# Java Code Optimization

| | |
|---|---|
| *Course* | **Java Programming** |
| *Trainer* | **Tuan Le** |
| *Designed by* | **TTC** |
| *Last updated* | **6-Nov-14** |

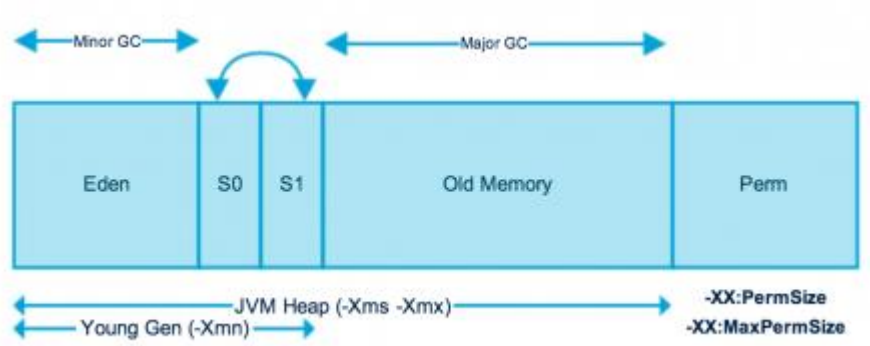
# Contents

- Memory management & garbage collection
- Effective Java
- Java Optimization Tips

# Objectives

- The course helps attendees to obtain some techniques that helps improve performance in java program.

# Java (JVM) Memory Model




- JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

# Java (JVM) Memory Model

- Young Generation Spaces:
  - Most of the newly created objects are located in the Eden memory space.
  - When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
  - Objects that are survived after many cycles of GC, are moved to the Old generation memory space.
- Old Generation
  - Contains the objects that are long lived and survived after many rounds of Minor GC.
  - Garbage collection is performed in Old Generation memory when it's full.
  - Old Generation Garbage Collection is called **Major GC** and usually takes longer time.
- Permanent Generation
  - The method area is implemented as a separated part.

# Java memory management

- Objects are created on heap in Java irrespective of their scope e.g. local or member variable
- Class variables or static members are created in method area and both heap and method area is shared between different thread.
- Java lets you allocate objects as necessary and trust that they'll be reclaimed and recycled by the JVM
- The basic principle of garbage collection is the same in all cases:
  - Identify objects that are no longer in use by the program.
  - Recycle the memory used by these objects to create new ones.

# Java memory management

- Define Java heap space for your application
  - In Eclipse IDE, if your program is consuming a lot of memory (loading big data) like this :

```
List<Domain> list = domainBo.findAllDomain(100000);
for(Domain domain : list){
   process(domain.getDomainName());
}
```

  - It can easily hit java.lang.OutOfMemoryError: Java heap space :

