Benefits of Saving Memory

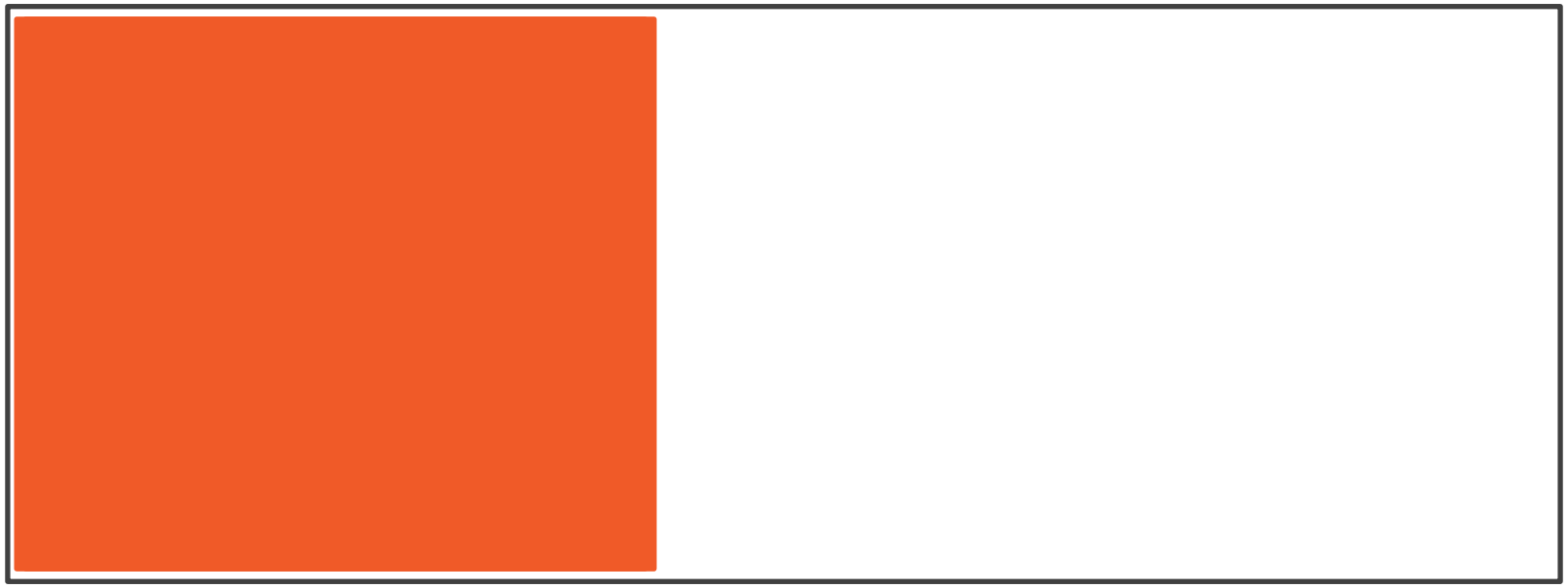
Reduced GC activity

More instructions and data can fit in RAM and in the CPU caches

Improved performance from having less data to move around in memory

Better use of hardware

Memory Pressure



Garbage Collection Effects

GC pauses

Background GC thread activity

Benefits of Saving Memory

Reduced GC activity

More instructions and data can fit in RAM and in the CPU caches

Improved performance from having less data to move around in memory

Better use of hardware

Java Types

Primitive Types

Object Types

Primitive Type Sizes

1 byte: boolean, byte

2 bytes: char, short

4 bytes: int, float

8 bytes: long, double

