Advanced Java - Part 1

Default Methods, Lambdas, ArrayList vs LinkedList

Presented by **João Bispo**
E-mail: jbispo@fe.up.pt

Default Methods

- Interfaces now can define implementations
- Ideal for convenience methods, replaces some uses of Abstract Class

Default Methods

- Interfaces now can define implementations
- Ideal for convenience methods, replaces some uses of Abstract Class

```
interface TreeNode<K>

  List<K> getChildren();

  boolean hasChildren();
```

Can use:

- Other interface methods
- **this**

Default Methods

- Interfaces now can define implementations
- Ideal for convenience methods, replaces some uses of Abstract Class

```
interface TreeNode<K>

  List<K> getChildren();

  default boolean hasChildren() {

     if (getChildren().isEmpty()) {
        return false;
     }

     return true;
  }
```

Can use:

- Other interface methods
- **this**

Default Methods

Found 64 default methods in SPeCS repository

When to use Abstract Class now?

- When we need private state (interfaces remain stateless)

A extends *I1* and *I2*, both have method *m* and there is a default method

- Compile-time error, *A* needs to implement *m*

Default methods might obscure class hierarchy

- Keeps class hierarchy simple, favor composition over inheritance…

Lambdas

- A terser way to write anonymous classes.

```
Runnable r = new Runnable() {

  @Override
  public void run() {
    System.out.println("HELLO");
  }

};
```

Lambdas

Found ~100 lambdas in SPeCS repository

- A terser way to write anonymous classes.

```
Runnable r = new Runnable() {

  @Override
  public void run() {
    System.out.println("HELLO");
  }

};
```

```
Runnable r = () -> {
    System.out.println("HELLO");
};
```

OR

```
Runnable r = () -> System.out.println("HELLO");
```

Lambdas

- Lambda == Passing a ‘function’ as an argument

```
class ProcessUtils {

  public static void getNanoTime(Runnable runnable) {

  …
```

```
ProcessUtils.getNanoTime( () -> System.out.println("HELLO") );
```

Anatomy of a (Java) Lambda

- Functional Interface: A Java interface that only has one function left to implement
  - Runnable, ActionListener, Comparator, Callable…
  - Default methods help here

```
public interface Check {

   boolean check(ProviderData data);
}
```

(Functional Interface)

Anatomy of a (Java) Lambda

```
// Returns true if the number of inputs in ProviderData is the same as the given number
public Check numberOfInputs(int numberOfInputs) {

    return (ProviderData data) ->
    {

       return data.getInputTypes().size() == numberOfInputs;
    };

}
```

Function left to implement in the Functional Interface (e.g., check() in Check)

Anatomy of a (Java) Lambda

```
// Returns true if the number of inputs in ProviderData is the same as the given number
public Check numberOfInputs(int numberOfInputs) {

    return (ProviderData data) ->
    {

        return data.getInputTypes().size() == numberOfInputs;
    };

}
```

Arguments of the function

- types are optional

Anatomy of a (Java) Lambda

```
// Returns true if the number of inputs in ProviderData is the same as the given number
public Check numberOfInputs(int numberOfInputs) {

    return (data) ->
    {
        return data.getInputTypes().size() == numberOfInputs;
    };

}
```

Body of the function

- If only one line, *{}* and *return* are optional

Anatomy of a (Java) Lambda

```
// Returns true if the number of inputs in ProviderData is the same as the given number
public Check numberOfInputs(int numberOfInputs) {

    return (data) -> data.getInputTypes().size() == numberOfInputs;

}
```

We can use the input arguments, and any variable in scope that is *effectively final.*

- "A variable or parameter whose value is never changed after it is initialized is effectively final."
- In practice, the same as final, but without the need for the keyword

Lambda Example

```
public CNativeType newNumeric(Number number) {
        Class<?> numberClass = number.getClass();

        ...
```

Lambda Example

```
public CNativeType newNumeric(Number number) {
        Class<?> numberClass = number.getClass();

        if (numberClass.equals(Character.class)) {
           return newChar();
        }

        if (numberClass.equals(Byte.class)) {
           return newChar();
        }

        if (numberClass.equals(Short.class)) {
           return newShort();
        }

        ...
```