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.HashMap.<init>(HashMap.java:209)
at java.util.LinkedHashMap.<init>(LinkedHashMap.java:181)
```

# Java memory management

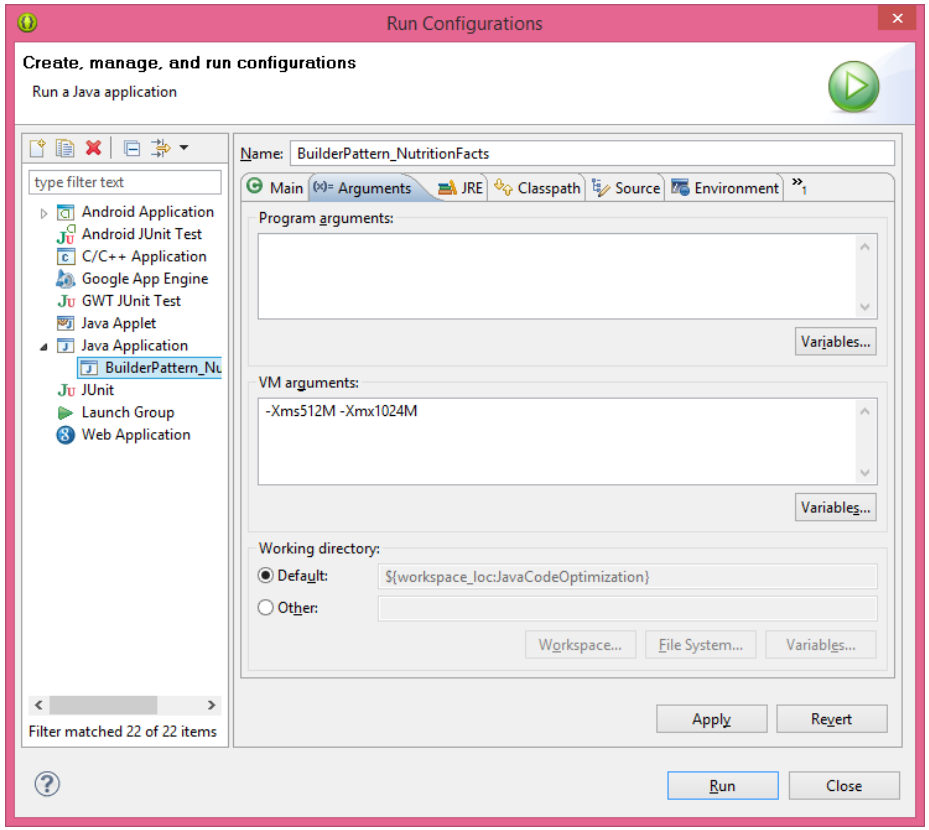
- Define Java heap space for your application
  - Solution – VM arguments

    On Eclipse menu, clicks Run -> Run Configurations.., select the Java application you want to run, clicks on the Arguments tab:

    -Xms<size> - Set initial Java heap size

    -Xmx<size> - Set maximum Java heap size

    For example, -Xms512M -Xmx1024M



# Java memory management

- Define Java heap space for your application using command line
  - **Java Heap Size**

    ```
    $ java -Xms512m -Xmx1024m JavaApp
    ```

  - **Perm Gen Size**

    ```
    $ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
    ```

  - **Java Stack Size**

    ```
    $ java -Xss512k JavaApp
    ```

# When an object becomes eligible for Garbage Collection

- Object becomes eligible for garbage collection if its *all references are null.*
- Cyclic dependencies are not counted as reference
  - If Object A has reference of object B and object B has reference of Object A and they don't have any other live reference
- Object is created inside a block and reference goes out scope once control exit that block.
- Parent object set to null.

# Static factory methods (1/3)

- A class can provide a public *static factory method* that returns an instance of the class

```
public static Boolean valueOf(boolean b) {
        return b ? Boolean.TRUE : Boolean.FALSE;
}
```

- Advantages:
  - Unlike constructors, they have names
  - They are not required to create a new object each time they're invoked
  - They can return an object of any subtype of their return type

# Static factory methods (2/3)

- Advantages:
  - They reduce the verbosity of creating parameterized type instances
  - This typically requires you to provide the type parameters twice in quick succession:

```
Map<String, List<String>> m = new HashMap<String, List<String>>();
```

  - With static factories, the compiler can figure out the type parameters for you

```
public static <K, V> HashMap<K, V> newInstance() {
        return new HashMap<K, V>();
}
```

  - Then you could replace the wordy declaration above with this succinct alternative:

```
Map<String, List<String>> m = HashMap.newInstance();
```

# Static factory methods (3/3)

- Disadvantages:
  - Classes without public or protected constructors cannot be subclasses
  - They are not readily distinguishable from other static methods. Here are some common names for static factory methods:
    - **valueOf, getInstance, newInstance, get*Type*, new*Type***

# Builder Pattern

- Static factories and constructors share a limitation: they do not scale well to large numbers of parameters: *telescoping constructor* pattern

```
class NutritionFacts {
    private final int servingSize; // (mL) required
    private final int servings; // (per container) required
    private final int calories; // optional
    private final int fat; // (g) optional
    private final int sodium; // (mg) optional
    private final int carbohydrate; // (g) optional
    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories) {
        this(servingSize, servings, calories, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories, int fat,
            int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories, int fat,
            int sodium, int carbohydrate) {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