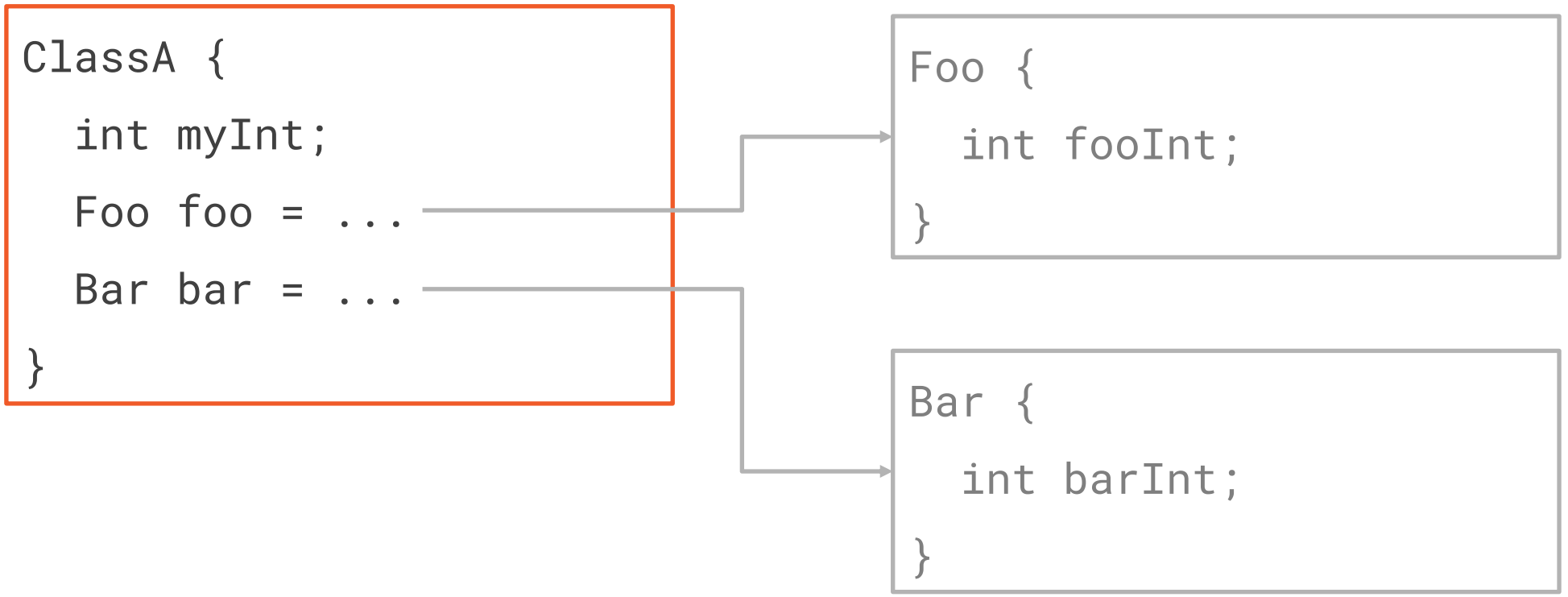
Object Type Sizing

Shallow size

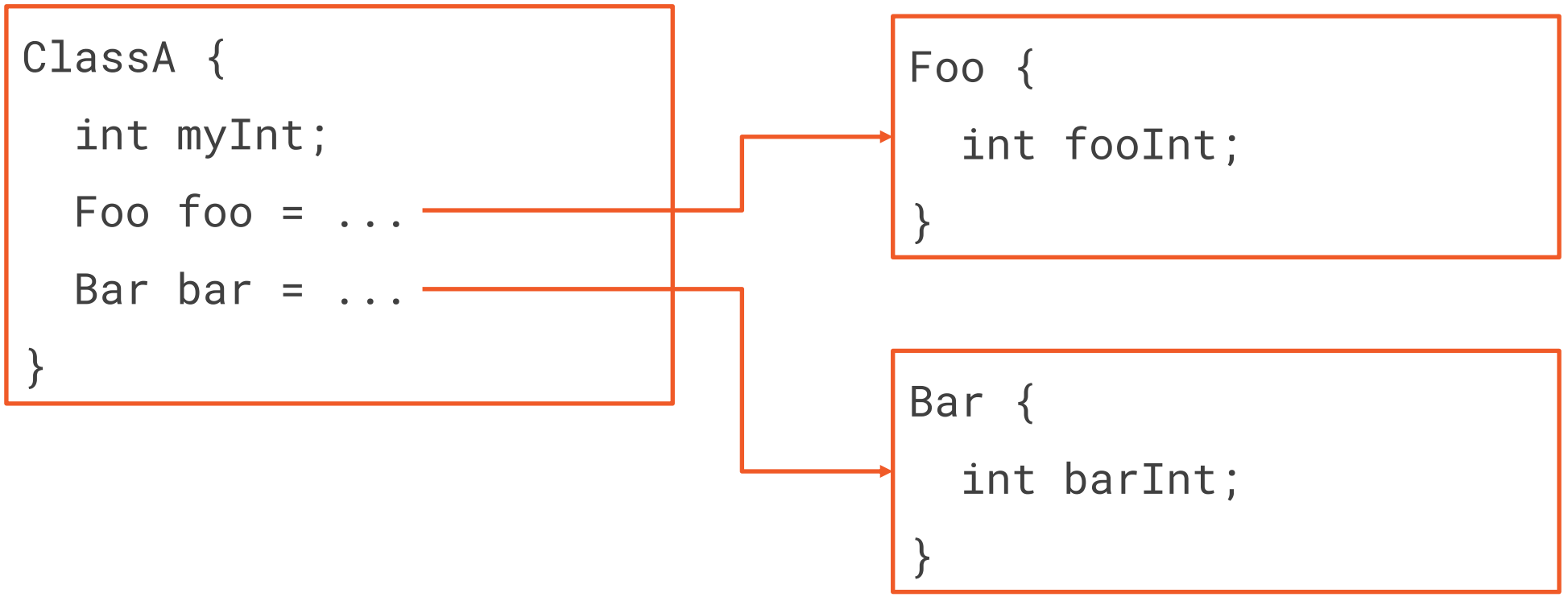
Deep size

Retained size

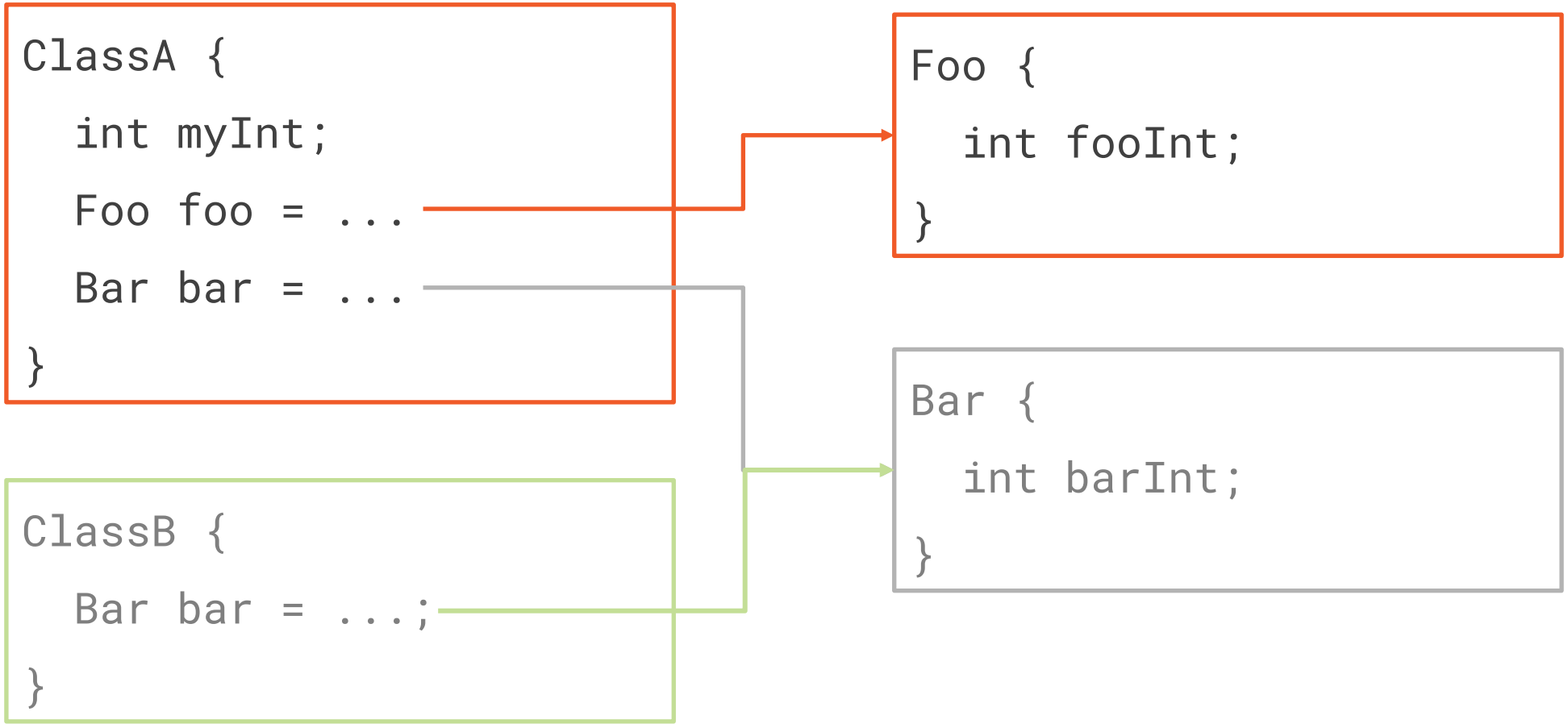
Shallow Size



Deep Size


ClassA {
int myInt;
Foo foo = ...
Bar bar = ...
}
Foo {
int fooInt;
}
Bar {
int barInt;
}

Retained Size



```
MyClass {
  int myInt;
  short myShort;
  boolean myBool;
  MyOtherClass otherClass;
}
```

- +12 bytes
- +4 bytes
- +2 bytes
- +1 bytes
- +4 / +8 bytes

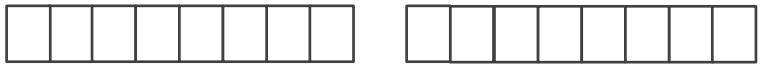
Java Memory Alignment

```
ClassA {

}
```

8 bytes 8 bytes

```
ClassB {
  int myInt;
}
```



8 bytes 8 bytes

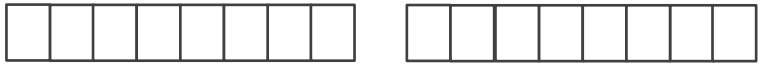
Java Memory Alignment

```
ClassA {

}
```

