



IBM Systems & Technology Group

Java Performance Analysis

Mario J Lorenzo

IBM Electronic Service Agent

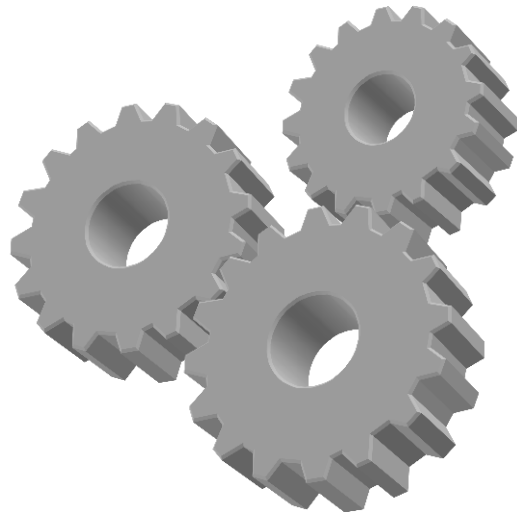
ON DEMAND BUSINESS™

Introduction

- **Many of the costly issues discovered after delivery of a system may have been remedied in design or implementation if they were detected early in the development cycle.**
- **Just like any human illness, early detection is the key to a healthy and functional system that, more often than not, averts the dangers and cost of surgery.**
- **By integrating performance and scalability analysis within the development process these issues can be detected and corrected.**

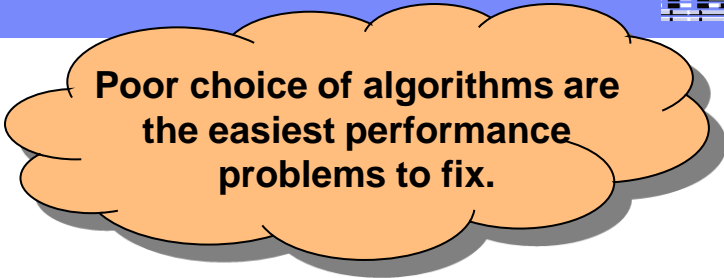
Overview

- **This presentation will discuss various techniques that help identify and solve many of the most common Java performance issues found today.**
- **This presentation will introduce and discuss the following topics:**
 - Excessive object creation
 - Importance of Interface design
 - Avoiding re-computation
 - Interchange types
 - Using thread pooling to avoid excessive thread creation
 - Remote Interface Design
 - Immutable and Mutable Object traps
 - Using Weak Listeners
 - Tips on Servlet & JSP
- **Each topic will contain an introduction, examples, and design techniques that can be used to increase performance and scalability.**



Interface Design

Interface Design Introduction



Poor choice of algorithms are the easiest performance problems to fix.

- **Poor interface design is the seed to many performance problems.**
- **Interfaces implemented by components can have a significant effect on the behavior and performance of the programs that use them.**
- **So how do interfaces effect performance?**

How interface design impacts performance?

- A class's interface defines what operations the class can perform.
- It can also define its **object-creational behavior** and the sequence of method calls required to use it.
- A class's constructors and methods will dictate whether an object can be reused.
- Whether its methods will create (or require a caller) to create intermediate objects.
- How many calls needed in order to use a given class.



All of these factors affect performance

Avoid Excessive object creation



- Look out for object creation inside loops
- Avoid unnecessarily creating temporary or intermediate objects
- **String (and Immutable) classes aka String Trap**
 - Immutable: new object must be created each time it is modified or constructed.
 - Major source of object creation



- Tip: use StringBuffer to avoid String Trap (String Builder for Java 5 and above)
- Sometimes impossible to avoid
 - When using interfaces that are defined only in terms of Strings

Example

■ A Computational Biology application

- Uses pattern matching to detect recurring strings in a DNA sequence.
 - Ex. ACGTCCT or ACGTCCT
- The application will reuse a helper class (ExpMatcher) that will aid in the matching of a given pattern within a sequence.
- Since this app is meant to be fast and efficient we stay away from using String and instead we use a character buffers.

Example

■ Very bad interface

```
public class ExpMatcher {  
    public ExpMatcher(String regExp, String inputText) {...}  
    public String getNextMatch() {...}  
}
```

■ Client

```
while(...) {  
    String arg = new String(charBuffer, start, end);  
    ExpMatcher matcher = new ExpMatcher(regExp, arg);  
    String result = matcher.getNextMatch();  
    If(result != null) {...}  
}
```

- Can't use ExpMatcher more than once since its tied to the input text
- This means you have to construct a new ExpMatcher every time.

