



# Java Roads Less Traveled

---

Gary Murphy  
Hilbert Computing, Inc.  
glm@hilbertinc.com



# Motivation

---

- Java is a large development system with lots of class libraries and capabilities.
- In most development environments, we don't get a chance to explore much.
  - We use a comfortable subset of the things available
  - We may not recognize language constructs in other people's code
  - We may not be aware of capabilities that would be time-saving for us

# Constructor Overloading

- A constructor can call an overloaded version of the constructor with `this(...)`

```
package com.hilbertinc.css2005;
public class ConstructorThis {
    private String name;
    public ConstructorThis(String name) {
        super();
        this.name = name;
    }
    public ConstructorThis() {
        this("default");
    }
}
```



# instanceof & Class Loaders

---

- The **instanceof** operator is used to determine if one class is of the same type as another.
- A class is an instance of another if:
  - The Class object from which it is constructed is the same type **and...**
  - The class loader from which the two compared instances were instantiated is the same



# isInstance()

---

- If I have an object of type Class, how do I see if an object instance is of that type?

```
public ArtifactSubset(Collection complete, Class type) {
    super();
    this.complete = complete;
    for (Iterator i = complete.iterator(); i.hasNext();) {
        Object element = i.next();
        if (type.isInstance(element)) {
            super.add(element);
        }
    }
}
```



# isAssignableFrom()

---

- This is similar to `isInstance()`, but is used with two Class objects:

```
public static Object create(Class type, Class assign){
    if (assign.isAssignableFrom(type)) {
        return create(type);
    }
    else {
        return null;
    }
}
```



# Name of a Class

---

- The name of a class can be obtained:
  - `SomeClass.class.getName()` ;  
in static methods.
  - `SomeClass instance = new SomeClass();`  
`instance.getClass().getName();`  
for the name of the class for an instance.
- This is preferable to coding the string literal since this will survive refactoring in IDEs.
- I use this for error reporting and populating factory objects



# References

---

- Allow for limited interaction with the garbage collector. There are three types of references:
  - **WeakReference** – the object to which it refers can be garbage-collected if there are no other references to it.
  - **SoftReference** – the object to which it refers can be gc'd if the memory is needed. This is at the discretion of the garbage collector
  - **PhantomReference** – the object to which it refers is garbage-collected as normal, but the **PhantomReference** object is enqueued when it is ready for finalization



# ReferenceQueue

---

- When one of the Reference objects is ready for finalization, it can be placed on a queue.

```
public void validate() {  
    Iterator iterator = getReferences().iterator();  
    while (iterator.hasNext()) {  
        Reference reference =  
            (Reference)iterator.next();  
        if (reference.isEnqueued()) {  
            iterator.remove();  
        }  
    }  
}
```



# SoftReference

---

- Soft references are useful for memory adjusted caches.
  - Soft references are reclaimed when the garbage collector determines that space is low in the JVM.
  - Maps of soft references instead of a map of the actual objects will allow the gc to reclaim space from the map



# WeakReference

---

- Can be used to keep references to objects in a collection as long as there are other hard references to that object.
  - This is useful to prevent “memory leaks” (really unwanted references)
- The `java.util.WeakHashMap` is an implementation of a `HashMap` with the keys saved as weak references.



# PhantomReference

---

- This is useful as sort of replacement for `finalize()` processing.
- Can augment explicit lifecycle management
  - I had a singleton object that started a thread. Callers were supposed to obey lifecycle methods, so I could reference count and terminate the thread. If callers didn't do that, my thread would not terminate. Phantom references can be used as a fall back implementation for lifecycle management when callers don't use your object correctly



# ThreadLocal

---

- Each Thread object gets an independent copy of the variable
- When the thread goes away, all copies of thread local instances are subject to garbage collection unless there are other references
- The programming interface is pretty arcane. Here's an example...



# ThreadLocal Example

---

```
public class ThreadVariableMap {
    private ThreadLocal variableMap
        = new ThreadLocal() {
        protected Object initialValue() {
            return new HashMap();
        }
    };
    public ThreadVariableMap() {
        super();
    }
    .
    .
    .
}
```



# ThreadLocal Example (Continued)

---

```
        .  
        .  
        .  
protected Map getVariables() {  
    return (Map) variableMap.get();  
}  
public Object getAttribute(String name) {  
    return getVariables().get(name);  
}  
public void setAttribute(String n, Object v) {  
    getVariables().put(n, v);  
}  
}
```



# ThreadLocal Usage

---

- Create a static reference to the ThreadVariableMap and you will have a unique copy per thread even though it looks like a singleton.
- Useful for “stuffing” the value of a transaction ID or the currently-authenticated user
- Bypasses the expressed object model, so this can be overused and dangerous.

