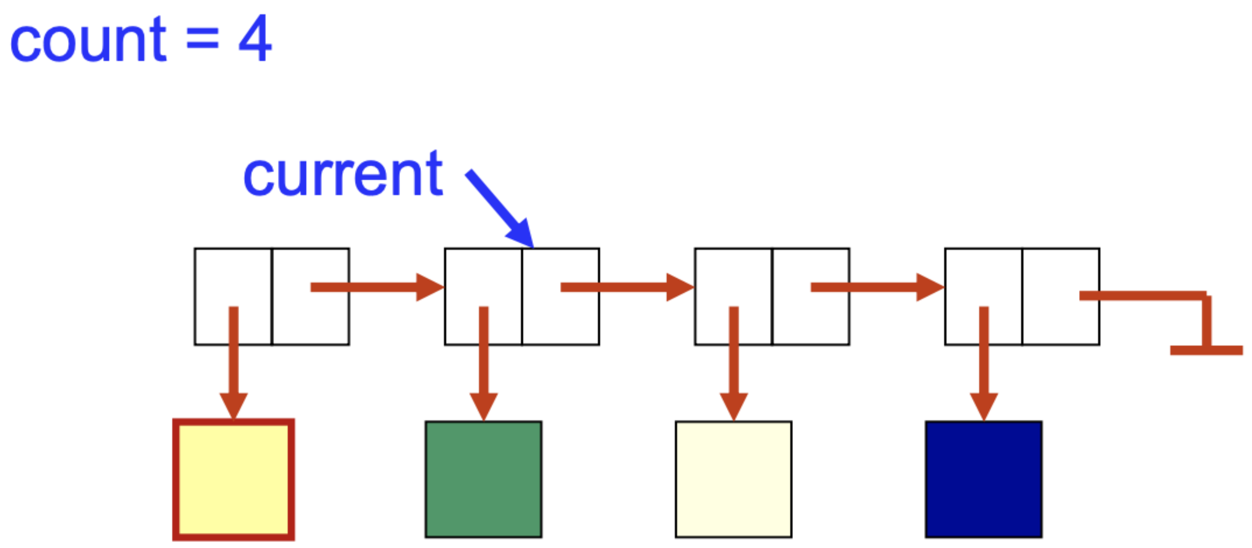
Linked Implementation of an Iterator

- Data items are stored in a linked list
- Variable *count* specifies the number of data items
- Variable *current* points to the node storing the *next* data item to be returned by the iterator



Linked Implementation of an Iterator

- Note that once a data item has been accessed the previous data item cannot be accessed again.



Linked Implementation of an Iterator

```
import java.util.*;

public class LinkedIterator<T> implements Iterator<T> {

  private int count; // Total number of elements in the collection
  // Node representing the current position in the iteration
  private LinearNode<T> current;

// Constructor to initialize the iterator with a linked list of items and its size
public LinkedIterator(LinearNode<T> collection, int size) {
                this.current = collection;
                this.count = size;}
}
```

Linked Implementation of an Iterator

```
// Checks if there is at least one more element in the iteration
@Override
public boolean hasNext() {
   return current != null;
}

// Retrieves the next element in the iteration.
// Throws NoSuchElementException if there are no more elements.
@Override
public T next() {
   if (!hasNext()) {throw new NoSuchElementException();}
   T result = current.getElement();
   current = current.getNext();
   return result;
}
```

Java Iterators

- Several collection classes in the Java API, such as **ArrayList**, **LinkedList**, and **TreeSet**, provide a method called iterator().
- This method returns an Iterator that enables sequential access to each item in the collection without requiring knowledge of the collection's internal structure or data organization.

```
import java.util.*;

public class IteratorExample {
public static void main(String[] args) {
// Example using ArrayList
Collection<String> names = new ArrayList<>(List.of("Alice", "Bob", "Charlie"));

// Obtaining an iterator for the collection
Iterator<String> iterator = names.iterator();

// Traversing the collection using the iterator
while (iterator.hasNext()) {System.out.println(iterator.next());}
  }
}
```

Note —A TreeSet in Java is a collection that stores elements in **sorted**, **ascending** order. It is part of the Java.util package and implements the **Set interface**, meaning it does not allow duplicate elements.

Java Iterators (cont.)

- Note that the iterator method returns a type Iterator<T>. However, Iterator<T> is an interface, not a class!
- This means that when a method returns an interface type, it actually returns an object from a class that implements this interface.