Example

■ Bad interface:

```
public class ExpMatcher {  
    public ExpMatcher(String regExp){...}  
    public String match(String inputText) {...}  
    public String getNextMatch() {...}  
}
```

■ Client

```
ExpMatcher matcher = new ExpMatcher(someRegExp);  
while(...) {  
    String arg = new String(charBuffer, start, end);  
    String result = matcher.match(arg);  
    if(result != null) {...}
```

- Requires caller to create a String to represent the text to be matched
- Returns a String even if our app doesn't care about the result
- Our goal of avoiding String is lost

Example

■ Better interface

```
public class ExpMatcher {  
    public ExpMatcher(String regExp){...}  
    public int match(String inputText) {...}  
    public int match(char[] inputText) {...}  
    public int match(char[] inputText, int offSet, int length) {...}  
    public int getNextMatch() {...}  
    public int getMatchLength() {...}  
    public String getMatchText() {...}  
}
```

■ Client

```
ExpMatcher matcher = new ExpMatcher(someRegExp);  
//...  
int offset = matcher.match(charBuffer, start, end);  
if (offset > 0) {...}
```

Example Conclusion: what did we learn?

- **The Strings used in the examples were just to exchange data**



–this is called an [interchange type](#)

- When caller nor callee is likely to actually want the data in a given format.
- ex. JDBC [ResultSet](#)

- **The interface forced the caller to use Strings**

- **Performance takes a hit due to the set-up time to make the call and recovery time after the call.**

- **Unfortunately these types of issues cannot be resolved quickly (like last minute fixes).**

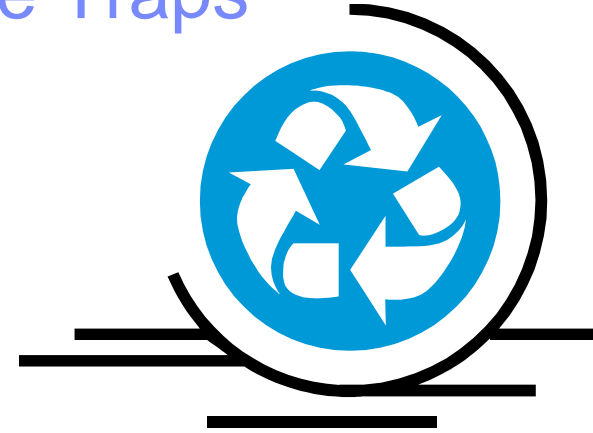
–**Why?** Because it is difficult to change a class' interface

- **Spend extra time during design phase considering the performance impact of your class interfaces!**



Java Techniques to Avoid Performance Traps

- **Caching**
 - avoid re-computation
- **Thread pooling**
 - avoid costly thread creation
- **Flyweight pattern**
 - separate objects intrinsic state from extrinsic state
- **Weak references**
 - Use Weak data structures so that listeners that are no longer used can be cleaned up by GC
- **Interface Designs**
 - Accept input in various types (avoid temporary object creation)
 - Avoid object creation and Interchange Types
 - Provide finer-grained accessor functions
 - Exploit mutability – provide method that accept mutable objects to pass in results
- **Remote Interface Design**
 - Opposite approach to above cited Interface design tips
 - Avoid constant remote method invocation by providing methods that retrieve several items simultaneously
 - Avoid returning remote objects when caller does not need to hold reference to remote object
 - Avoid passing complex remote objects to remote methods when no copy is needed.



Look out for
Recomputation!
Use caching to
remedy!

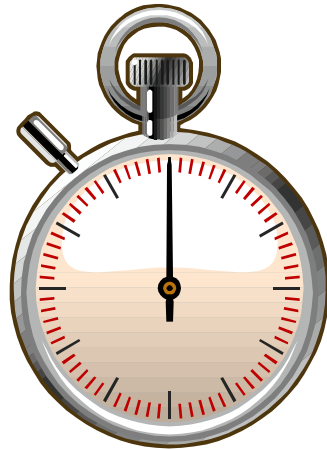
Java Performance Traps

- **Strings (Immutable objects)**
- **Thread creation**
- **Interface design**
- **Poor algorithm**
- **Excessive Object creation**
- **Remote Interface design**
- **Interchange types**
- **Mutable objects**
- **Listeners**
- **Servlets & JSP**

The End

- **Performance analysis should be an integral part of your design and development cycles!**

A little



now saves a lot of



later!