*See ThreadVariableMap code on CD*

# InheritableThreadLocal

---

- This extends `ThreadLocal` such that when a child thread is created, it automatically receives initial values from the parent's copy.
  - Those initial values can be changed via an override of the `childValue(...)` method
- Again, thread locals can be over used.
  - I recommend these being available as services within a framework, but not a part of typical application code



# 'volatile' Keyword

---

- Fields in an object can be declared with the modifier `volatile`.
  - Java threads can keep working copies in caches for performance
  - If a variable is `volatile`, it will not be cached. This will cause all updates to be done in main memory.
  - This is not a replacement for synchronized methods/blocks.
  - It is done primarily for atomic operations such as updating an `int`.



# Infrequently Initialized Fields

---

- Consider the following:

```
public class Descriptor {  
    private String[] list = new String[0];  
  
    .  
    .  
    .  
    public String[] getList() {  
        return list;  
    }  
}
```



# Infrequently Initialized Fields

---

- What if there are a lot of `Descriptor` objects and most of them never get a populated list.
  - We create a `String[]` object for each descriptor, only to be garbage-collected
- We can do lazy-construction...
  - Initialize the list to a null reference
  - When `getList()` is called, construct the empty string object

# Infrequently Initialized Fields

- ... or we can use this technique:

```
public class Descriptor {
    private static final String[] empty =
        new String[0];

    private String[] list = empty;
    .
    .
    .
    public String[] getList() {
        return list;
    }
}
```



# Hash Code

---

- `java.lang.Object` has a method called `hashCode ()`
  - The requirement is if two objects are `equal ()`, their hash codes have to be equal
  - It is not a requirement that hash codes have to be unique, so you could always return a literal number such as zero, but you shouldn't because...
- Hash codes are used as a quick test of equality in collections. If hash codes are equal, the more expensive `equals ()` is called.



# Math and StrictMath

---

- Ever wondered where `min`, `max`, etc. functions are located?
  - They are in the `java.lang.Math` class.
- If you need to handle floating point arithmetic in a portable way...
  - Floating point arithmetic is different on different CPU architectures
  - The `StrictMath` class implements IEEE 754 core functions



# Runtime

---

- Can be used to run a separate process with the `exec ( . . . )` methods.
- Memory statistics are available via:
  - `freeMemory ( )`
  - `totalMemory ( )`
  - `maxMemory ( )`
- Shutdown hooks are ready-to-run threads that are started when the JVM is terminating.
  - I used to terminate a singleton thread object



# Shutdown Hook Example

---

```
public class ShutdownHook extends Thread {
    private Something reference;

    public ShutdownHook(Something reference) {
        super();
        this.reference = reference;
    }
    public void run() {
        reference.cleanup();
    }
}
```



# Shutdown Hook Example

---

```
•  
•  
•  
ShutdownHook hook = new ShutdownHook ();  
Runtime.getRuntime().addShutdownHook(hook);  
•  
•  
•
```

# Inspecting a Stack Trace

- Any Throwable object can return an array of stack frames. One stack frame per method call.
- Could make a good Logger/Log4j appender

```
protected void inspect(Throwable throwable) {
    StackTraceElement[] trace =
        throwable.getStackTrace();
    String name = trace[0].getClassName();
    if (name.startsWith("com.hilbertinc.")) {
        System.err.println("Failed at "+name+
            " (" +trace[0].getLineNumber()+")");
    }
}
```



# Zip File Manipulation

---

- Java supports zip files manipulation
  - ZipOutputStream and ZipInputStream
  - ZipEntry represents a file entry in the zip archive
- An example follows...



# Zip Example Code

---

```
OutputStream fStream = new FileOutputStream(getZipName());
ZipOutputStream zipStream = new ZipOutputStream(fStream);
File[] children = base.listFiles();
for (int i = 0; i < children.length; i++) {
    File child = children[i];
    if (child.isFile()) {
        ZipEntry entry = new ZipEntry(child.getName());
        entry.setTime(child.lastModified());
        zipStream.putNextEntry(entry);

        // Copy the file contents into the zip file

        FileInputStream source = new FileInputStream(child);
        byte[] block = new byte[4096];
        int length;
        while(-1 != (length = (source.read(block)))) {
            zipStream.write(block, 0, length);
        }
        zipStream.closeEntry();
        source.close();
    }
}
zipStream.close();
fStream.close();
```



# Unzip Example Code

---

```
InputStream fStream = new FileInputStream(zipFile);
ZipInputStream zipStream = new ZipInputStream(fStream);
ZipEntry entry = zipStream.getNextEntry();
while(null != entry) {
    if (!entry.isDirectory()) {
        OutputStream expand = new FileOutputStream(entry.getName());
        byte[] block = new byte[4096];
        int length;
        while(-1 != (length = (zipStream.read(block)))) {
            expand.write(block, 0, length);
        } // block copy loop
        expand.close();
    } // if directory
    zipStream.closeEntry();
    entry = zipStream.getNextEntry();
} // while
zipStream.close();
fStream.close();
```