Lambda Example

```
public CNativeType newNumeric(Number number) {

        // Get type from table
        NProvider provider = converter.get(numberClass);
        …
```

```
private Map<Class<?>, NProvider> initConverter() {
        Map<Class<?>, NProvider> converter = new HashMap<>();

        converter.put(Character.class, () -> newChar());
        converter.put(Byte.class, () -> newChar());
        converter.put(Short.class, () -> newShort());
        …
```

ArrayList vs LinkedList

In a class hierarchy, good practice to use most abstract class possible

- E.g., List instead of ArrayList

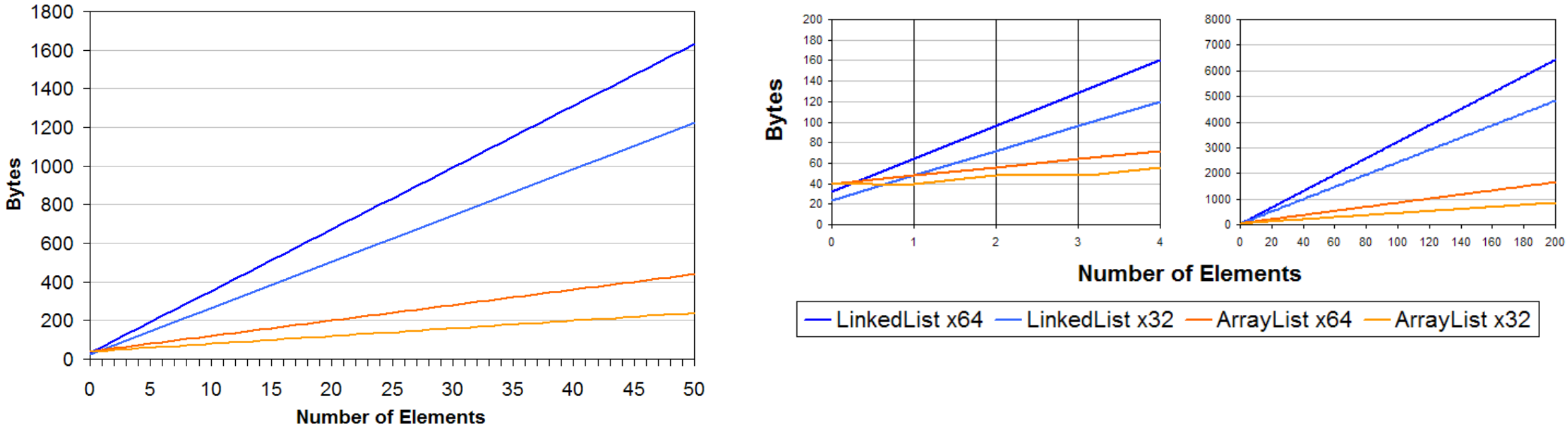
However, implementation can have a real impact in performance

ArrayList vs LinkedList

In a class hierarchy, good practice to use most abstract class possible

- E.g., List instead of ArrayList

However, implementation can have a real impact in performance



(source: http://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist/7671021#7671021)

ArrayList vs LinkedList

If LinkedList uses more memory than ArrayList, why use LinkedList?

- As a rule of thumb, ArrayList is what you should use

ArrayList vs LinkedList

However, different asymptotic complexities when using the list

- LinkedList ideal when adding/removing elements while iterating
- Or adding elements to the head of the list

Operation	ArrayList Complexity	LinkedList Complexity
get(int index)	O(1)	O(n)
add(E element)	O(1) amortized, O(N) worst	O(1)
add(int index, E element)	O(n - index) amortized	O(n)
remove(int index)	O(n - index)	O(n)
Iterator.remove()	O(n - index)	O(1)
ListIterator.add(E element)	O(n - index)	O(1)