8 bytes 8 bytes

```
ClassB {
  int myInt;
}
```



8 bytes 8 bytes

Java Memory Alignment

```
ClassA {

}
```

8 bytes 8 bytes

```
ClassB {
  int myInt;
}
```

8 bytes 8 bytes

Java Memory Alignment

```
ClassA {

}
```

8 bytes 8 bytes

```
ClassB {
  int myInt;
  String myStr = ...; (+ 96 bytes)
}
```

8 bytes 8 bytes 8 bytes

Managing object sizes is an important consideration in saving memory

Overview

- Reducing Object Size
- Avoid Creating Unnecessary Objects
- Managing Strings
- Avoid Keeping Objects Around for Longer Than Needed

Reducing Object Size

```
public class Employee {
  String name;
  String department;
  int age;
  int status;
  List<String> previousJobs = new ArrayList<>();
}
```

```
public class Employee {
  String name;
  String department;
  int age;
  int status;
  List<String> previousJobs = new ArrayList<>();
}
```

```
public class Employee {
  String name;
  String department;
  int age;
  int status;
  List<String> previousJobs = new ArrayList<>(10);
}
```

Primitive Data Type Ranges

Data Type	Range
byte	-127 to 128
short	-32,768 to 32,767
int	-2,147,483,648 to 2,147,483,647
long	$-9.22 * 10^{18}$ to $9.22 * 10^{18}$

```
public class Employee {
  String name;
  String department;
  int age;
  int status;
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  int status;
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  int status;
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
}
```

```
public class Employee {
  String name;
  String department;
  int age;
  int status;
  List<String> previousJobs =
     new ArrayList<>();
}
```

(48 bytes)

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
}
```

(24 bytes)

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  Date startDate; // 24 bytes
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  LocalDate startDate; // 32 bytes
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  long startDateMillis; // 8 bytes
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  long startDateMillis; // 8 bytes
  long getDaysSinceStart() {
    LocalDate startDate = Instant.ofEpochMilli(startDateMillis)
      .atZone(ZoneId.systemDefault())
      .toLocalDate();
    return ChronoUnit.DAYS.between(startDate, LocalDate.now());
  }
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  int startDate; // e.g. 20091101 (4 bytes)
}
```

BigDecimal

Solves rounding issues when dealing with floating point numbers

32 bytes (shallow size only)

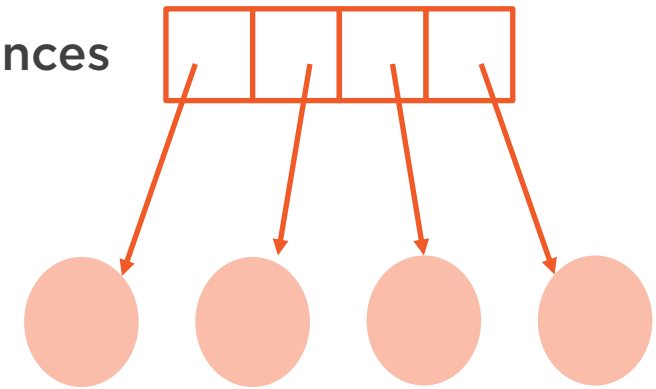
Alternative: float (4 bytes) / double (8 bytes) + Math.round()

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  long currentSalaryInCents;
}
```

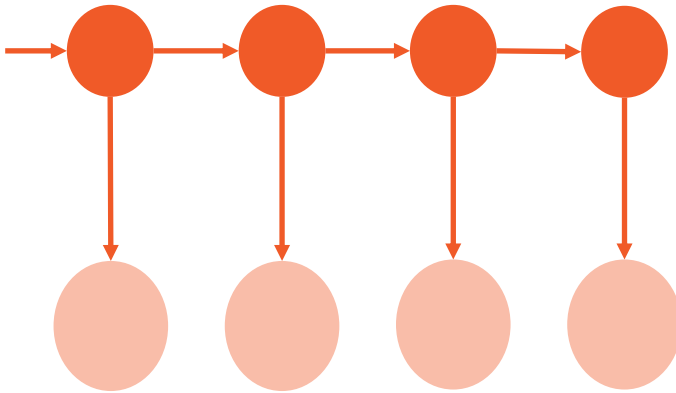
Prefer Primitive Types over Object Types

ArrayList

4 byte references



LinkedList



24 byte nodes

Object Versions of Primitive Types

When a collections class is used

When the variable can have a valid unset state

Object Versions of Primitive Types

When a collections class is used

- Alternative collections library like FastUtil or Trove

When the variable can have a valid unset state

Object Versions of Primitive Types

When a collections class is used

- Alternative collections library like FastUtil or Trove

When the variable can have a valid unset state

- Use a sentinel value (e.g. -1)

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  long currentSalaryInCents;
  boolean[] accessRights;
}
```

```
public class Employee {
  String name;
  String department;
  short age;
  byte status;
  long currentSalaryInCents;
  BitSet accessRights;
}
```

Avoid Creating Unnecessary Objects

```
public class MyClass {
  Result performAction(Param param) {
     ...
     if (failureCondition) {
       return null;
     }
  }
}
```

```
public class MyClass {
  Result performAction(Param param) {
     ...
     if (failureCondition) {
       return new Result();
     }
  }
}
```

```
public class MyClass {
  private static final Result EMPTY_RESULT = new Result();
  Result performAction(Param param) {
     ...
     if (failureCondition) {
       return EMPTY_RESULT;
     }
  }
}
```

```
public class MyClass {
  List<Result> performAction(Param param) {
     ...
     if (failureCondition) {
       return new ArrayList<>();
     }
  }
}
```

```
public class MyClass {
  private static final List<Result> EMPTY_RESULTS =
   new ArrayList<Result>();
  Result performAction(Param param) {
     ...
     if (failureCondition) {
       return EMPTY_RESULTS;
     }
  }
}
```

```
public class MyClass {
  Result performAction(Param param) {
     ...
     if (failureCondition) {
       return Collections.emptyList();
     }
  }
}
```

Efficient Collections Factory Methods

Collections.singletonList()

Collections.singletonMap()

Collections.emptyList()

Collections.emptyMap()

Collections.emptySet()