Using an Iterator

- When the iterator() method is called on a collection, it returns an "iterator object" for that collection. This iterator allows us to use the *hasNext()* and *next()* methods to move through each element in the collection.
- Example: Suppose we had an ArrayList that was created by

```
ArrayList<Person> myList = new ArrayList<>(); // Assume items are added to myList here
```

and then had items added to it. We can use an iterator to display the contents of **myList**:

```
Iterator<Person> iter = myList.iterator();
    while (iter.hasNext()) {
System.out.println(iter.next());
}
```

- This code will also work with other collections like **LinkedList**, or **TreeSet**, allowing us to print elements in any collection type.

Question!

Consider a LinkedIterator for a linked list {5, 10, 15, 20, 25}. The iterator supports next(), hasNext(), and remove() methods. Which of the following statements is true about the behaviour of the iterator?

A) Calling next() after hasNext() returns false will safely return null.

B) Calling remove() immediately after the iterator is created will throw an IllegalStateException.

C) After calling next() on all elements, hasNext() will continue to return true until remove() is called.

D) The iterator's position resets to the start of the list after remove() is called on the last element.

01:00

Question!

Consider the following code:

```
ArrayList<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5));
Iterator<Integer> iterator = numbers.iterator();

while (iterator.hasNext()) {
 Integer number = iterator.next();
 if (number % 2 == 0) {iterator.remove();}
}
```

What will be the content of the 'numbers' list after executing this code?

A) '[1, 2, 3, 4, 5]'

B) '[1, 3, 5]'

C) '[2, 4]'

D) '[]'

01:00

Question!

What will happen if the following code is executed?

```
ArrayList<String> words = new ArrayList<>(List.of("one", "two", "three"));
Iterator<String> iterator = words.iterator();
while (iterator.hasNext()) {
   String word = iterator.next();
   if (word.equals("two")) {words.remove(word);}
}
```

A) The code will execute successfully, removing **"two"** from the list.

B) The code will throw a **'NoSuchElementException'**.

C) The code will throw a **'ConcurrentModificationException'**.

D) The code will not remove any elements from the list.

01:00

Customization of Iterators

- Sometimes, the default 'Iterator' implementations don’t meet all needs. Creating custom iterators allows developers to specify unique traversal behaviours.
- Examples of Custom Traversal:
  - **Reverse Iterator** – An iterator that traverses a collection in reverse order. This can be particularly useful for collections where elements, such as a stack-like structure, are processed from the end.
  - **Conditional Iterator** – An iterator that only returns elements that meet specific criteria (e.g., only even numbers or objects that satisfy a particular condition).

Reverse Iterator in Java

```
public class ReverseArrayIterator<T> implements Iterator<T> {
private int current;
private T[] items;

public ReverseArrayIterator(T[] collection) {
     this.items = collection;
     this.current = collection.length - 1; // Start from the last element
}
@Override
public boolean hasNext() {return current >= 0;}

@Override
public T next() {
if (!hasNext()) {throw new NoSuchElementException();}
return items[current--];}
}
```

Conditional Iterator in Java

```
public class ConditionalIterator<T> implements Iterator<T> {
private T[] items;
private int current;
private Predicate<T> condition;

public ConditionalIterator(T[] items, Predicate<T> condition) {
        this.items = items;
        this.condition = condition;
// Initialize to the first matching element
         this.current = findNext(0);
}

// Helper method to find the next index that meets the condition
private int findNext(int start) {
for (int i = start; i < items.length; i++) {
        if (condition.test(items[i])) {return i; } }
return -1;} // No more elements satisfying the condition
```

```
@Override
public boolean hasNext() {
return current != -1;}

@Override
public T next() {
if (!hasNext()) {
throw new NoSuchElementException();}
T result = items[current];
current = findNext(current + 1);
// Move to the next matching element
return result;
} }
```

Error Handling in Iterators

- NoSuchElementException with next() – The next() method in an iterator retrieves the next element in the collection. However, if next() is called when there are no more elements (when hasNext() returns false), it will throw a NoSuchElementException.
  - **Solution**: Always check hasNext() before calling next() to confirm an available element. This is often done in a while loop:

```
while (iterator.hasNext()) {System.out.println(iterator.next());}
```

- UnsupportedOperationException with remove() – The remove() method in an iterator is optional, meaning not all collections support it. If you call remove() on an iterator that doesn't support this operation, it will throw an UnsupportedOperationException.
  - Solution: wrap remove() in a try-catch block

Error Handling in Iterators (cont.)