# Builder Pattern

- The telescoping constructor pattern works, but it is hard to write client: using JavaBeans pattern
  - JavaBean may be in an inconsistent state partway through its construction

```
class NutritionFacts {
    private int servingSize = -1; // Required; no default value
    private int servings = -1; // " " " "
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }
    // Setters
    public void setServingSize(int val) { servingSize = val; }
    public void setServings(int val) { servings = val; }
    public void setCalories(int val) { calories = val; }
    public void setFat(int val) { fat = val; }
    public void setSodium(int val) { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

# Builder Pattern

- Combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern: *Builder* pattern

```
class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }
```

# Builder Pattern

```
        public Builder calories(int val) {calories = val;
            return this;
        }
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
        public Builder sodium(int val) {
            sodium = val;
            return this;
        }
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

# Builder Pattern

- This client code is easy to write and, more importantly, to read

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
calories(100).sodium(35).carbohydrate(27).build();
```

# Singleton pattern using enum type

- Making constructor private and having another factory method to get instance

```
public class OldSingleton {
  private static OldSingleton instance = null;
  private OldSingleton()  {
  }
  public static OldSingleton getInstance() {
    if(instance == null)
      instance = new OldSingleton();
    return instance;
  }
}
```

# Singleton pattern using enum type

- Making constructor private and having another factory method to get instance:

```
public class OldSingleton {
  private static final OldSingleton instance = new OldSingleton();
  private OldSingleton() {
  }
  public static OldSingleton getInstance() {
    return instance;
  }
}
```

- First one is not thread safe and the second creates the Singleton Object even before it is actually required.

# Singleton pattern using enum type

- Refine our first approach to make it thread safe

```
public class OldSingleton {
  private static OldSingleton instance = null;
  private OldSingleton() {
  }
  public static synchronized OldSingleton getInstance(){
    if(instance == null)
      instance = new OldSingleton();
    return instance;
  }
}
```

# Singleton pattern using enum type

- Do one more refinement by fine grained locking

```
public class OldSingleton {
  private static OldSingleton instance = null;
  private OldSingleton() {
  }
  public static OldSingleton getInstance(){
    if(instance == null) {
      synchronized (OldSingleton.class) {
       if(instance == null)
         instance = new OldSingleton();
      }
    }
  return instance;
 }
}
```

# Singleton pattern using enum type

- Create singleton using Java enum.

```
public enum MySingleton {
    INSTANCE;
    public void sayHello() {
       System.out.println("Hellod");
    }
    public void sayBye()  {
      System.out.println("Bye);
    }
}
```

```
public class TestSingleton {
  public static void main(String[] args) {
    MySingleton singleton = MySingleton.INSTANCE;
    singleton.sayHello();
    singleton.sayBye();
  }
}
```

# Avoid creating unnecessary objects

- If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly

```
String s = new String("stringette");
```

- This code guarantees that the object will be reused

```
String s = "stringette";
```

# Avoid creating unnecessary objects

- The isBabyBoomer method unnecessarily creates a new Calendar, TimeZone, and two Date instances each time it is invoked

```
public class Person {
    private final Date birthDate = null;

    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        // Unnecessary allocation of expensive object
        Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0
                && birthDate.compareTo(boomEnd) < 0;
    }
    public static void main(String[] args) {

    }

}
```

# Avoid creating unnecessary objects

- The version that follows avoids this inefficiency with a static initializer

Optimized_Person.java

```
package unnecessaryobjects;

import java.util.Date;

public class Optimized_Person {
    private final Date birthDate=null;
    private static final Date BOOM_START;
    private static final Date BOOM_END;
    static {
        Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }
    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0
                && birthDate.compareTo(BOOM_END) < 0;
    }
    public static void main(String[] args) {
    }

}
```

# Avoid creating unnecessary objects

- Change Long to long in code below:

```
public static void main(String[] args) {
  Long sum = 0L;
  for (long i = 0; i < Integer.MAX_VALUE; i++) {
    sum += i;
  }
  System.out.println(sum);
}
```

- Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer
- If the conversion goes the other way, this is called unboxing.

# Eliminate obsolete object references

- The program has a “memory leak,” which can silently manifest itself as reduced performance.

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

# Eliminate obsolete object references

- NULL out references once they become obsolete

```
public class FixedStack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public FixedStack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

# Using String literal instead of String object

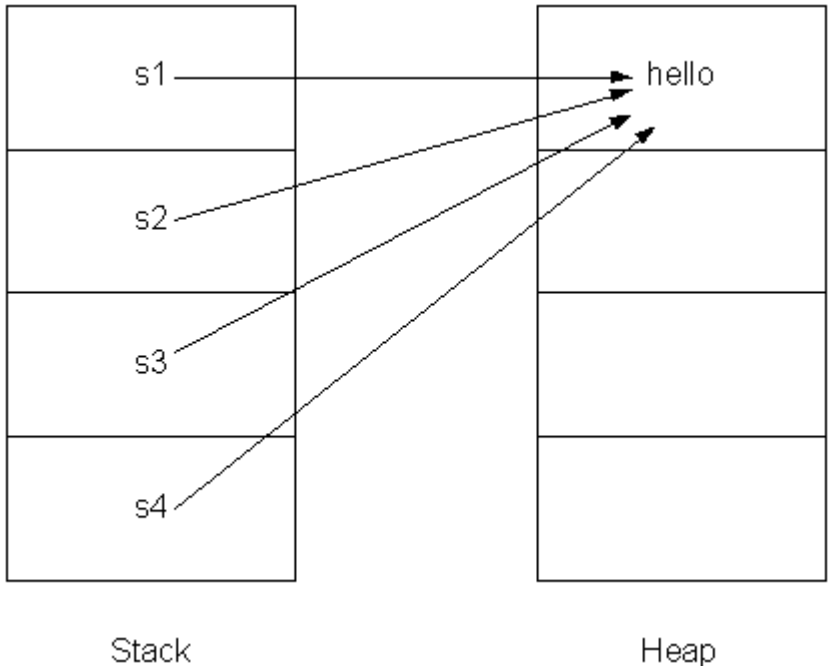
- You can create String objects in the following ways.
- Way #1
  - String s1 = "hello";
  - String s2 = "hello";
- Way #2.
  - String s3 = new String("hello");
  - String s4 = new String("hello");
- Which of the above gives better performance?

# Using String literal instead of String object

- How the JVM works with Strings:
- Using == operator and String.equals() method.
  - == operator returns true if the references point to the same object
  - String.equals() method checks the contents of the String

# Using String literal instead of String object

- Optimization by Interning Stings
  - String.intern() method avoids duplicating String objects



# Optimization techniques when Concatenating Strings

- Compile time resolution versus run time resolution
  - Interestingly the + operator is faster than StringBuffer.append() method.
  - Before compilation:
    - String result = "This is"+"testing the"+"difference"+"between"+"String"+"and"+"StringBuffer";
  - After compilation
    - String result = "This is testing the difference between String and StringBuffer";
  - String object is resolved at compile time where as StringBuffer object is resolved at run time

# Optimization techniques when Concatenating Strings

- Compile time resolution versus run time resolution

```
public class StringConcat {
    public static void main(String[] args) {
        // Test the String Concatination
        int count = 74836;
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < count; i++) {
            String result = "This is" + "testing the" + "difference"
                    + "between" + "String" + "and" + "StringBuffer";
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time taken for string concatenation using + operator : "
                + (endTime - startTime) + " milli seconds");
        // Test the StringBuffer Concatination
        long startTime1 = System.currentTimeMillis();
        for (int i = 0; i < count; i++) {
            StringBuffer result = new StringBuffer();
            result.append("This is");
            result.append("testing the");
            result.append("difference");
            result.append("between");
            result.append("String");
            result.append("and");
            result.append("StringBuffer");
        }
        long endTime1 = System.currentTimeMillis();
        System.out.println("Time taken for String concatenation using StringBuffer : "
                + (endTime1 - startTime1) + " milli seconds");
    }
}
```