ArrayList or HashMap with the Default Constructor

ArrayList

HashMap

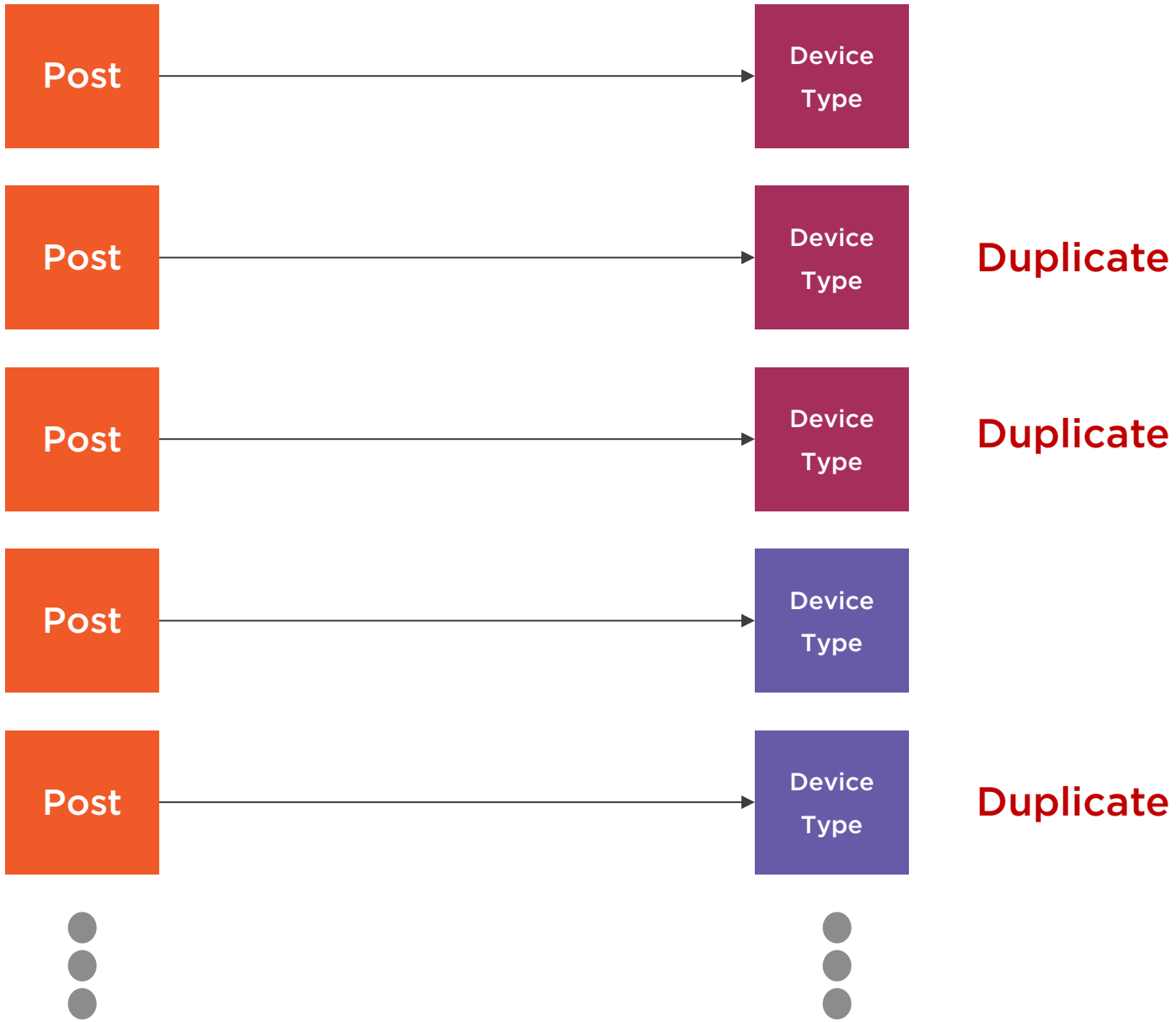
ArrayList or HashMap with the Default Constructor

ArrayList

HashMap

```
public class Post {
  private String author;
  private String body;
  private ZonedDateTime postDatetime;
  private DeviceType deviceType;
  ...
}
```

```
public class Post {
  private String author;
  private String body;
  private ZonedDateTime postDatetime;
  private DeviceType deviceType;
  ...
}
```


Post
Device Type
Post
Device Type
Duplicate
Post
Device Type
Duplicate
Post
Device Type
Post
Device Type
Duplicate

Interning

Method of storing only one copy of a distinct object

Objects are compared using their equals() methods and duplicates are discarded

Should only be used for immutable objects

Interner Libraries

Guava

Has strong hashmap and weak hashmap implementations

triava

Faster than Guava

Lazy Initialization

The tactic of delaying the creation of an expensive object until the first time it is needed

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection();

  public void doSomething() {
    ...
  }
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection();

  public void doSomething() {
    ...
  }
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection();

  public void doSomething() {
    ...
  }
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection();

  public void doSomething() {
    ...
  }
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection();

  public void doSomething() {
    ...
  }
  // only called 10% of the time
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

```
public class Session {
  private Connection conn =
     ConnectionFactory.newConnection(); // unused 90% of the time

  public void doSomething() {
    ...
  }
  // only called 10% of the time
  public void writeSomething(Object thing) {
    conn.write(thing);
  }
}
```

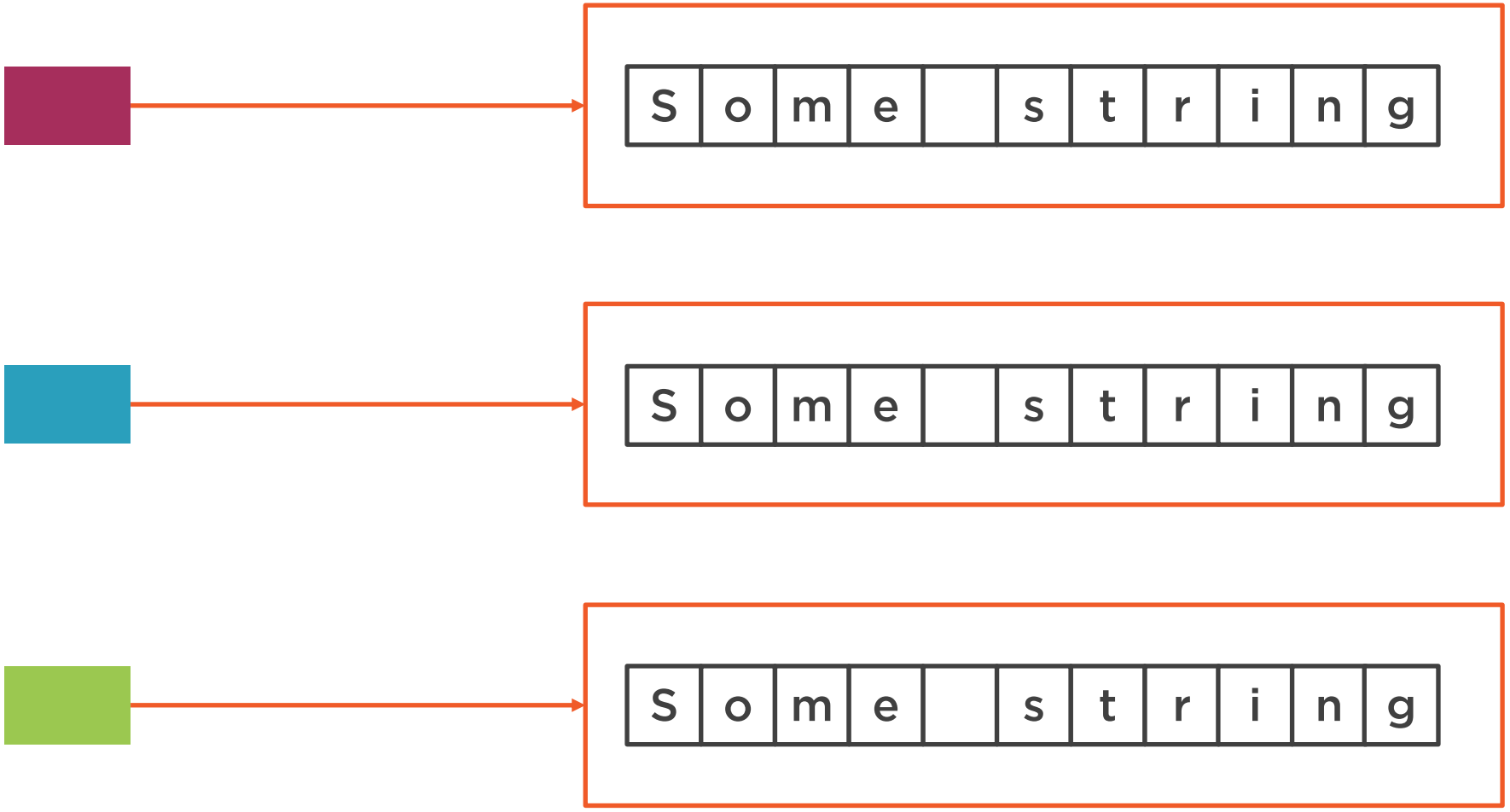
```
public class Session {
  private Connection conn;
  private Connection getConnection () {
    if (conn == null) {
      conn = ConnectionFactory.newConnection();
      return conn;
    }
  }
  public void doSomething() { ... }
  public void writeSomething(Object thing) {
    getConnection().write(thing);
  }
}
```

```
public class Session {
  private Connection conn;
  private Connection getConnection () {
    if (conn == null) {
      conn = ConnectionFactory.newConnection();
      return conn;
    }
  }
  public void doSomething() { ... }
  public void writeSomething(Object thing) {
    getConnection().write(thing);
  }
}
```