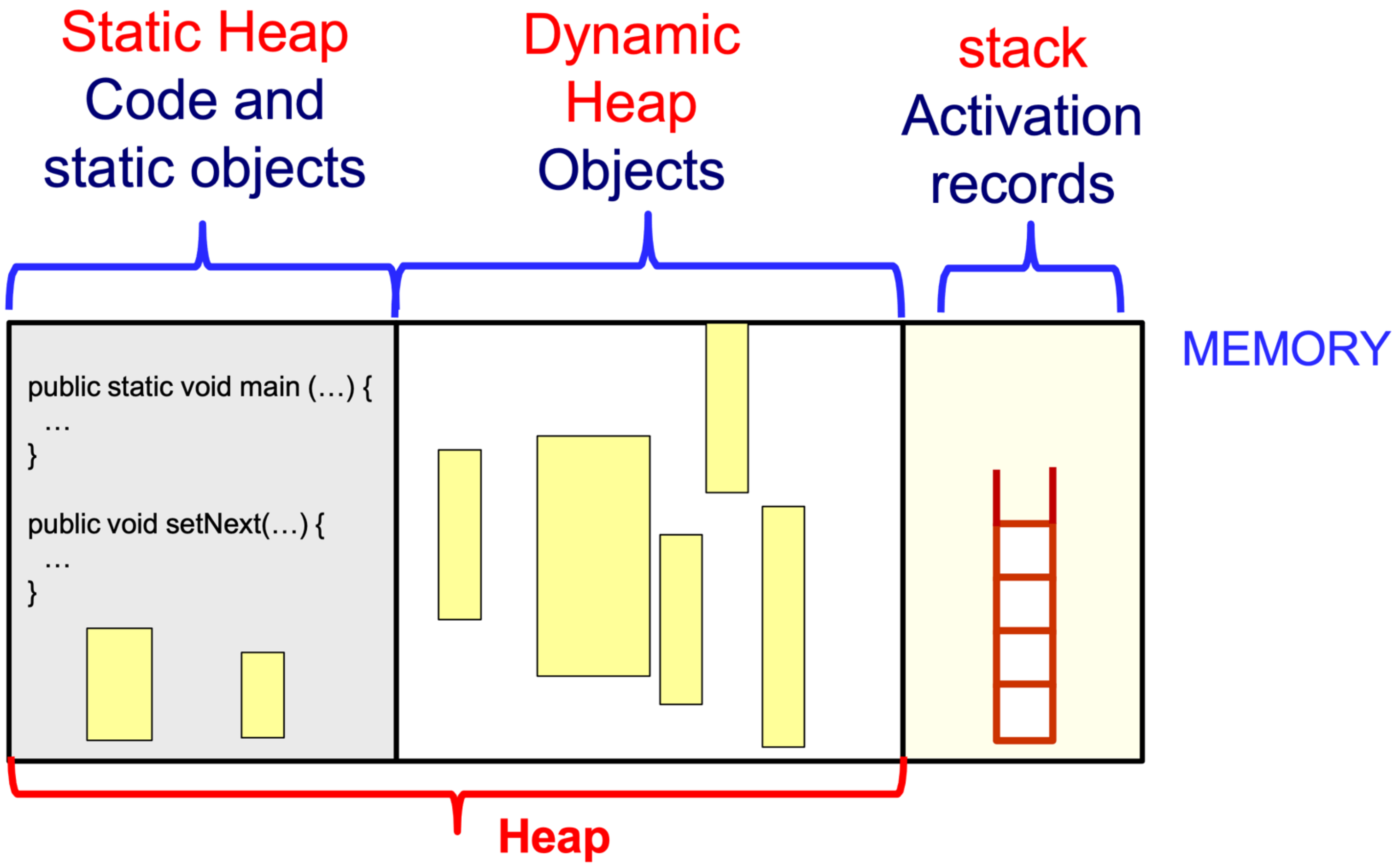
- IllegalStateException with remove() – The remove() method will throw an IllegalStateException if it is called before next() has been called in the iteration or if it's called **twice** in a row without another call to next().
  - Solution: Only call remove() immediately after the next() to ensure it applies to the most recently retrieved element.
  - Example:

```
while (iterator.hasNext()) {
String element = iterator.next();
  if (element.equals("someValue")) {iterator.remove();} // Only call after next()
}
```