ArrayList vs LinkedList

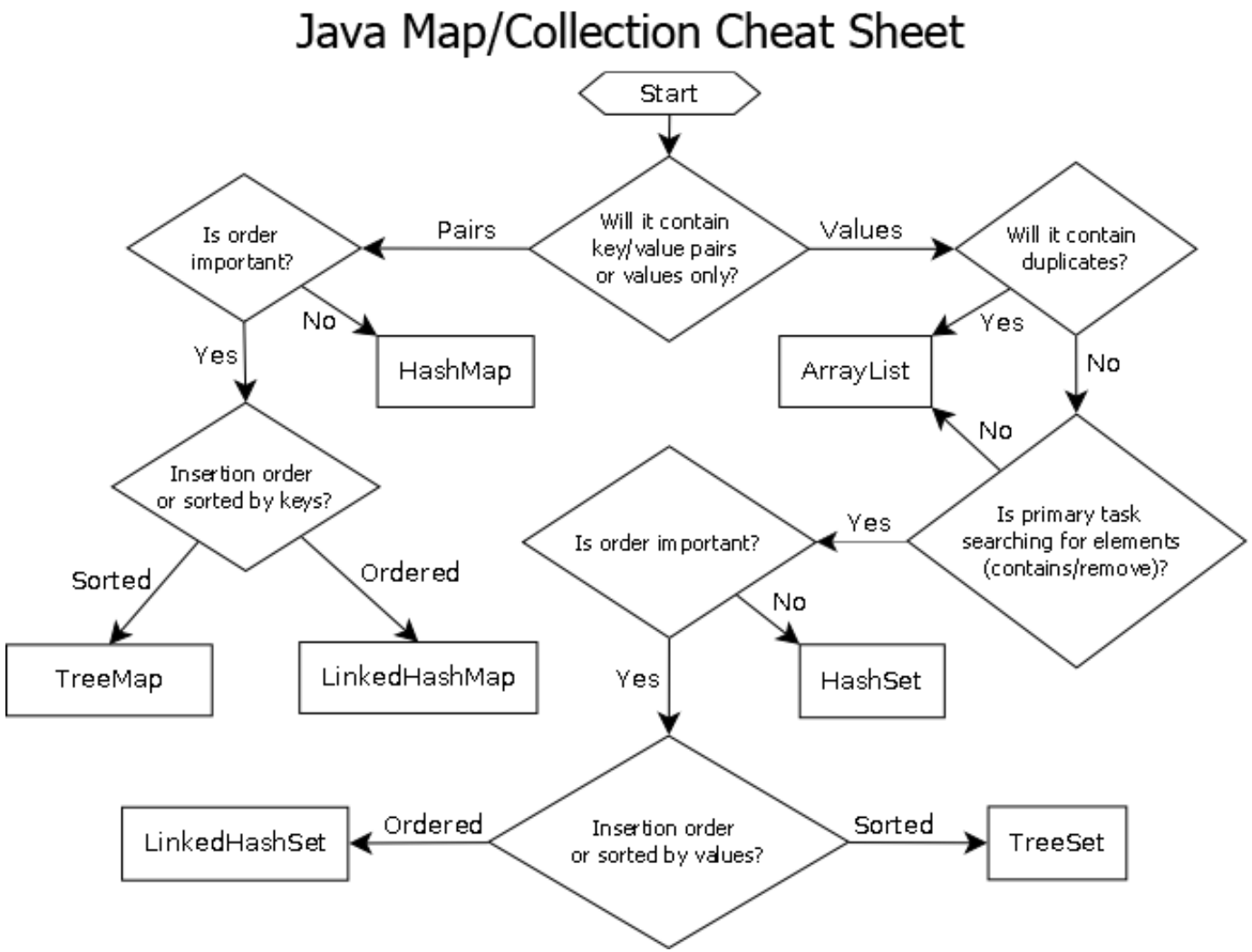
Megablock Detector uses a Pushing Queue

Example: PushingQueue

- Elements can only be added at the head of the queue.
- Every time an element is added, every other elements gets “pushed”
- If an element is added when the queue is full, the last element in the queue is dropped.

	Insert (ms)		
Elements	**ArrayList**	**LinkedList**	**LL Speedup**
100	0.24	0.75	0.32
1000	0.89	2.24	0.40
10000	65.54	4.32	15.17
100000	6,730.00	68.38	98.42

Java Collections



(source: http://www.sergiy.ca/guide-to-selecting-appropriate-map-collection-in-java/)