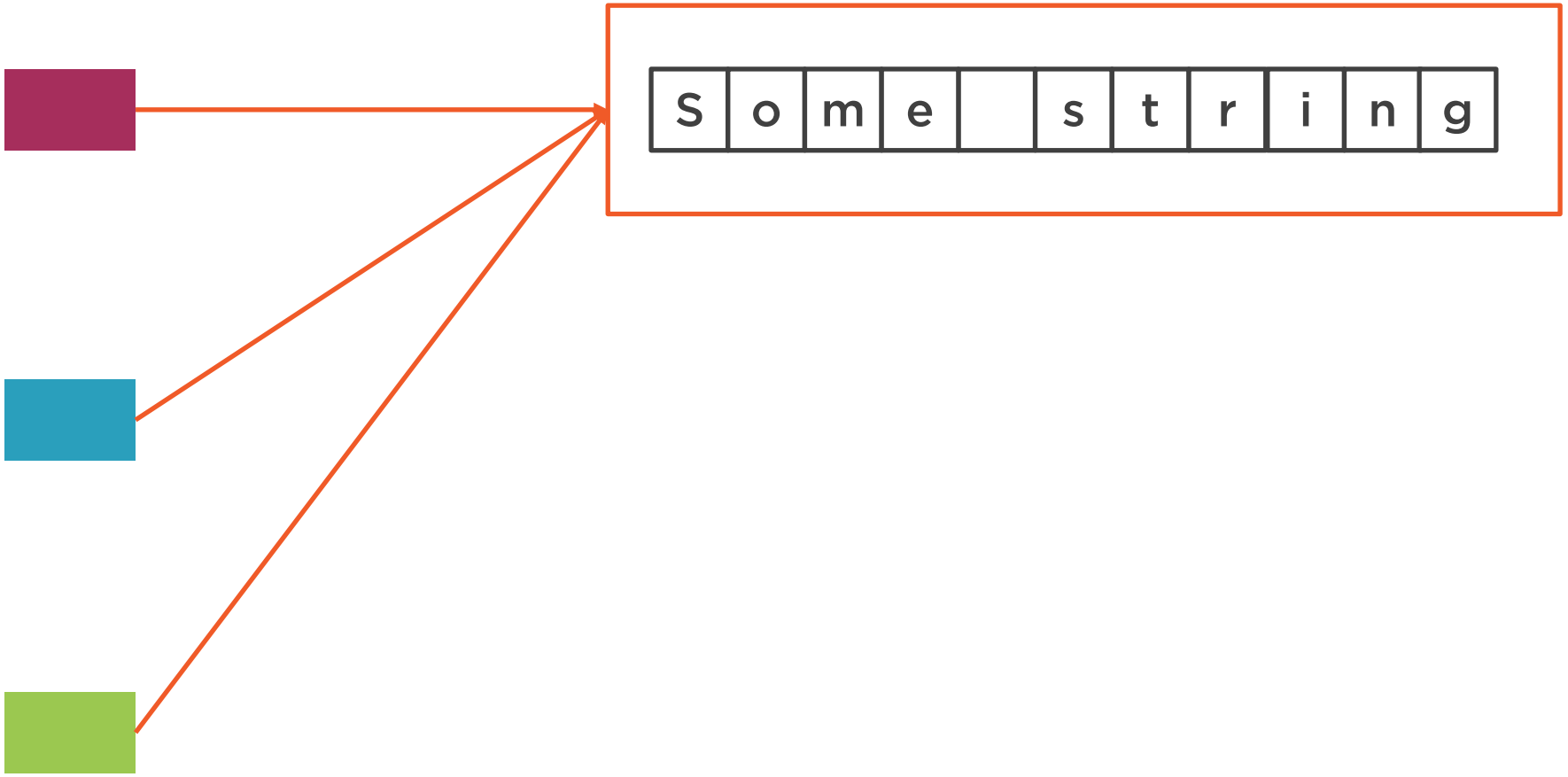
```
public class Session {
  private Connection conn;
  private synchronized Connection getConnection () {
    if (conn == null) {
      conn = ConnectionFactory.newConnection();
      return conn;
    }
  }
  public void doSomething() { ... }
  public void writeSomething(Object thing) {
    getConnection().write(thing);
  }
}
```

Managing Strings

String Duplication

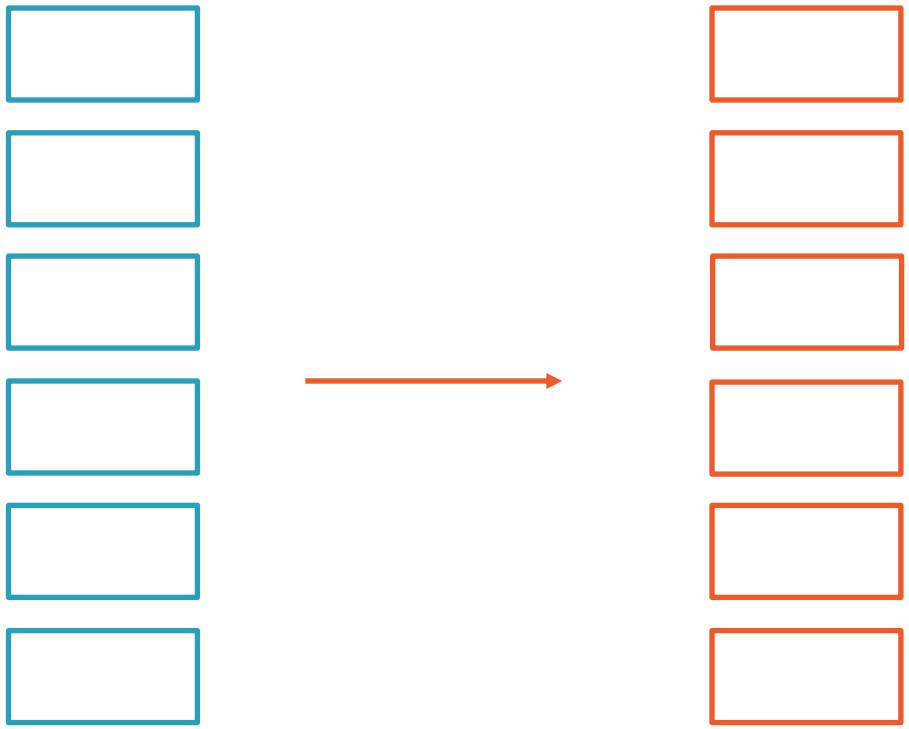


String Duplication



```
public class Post {
  private String author;
  private String body;
  private ZonedDateTime postDatetime;
  private DeviceType deviceType;
  private String language;
  ...
}
```