Memory Allocation in Java

Memory Allocation in Java

- When a program is being executed, separate areas of memory are allocated:

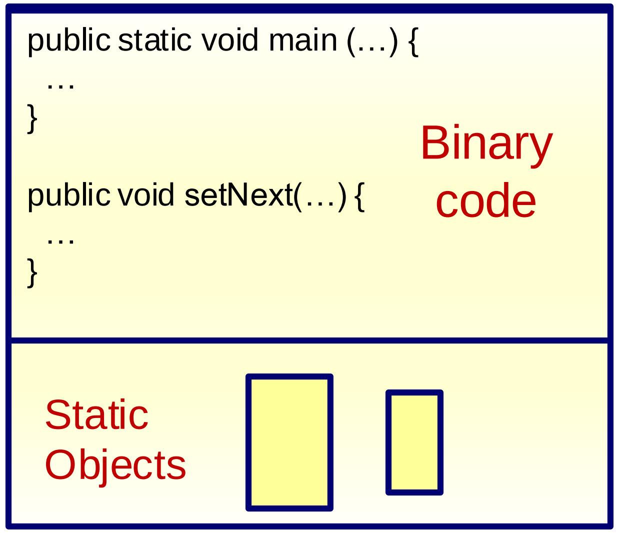


Static Heap

Used to store

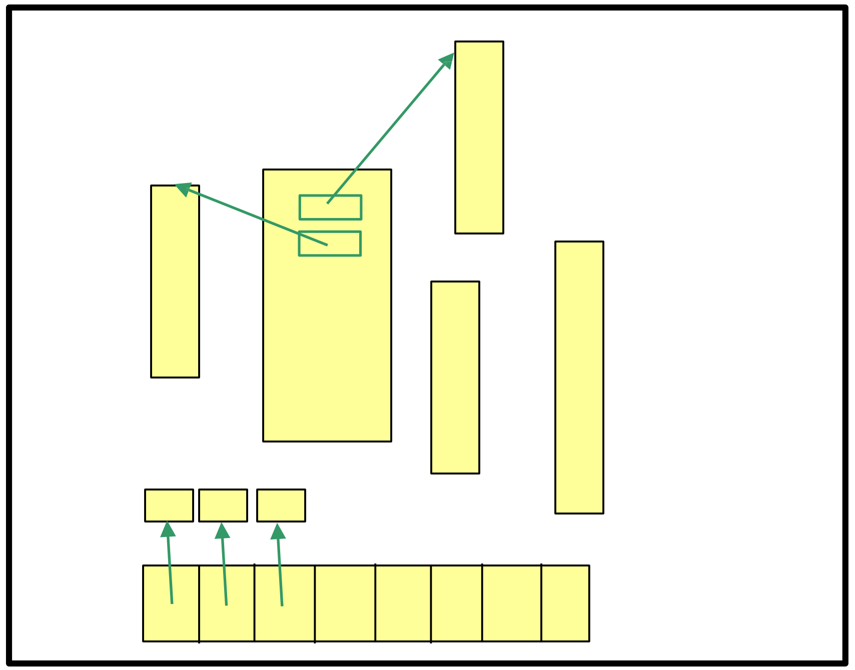
- code for class methods
- static objects

- The amount of memory used for this area is fixed (does not change) during a program's execution because the code does not change, and new static objects cannot be created.