- Time taken for string concatenation using + operator : 0 milli seconds
- Time taken for String concatenation using StringBuffer : 46 milli seconds

# Optimization techniques when Concatenating Strings

- Using StringBuffer instead of String

```
public class StringBufferTest {
    public static void main(String[] args) {
        // Test the String Concatenation using + operator
        long startTime = System.currentTimeMillis();
        String result = "hello";
        for (int i = 0; i < 1500; i++) {
            result += "hello";
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time taken for string concatenation using + operator : "
            + (endTime - startTime) + " milli seconds");

        // Test the String Concatenation using StringBuffer
        long startTime1 = System.currentTimeMillis();
        StringBuffer result1 = new StringBuffer("hello");
        for (int i = 0; i < 1500; i++) {
            result1.append("hello");
        }
        long endTime1 = System.currentTimeMillis();
        System.out.println("Time taken for string concatenation using StringBuffer :  "
            + (endTime1 - startTime1) + " milli seconds");
    }
}
```

- Time taken for string concatenation using + operator : 16 milli seconds
- Time taken for string concatenation using StringBuffer :  0 milli seconds

# Class and Instance Initialization

```
class cls_init1 {
    static class Data {
            private int month;
            private String name;
            Data(int i, String s) {
                    month = i;
                    name = s;
            }
    }
    Data months[] = {
            new Data(1, "January"),
            new Data(2, "February"),
            new Data(3, "March"),
            new Data(4, "April"),
            new Data(5, "May"),
            new Data(6, "June")
    };
    public static void main(String args[]) {
            final int N = 250000;
            cls_init1 x;
            Timer t = new Timer();
            for (int i = 1; i <= N; i++)
                    x = new cls_init1();
            t.print("data declared non-static");
    }
}
```

```
class Timer {
    long t;
    public Timer() {
            reset();
    }
    public void reset() {
            t = System.currentTimeMillis();
    }
    public long elapsed() {
            return System.currentTimeMillis() - t;
    }
    public void print(String s) {
            System.out.println(s + ": " + elapsed());
    }
}
```

# Class and Instance Initialization

- Turn the number/name data into a class variable, with a single copy across all instances

```
class cls_init2 {
    static class Data {
            private int month;
            private String name;
            Data(int i, String s) {
                    month = i;
                    name = s;
            }
    }
    static Data months[] = {
            new Data(1, "January"),
            new Data(2, "February"),
            new Data(3, "March"),
            new Data(4, "April"),
            new Data(5, "May"),
            new Data(6, "June")
    };
    public static void main(String args[]) {
            final int N = 250000;
            cls_init2 x;
            Timer t = new Timer();
            for (int i = 1; i <= N; i++)
                    x = new cls_init2();
            t.print("data declared static");
    }
}
```

# Inner Classes

- JVM has restrictions on calling private members from outside of their class.
- A special access method is generated by the compiler and added internally to the meth_inner class

```
public class meth_inner {
        private void f() {
            System.out.print("hello!");
        }
        class A {
                A() {
                        f();
                }
        }
        public meth_inner() {
                A a = new A();
        }
        public static void main(String args[]) {
                meth_inner x = new meth_inner();
        }
}
```

# Inner Classes

- A generated method has a name access$0(), and it in turns calls f().
- If you use the JDK utility program that disassembles .class files, by saying:

```
$ javap -c meth_inner
```

- The output includes the following method definition:

```
Method void access$0(meth_inner)
  0 aload_0
  1 invokespecial #7
  4 return
```

# Using Number instead of Strings

- Creating a Double from a string takes about 15 times as long as from a number.

```
public class meth_inner {
    public static void main(String args[]) {
        final int N = 100000;
        Double d;
        Timer t = new Timer();
        for (int i = 1; i <= N; i++)
            d = new Double(12.34);
        t.print("as number");

        t.reset();
        for (int i = 1; i <= N; i++)
            d = new Double("12.34");
        t.print("as string");

    }
}
```

# Loops

- Most people write their for loops like this:

```
for (i=0; i<n; i++) {
  // do some stuff
}
```