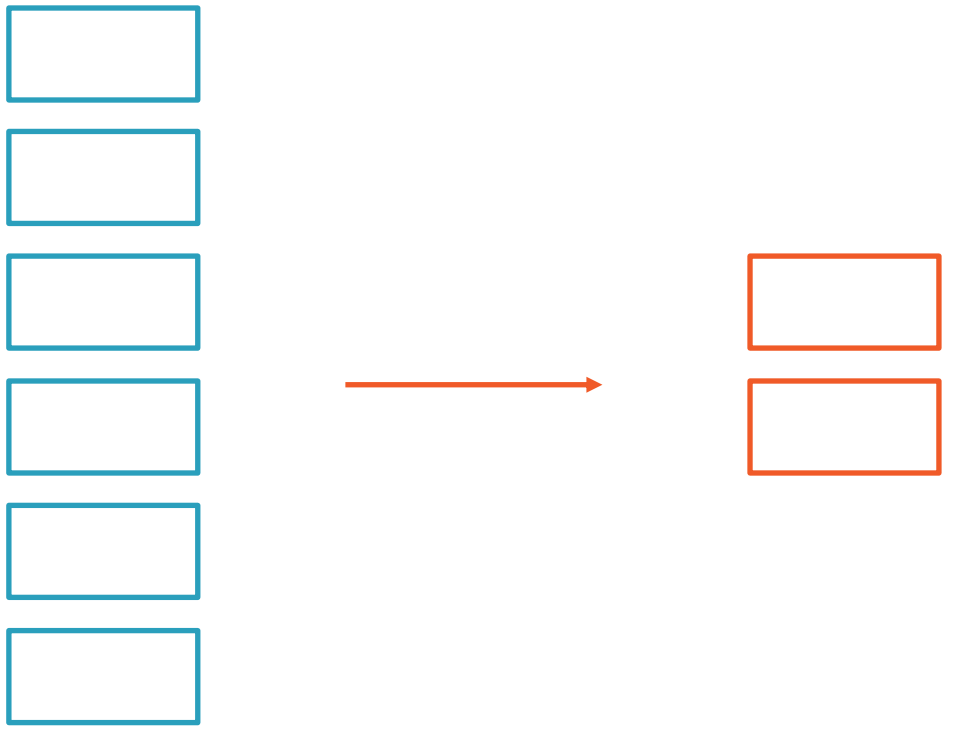
Without Interning



1 M pointers

1 M string objects

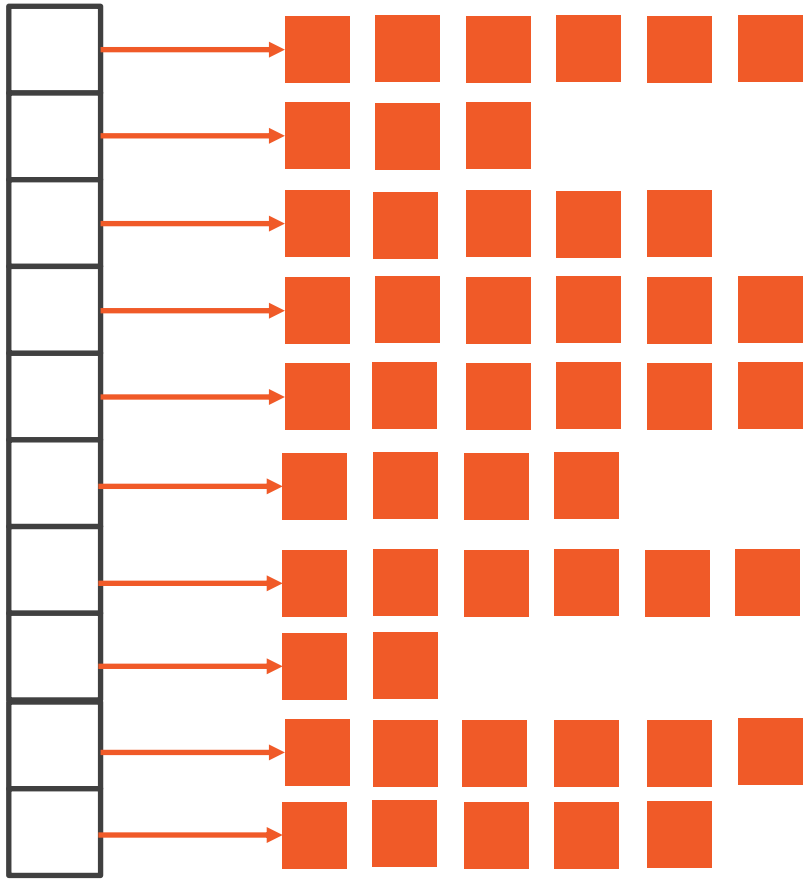
With Interning