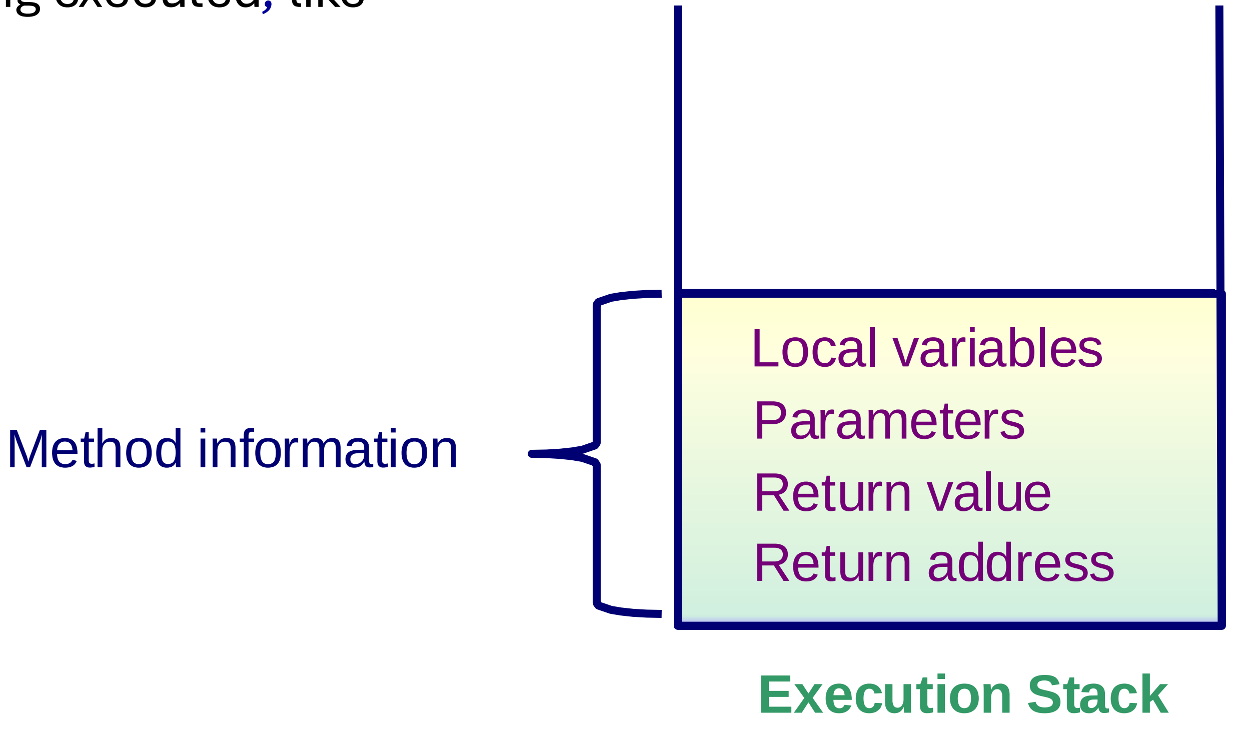
Dynamic Heap

- Used to store objects dynamically created during the execution of a program. Information that is stored for each object:
    - values of its instance variables
    - reference to its code