- But, since in almost every language, comparing an int to 0 is faster.

```
 for (i=n-1; i>=0; i--){
  // do some stuff
}
```

- You don't have to do the substraction at the beginning of the loop.

```
for (i=n; --i>=0;) {
  // do some stuff
}
```

# Java Optimization Tip #1

- Data Structures
  - Have a profound influence on performance
    - Early design helps once
  - Choice of data structure can constrain what algorithms you can use
- Proportionality to Caller
  - Foo() takes 1 ms. Bar() calls foo.
    - If Bar() takes 20 ms, it's not worth looking at Foo()
    - If Bar() takes 2ms, then we should look at Foo()

# Java Optimization Tip #2

- 1-1 User Event Rule
  - If something happened a fixed number of times (1-3) for each user event, then it's not worth looking at
  - If something happens 100s of times for each user even then it is worth looking at
- 1-10-100 Rule
  - Assignment – 1 unit of time
  - Method call – 10 units of time
  - New Object or Array – 100 units of time
    - Rule of thumb only. Not scientific.
    - Hard to determine the actual cost
  - Bad idea to try and maintain your own free list. The GC knows best.

# Java Optimization Tip #3

- int getWidth() vs. Dimension getSize()
  - getSize() requires a heap allocated object
  - getWidth() and getHeight() may just be inlined to move the two ints right into the local variables of the caller code

# Java Optimization Tip #4

- Locals are faster than Instance variables
  - Local (stack) variables faster than any member variables of objects
    - Easier for the optimizer to work with
- Inside loops, pull needed values into local variables
  - 1. Slow: message send
    - … i < piece.getWidth()
  - 2. Medium: instance variable
    - … i < piece.width
  - 3. Fast: local variable
    - … final int width = piece.getWidth
    - … i < width
      - This is faster since the JIT can put the value in a native register

# Java Optimization Tip #4

- static (class) methods
  - These are the fastest to call, taking around 220ns.
- final methods
  - These are somewhere between static and instance methods, taking around 300ns.
- instance methods
  - These are a little slower, taking around 550ns.
- interface methods
  - These are surprisingly slow, taking on the order of 750ns to call.
- synchronized methods
  - These are by far the slowest, since an object lock has to be obtained, and take around 1,500ns.

# Java Optimization Tip #5

- Avoid Synchronized (Vector)
  - Synchronized methods have a cost associated with them
    - This is significantly improved in Java 1.3
  - Can have synchronized and unsynchronized methods and switch based on some flag
  - Use “immutable” objects to finesse synchronization problems
  - *Immutable* object cannot be changed after it is constructed.
  - Vector class is sychronized for everything
    - Use ArrayList instead!
    - If you can use a regular array, even better

# Java Optimization Tip #6

- Don’t Parse
  - Obvious but slow strategy – read in XML, ASCII, etc.
  - Build a big data structure
- Faster approach
  - Read into memory, but keep as characters
  - Search/Parse when needed
  - Or Parse only subparts

# Java Optimization Tip #7

- Avoid weird code
  - JVM will optimize most stadard coding styles
    - So write code in the most obvious, common way
  - Weird code is often the result of an attempt at optimization!
- Let the JIT/Hotspot do it's thing!

# Java Optimization Tip #8

- Threading / GUI Threading
  - Use separate thread to ensure the GUI is *snappy*
- Pros
  - Makes best use of parallel hardware
- Cons
  - Software is harder to write
  - Bugs can be subtle
  - Locking/Unlocking costs

# Java Optimization Tip #9

- A local variable assigned only once can be declared final.
- A method argument that is never assigned can be declared final.
- Detects when a new object is created inside a loop
- ArrayList is a much better Collection implementation than Vector.
- AddEmptyString

```
String s = "" + 123; // bad
String t = Integer.toString(456); // ok
```

# References

- Effective Java
  - Joshua Bloch
- Performance improvement techniques in String and StringBuffer
  - comments@precisejava.com
- Java™ Performance Tuning and Java Optimization Tips
  - Glen McCluskey