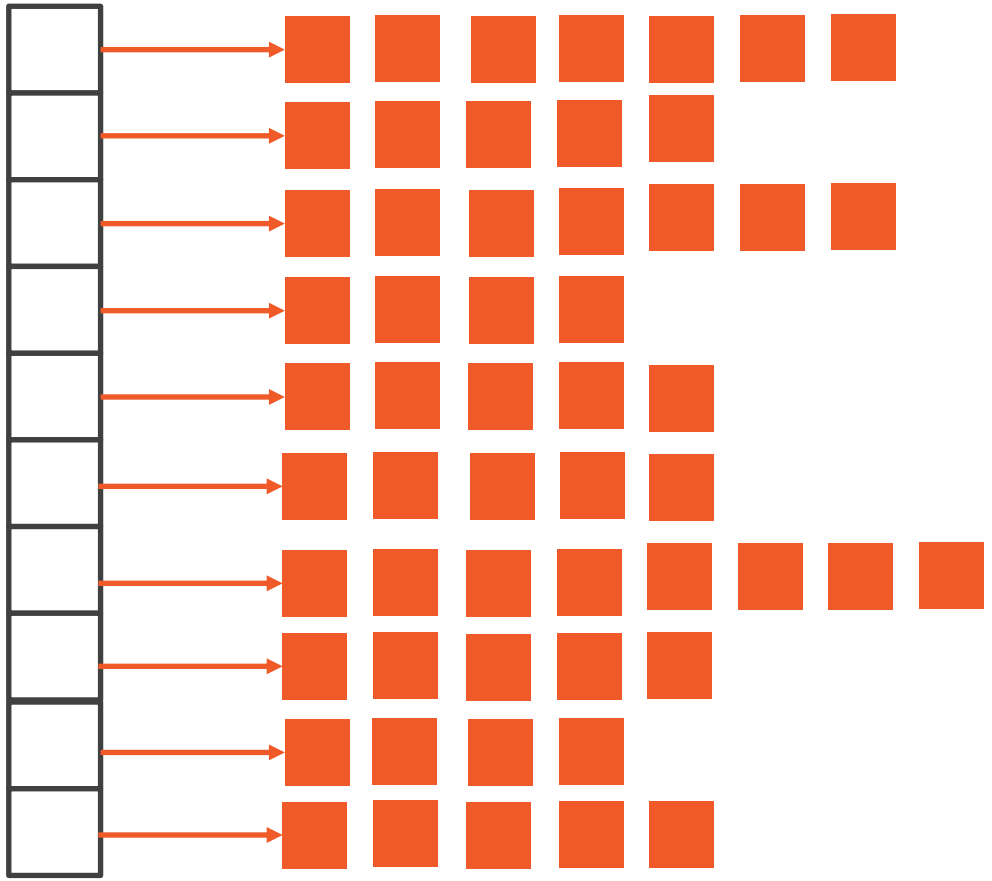
1 M pointers

20 string objects

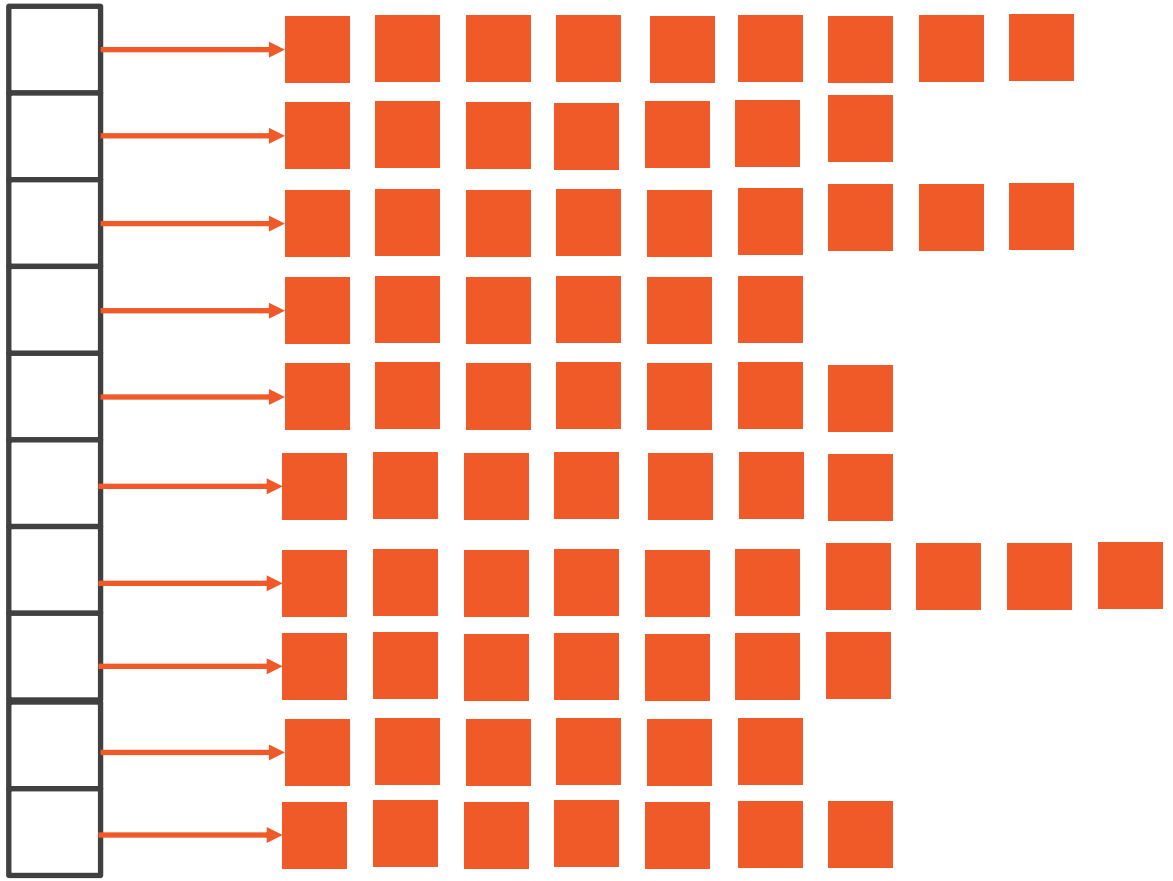
String.intern()



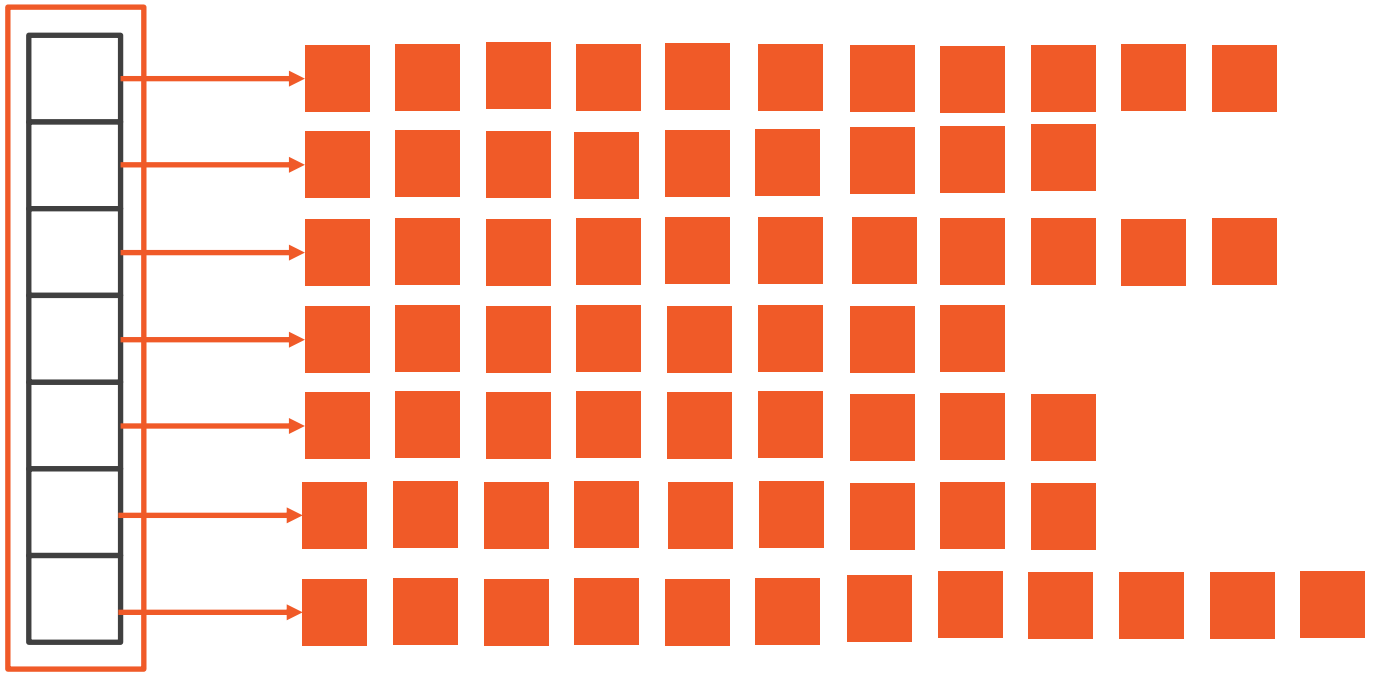
String.intern()