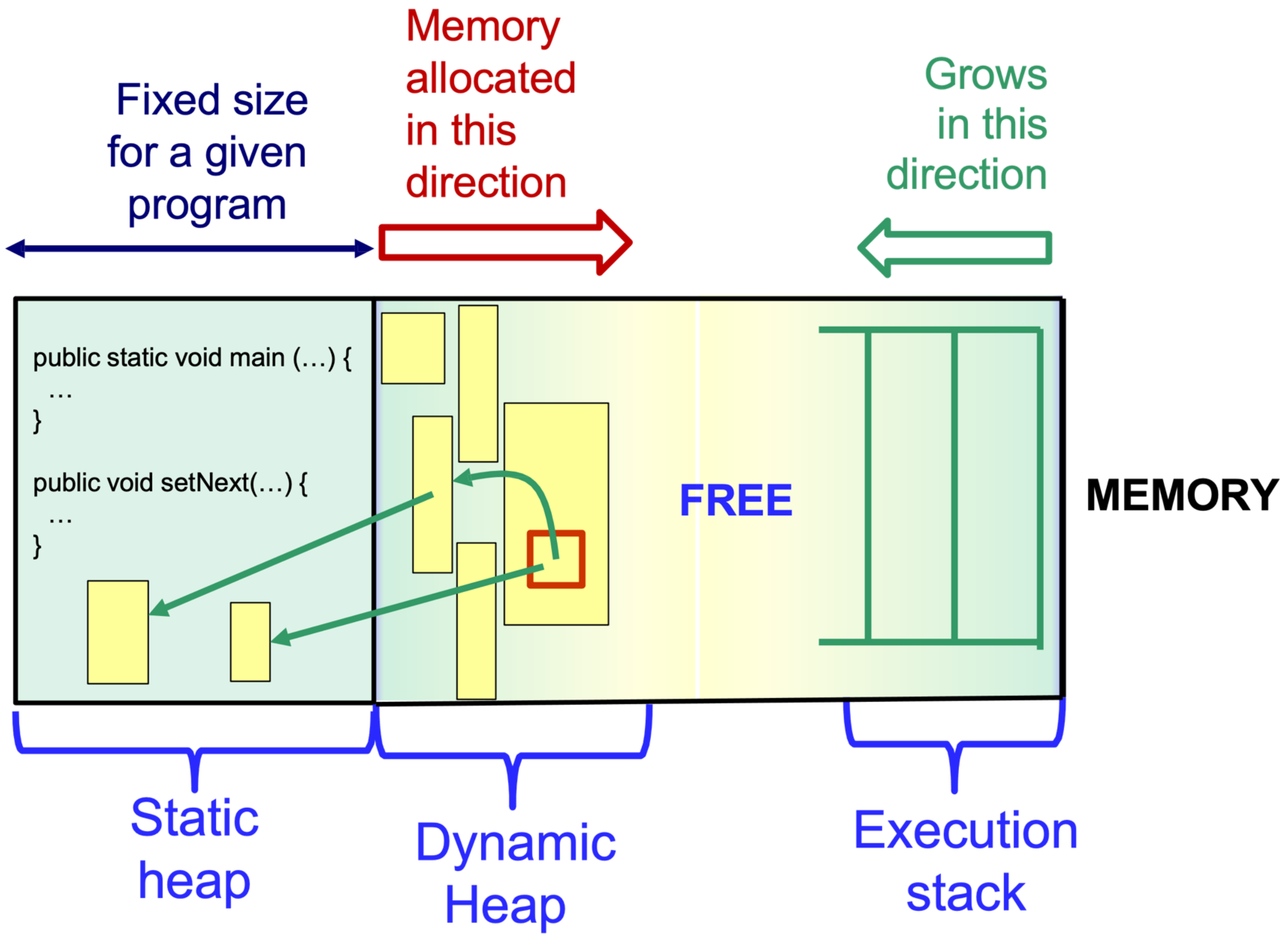


Execution Stack

- The execution stack (also called runtime stack or call stack) is used to store information needed while a method is being executed, like
  - Local variables
  - Formal parameters
  - Return value
  - Return address

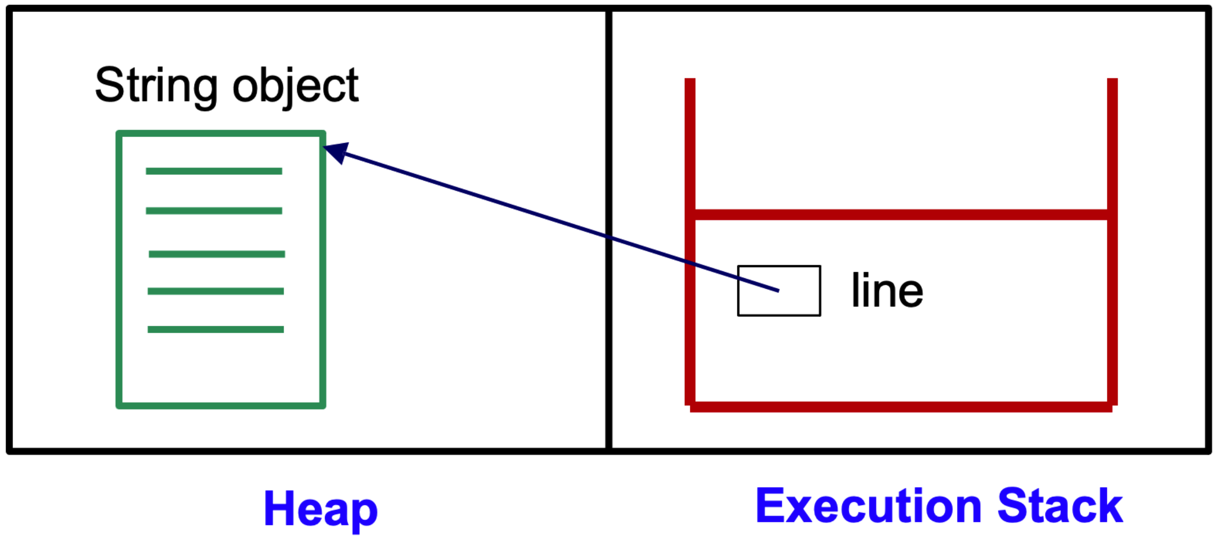


Memory Allocation for a Program



Memory Allocation in Java

- What happens when an object is created in a method by the new operator, as in
  - `String line = new String("hello");?`
    - The local variable line has memory allocated to it in the execution stack
    - The object is created in the heap



Execution Stack

- Execution stack (or runtime stack or call stack) is the memory space used to store the information needed by a method while the method is being executed
- When a method is invoked, an activation record (call frame, stack frame, or frame) for that method is created and pushed onto the execution stack.
- All the information needed during the execution of the method is stored in its activation record.