# Java Preferences API

---

- Java has APIs to save small amounts of data at the user level or the system level
  - This is similar in concept to the Windows Registry
  - The information is stored in the Windows Registry in the Sun implementation on Windows
  - The information was stored in the /etc directory on Unix systems in the Sun implementation, but now depends on the release due to write permission problems
  
- Implemented in JDK 1.4

# Java Preferences API

---

- Stored as a hierarchical collection
  - Separate trees for user-specific data and system-wide data
- Have path names similar to filesystem naming
  - There are relative and absolute path names
- Methods support type-specific data. Not limited to Strings.
- By convention, the Java class is used as the key



# Java Preferences API

---

- Intended for desktop application to store font choices, configuration information, window positioning, etc.
- I used this as an alternative to system parameters for low level trace:
  - Instead of

```
java -Dframework.trace=1 com...
```
  - use system or user preferences to store the trace preferences for classes.



# Preferences Example

---

```
protected void setForUser(String variableName,
                          String value) {
    Preferences preferences =
        Preferences.userNodeForPackage(getClass());
    if (null == value) {
        preferences.remove(variableName);
    }
    else {
        preferences.put(variableName, value);
    }
}
```



# Socket Factories

---

- **SocketFactory** and **ServerSocketFactory** objects create sockets.
  - Can be used instead of the **new** operator.
  - SSL implementations are subclasses
  - Custom factories can be used to set timeouts, security parameters, etc.
- The static **getDefault()** method will return a basic factory suitable for many purposes



# Timers

---

- Java has facilities for executing code in the future.
- The **Timer** class can schedule for a single execution or execution at a fixed rate.
- A concrete implementation of a **TimerTask** object will run on the timer's background thread.
  - Subclass **TimerTask** and implement the **run ()** method. Schedule the task on the timer.

# XML Entity Processing

- If an XML document is validated against the DTD when parsed, it needs access to the DTD
  - The reference is typically a URI that is a URL that resolves to the DTD
  - Your code may work until network access to the DTD isn't available or...
  - ... you may need to handle a URI that isn't a URL.
- The `EntityResolver` interface can handle this



# SIGQUIT

---

- A QUIT signal on the JVM will dump a current thread snapshot to standard output.
  - **Not a Java specification**, but works on Sun JVMs
  - A QUIT signal can be sent on Unix-type operating systems via the kill command:

```
kill -QUIT pid
kill -3 pid
```
  - If there is a terminal window, **Ctrl-\** will send a SIGQUIT to the JVM



# wait()/notify()

---

```
public class BlockingQueue {
    private LinkedList queue = new LinkedList();
    public BlockingQueue() {
        super();
    }
    public synchronized Object dequeue() {
        while (getQueue().isEmpty()) {
            try {
                wait();
            }
            catch (InterruptedException exception) {}
        }
        return getQueue().removeFirst();
    }
    public synchronized void enqueue(Object request) {
        getQueue().addLast(request);
        notify();
    }
}
```



# Serialization

---

- Serialization is a simple way to persist the state of a Java object.
  - The class must be marked as **Serializable**
  - All embedded objects must be marked as **Serializable** also.
  - The constructor of the class is not called when the object is deserialized.
  - I recommend using other persistence techniques such as marshaling to XML or O/R mapping



# Serialization Example

---

```
public class Serialize implements Serializable {
    public Serialize() {
        super();
        System.out.println("Constructing...");
    }
    public static void main(String[] args) {
        System.out.println("Running main...");
        Serialize example = new Serialize();
        FileOutputStream fostream = new FileOutputStream("/tmp/s.out");
        ObjectOutputStream ostream = new ObjectOutputStream(fostream);
        ostream.writeObject(example);
        System.out.println("Reloading...");
        FileInputStream fistream = new FileInputStream("/tmp/s.out");
        ObjectInputStream istream = new ObjectInputStream(fistream);
        Serialize reloaded = (Serialize)istream.readObject();
    } // Omitted closing files and try/catch to fit on a slide
}
```

# Serialization Output

---

- The output from the previous example is:

```
Running main...  
Constructing...  
Reloading...
```

# Serialization Versioning

- If you serialize objects, you should version them manually.
  - If you don't version your objects the compiler will do it for you.
- To version:

```
private static final long serialVersionUID = 1L;
```

the number can be anything and should change when the serialized contents change



# 'transient' Variables

---

- The transient modifier for a Java field will prevent it from being serialized.



# References

---

- <http://mindprod.com/jgloss/weak.html> – Roedy Green has an excellent discussion of Java References



# Thank You

---

- Thank you for taking the time to attend this session. I hope it has been helpful
- Fill out the session evaluations. They are helpful to Wayne & Peggy and me.
- Feel free to contact me via e-mail at:  
[glm@hilbertinc.com](mailto:glm@hilbertinc.com)
- I will be at the conference all week. Feel free to ask any questions that arise after the session.