String.intern()



String.intern()



```
java -XX:+PrintFlagsFinal -version | grep StringTableSize
```

Viewing the Size of the String Table

```
java -XX:StringTableSize=1000013
```

Setting the Size of the String Table

String Interning

Java automatically interns string literals and string valued constant expressions

Alternative to built-in Java String interner: triava or Guava interner

String Deduplication

Profiling tools like the Yourkit Java Profiler or Eclipse MAT can help find duplicate strings

The string deduplication feature was added to the G1 GC in Java 8u20

Deduplication Thread

Analyzes all strings being moved between heap regions and that meet a certain condition

Uses the hashcode of the string, the equals method, and a deduplication hash table to see if a string needs to be deduplicated

Only runs when there are available CPU cycles

```
java -XX:+UseStringDeduplication
```

Enable String Deduplication

When to Use String Deduplication

String deduplication should not take the place of string interning

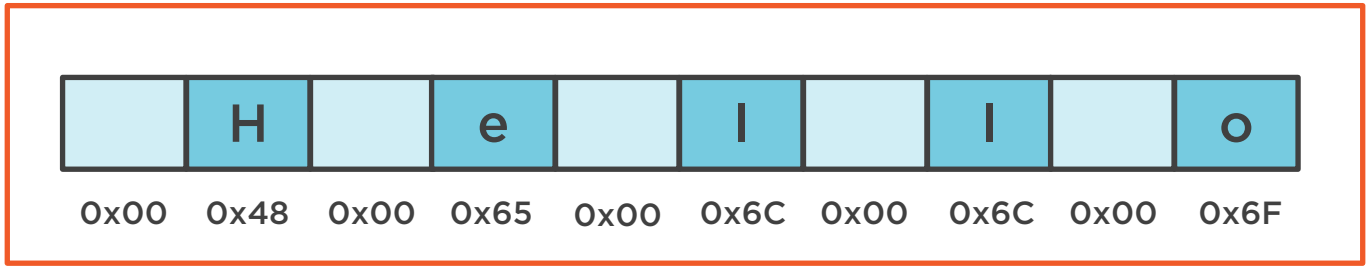
Use if your application produces lots of strings and you have available CPU

View effect of string deduplication:

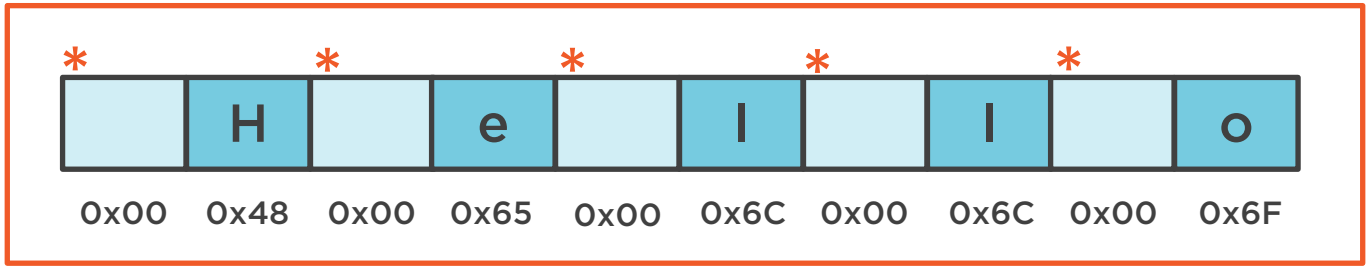
```
-XX:+PrintStringDeduplicationStatistics
```

```
-Xlog:stringdedup*=debug
```

Compact Strings

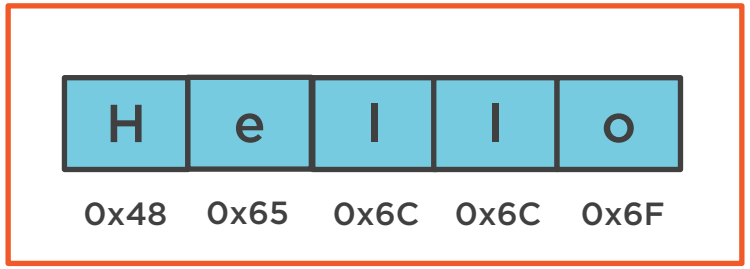


Compact Strings



* Wasted space when only Latin-1 characters are used in a string

Compact Strings



JEP 250

Store interned strings in class-data sharing (CDS) archives

Avoid Keeping Objects Around for Longer Than Needed

```
public class MyClass {
  private static Baz baz = new Baz();
  private Bar bar = new Bar();
  public Result doSomething() {
    Foo foo = new Foo();
    Result result = foo.calculate();
    return result;
  }
}
```

```
public class MyClass {
  private static Baz baz = new Baz();
  private Bar bar = new Bar();
  public Result doSomething() {
    Foo foo = new Foo();
    Result result = foo.calculate();
    return result;
  }
}
```

```
public class MyClass {
  private static Baz baz = new Baz();
  private Bar bar = new Bar();
  public Result doSomething() {
    Foo foo = new Foo();
    Result result = foo.calculate();
    return result;
  }
}
```

```
public class MyClass {
  private static Baz baz = new Baz();
  private Bar bar = new Bar();
  public Result doSomething() {
    Foo foo = new Foo();
    Result result = foo.calculate();
    return result;
  }
}
```

```
public class MyClass {
  private static Baz baz = new Baz();
  private Bar bar = new Bar();
  public Result doSomething() {
    Foo foo = new Foo();
    Result result = foo.calculate();
    return result;
  }
}
```

Local Scope Optimization

Objects that are in local scope can be optimized by the JIT compiler

Escape analysis

Compiler will deconstruct an object into it's fields and put the fields on the stack instead of the heap

```
public class Service {
  private ConcurrentMap cache = new ConcurrentHashMap();

  public Result calculate(Request request) {
    Result result = cache.computeIfAbsent(request, rqst ->
      doCalculate(rsqst));
    return result;
  }
}
```

```
public class Service {
  private ConcurrentMap cache = new ConcurrentHashMap();

  public Result calculate(Request request) {
    Result result = cache.computeIfAbsent(request, rqst ->
      doCalculate(rsqst));
    return result;
  }
}
```

```
public class Service {
  private ConcurrentMap cache = new ConcurrentHashMap();

  public Result calculate(Request request) {
    Result result = cache.computeIfAbsent(request, rqst ->
      doCalculate(rsqst));
    return result;
  }
}
```

Memory Leaks

Can occur with poorly implemented caches or other situations where references are allowed to linger

Consider using weak or soft references

Watch Pluralsight course: Understanding and Solving Java Memory Problems