Execution Stack

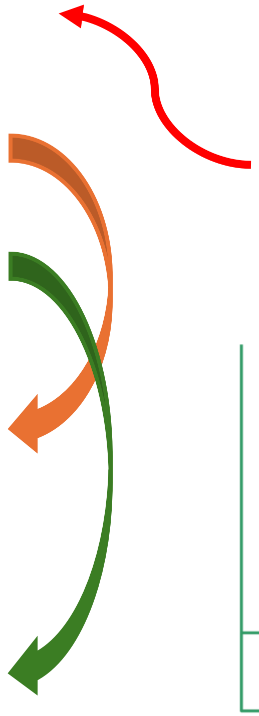
- An activation record contains:
  - Address to return to after method ends
  - Method's parameters
  - Method's local variables
  - Return value (if any)
- Note that the values stored in an activation record are accessible only while the corresponding method is being executed!

How Programs are Executed

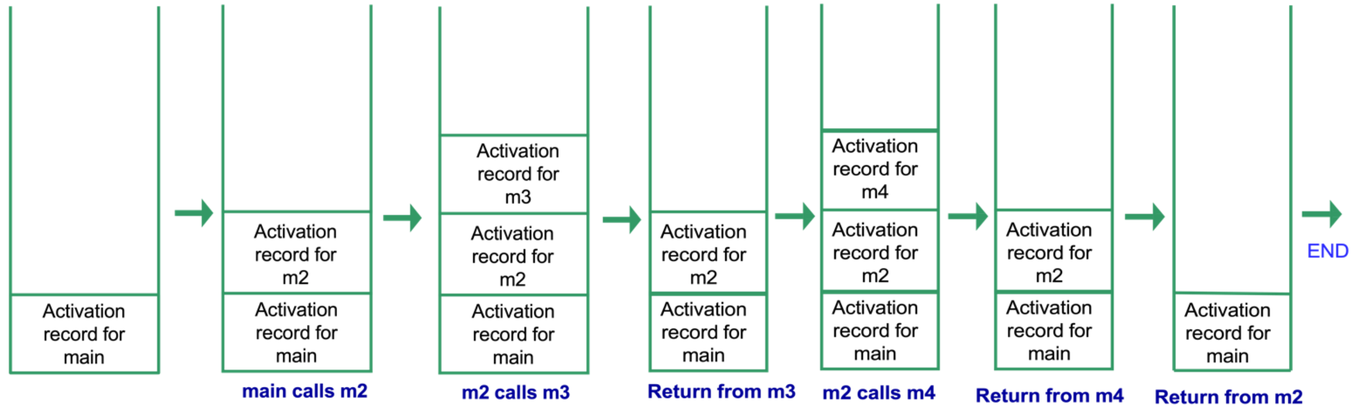
```
public static void m2() {
System.out.println("Starting m2");
System.out.println("m2 calling m3");
m3();
System.out.println("m2 calling m4");
m4();
System.out.println("Leaving m2");
return;}

public static void m3() {
System.out.println("Starting m3");
System.out.println("Leaving m3");
return;}

public static void m4() {
System.out.println("Starting m4");
System.out.println("Leaving m4");
return;}
```



```
public static void main(String args[]) {
System.out.println("Starting main");
System.out.println("main calling m2");
m2();
System.out.println("Leaving main");
}
```



Execution Stack for Execution of the above Program

Execution of the Program

- When the **main** method is invoked:
  - An activation record for the main is created and pushed onto the execution stack
- When the **main** calls the method **m2**:
  - An activation record for m2 is created and pushed onto the execution stack
- When **m2** calls **m3**:
  - An activation record for m3 is created and pushed onto the execution stack
- When **m3** terminates, its activation record is popped off and control returns to **m2**

Thank you