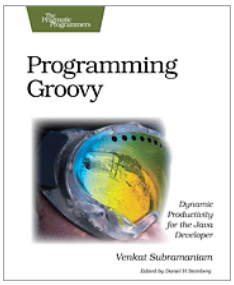
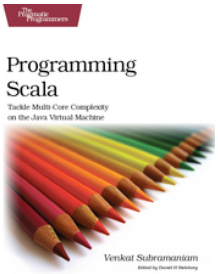
Effective Java

Venkat Subramaniam
venkats@AgileDeveloper.com





Jeopardy Style

- In this Jeopardy Style presentation we will discuss various topics in Java
- You will drive the presentation, selecting topics, answering questions, and bringing out your experience as much as mine
- You can download the examples and slides from my web site for future reference

What's here?

- This slides contains a select set of quiz problems.
- You can view the entire set of quiz from the files attached.

Singleton

- What's Wrong with this code?

```
public class Singleton
{
  private static Singleton _instance;

  private Singleton() {}

  public static Singleton getInstance()
  {
    if (_instance == null) _instance = new Singleton();
    return _instance;
  }

  //...
}
```

Singleton

- Several things!
- Hard to Test Client—the user of a singleton is hard to test as mocking it is quite difficult.
- Not Thread Safe and making it thread safe is not easy
  - If you're not sure check out <http://www.yoda.arachsys.com/csharp/singleton.html>.
- No Guarantee of Singleton. Using reflection, you can get around private!
  - You may have to throw exception from constructor if second instance created

Singleton

- Not extensible—private constructor makes is impossible to inherit from this class. So you can't extend it.
- Serialization can break singleton. You have to make fields transient and provide readResolve() method.

Singleton

- Why not use enum for this purpose?
- Not so intuitive, but can solve the problems.

```
public enum Singleton
{
  INSTANCE

  //...
}
```

- Concise
- Reflection-proof
- Serializable
- Thread safe

Cleanup

- What will the file output.txt contain?

```
public class Sample {
  private FileWriter _writer;

  public Sample() throws Exception {
    _writer = new FileWriter("output.txt");
  }

  protected void finalize() throws Exception {
    _writer.close();
  }

  public void info(String msg) throws Exception {
    _writer.write(msg);
  }

  public static void main(String[] args) throws Exception {
    Sample obj = new Sample();
    obj.info("test");
  }
}
```

Finalizer

- Illusionary—Programmers think these help with proper timely cleanup.
- Unpredictable—No guarantee if and when it will be called
- Unnecessary—Right way to solve the problem is to use try-finally
- Dangerous—Resources may be left unclaimed and may result in errors (like out of limited resources)
- May result in race conditions or delay as no guarantee which thread may reclaim

Finalizer

- Can leave you in limbo—Uncaught exceptions during finalization leaves an object in an invalid unreclaimed state
- Slow—Adding finalizers slows object destruction by about 400 times

Step 1

```
public class Sample {
  private FileWriter _writer;

  public Sample() throws Exception {
    _writer = new FileWriter("output.txt");
  }

  public void close() throws IOException
  {
    _writer.close();
  }

  public void info(String msg) throws Exception {
    _writer.write(msg);
  }

  public static void main(String[] args) throws Exception {
    Sample obj = new Sample();
    obj.info("test");
    obj.close();
  }
}
```

- Still has problems...

Step 2—try-finally

```
Sample obj = new Sample();
try
{
  obj.info("test");
}
finally
{
  obj.close();
}
```

Deterministic Cleanup

```
//...
interface SampleUser {
  public void use(Sample sample) throws Exception;
}

public static void use(SampleUser user) throws Exception {
  Sample sample = new Sample();
  try {
    user.use(sample);
  }
  finally {
    sample.close();
  }
}

public static void main(String[] args) throws Exception {
  Sample.use(new SampleUser()
  {
    public void use(Sample sample) throws Exception
    {
      sample.info("testing...");
    }
  });
}
```


Closures Can Simplyfy This

In languages that support closures (Groovy, Scala, JRuby, ...) you can take advantage to write concise code

```
class Sample
{
  def Sample() { println "initialize..."}
  def close() { println "cleanup..."}

  def operation1() { println "operation 1 called..."}
  def operation2() { println "operation 2 called..."}

  static def use(closure)
  {
    def sample = new Sample()
    try
    {
      closure.delegate = sample
      closure(sample)
    }
    finally
    {
      sample.close()
    }
  }
}

Sample.use {
  operation1()
  operation2()
}
```

```
initialize...
operation 1 called...
operation 2 called...
cleanup...
```

Clone

- What's up with this?

```
public class Equipment implements Cloneable
{
  private final int id;
  private static int uniqueId;

  public Equipment()
  {
    id = uniqueId++;
  }

  public int getId() { return id; }

  public Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}
```


Clone Issues

- Clone is problematic in several ways
- Constructor not called
- Not thread safe by default, you need to synchronize if you want thread safety
- No guarantee how clone is implemented in a class
- Incompatible with final fields
- Have to be careful handling internal state
- Should not invoke any non-final methods
- You need to suppress the CloneNotSupportedException to make it easier for client to use

Attempt 1

```
public Object clone() throws CloneNotSupportedException
{
  Equipment cloned = (Equipment) super.clone();
  cloned.id = uniqueId++; // Will not work
  return cloned;
}
```


Copy Constructor

- Directly using Copy Constructor hurts polymorphism
- Given a reference, how do you create an object of the type referenced?
- Using instanceof (Runtime Type Identification) will not help extensibility in this case

Mix clone and Copy Constr.

```
protected Equipment(Equipment other)
{
  id = uniqueId;
}

public int getId() { return id; }

public Object clone() throws CloneNotSupportedException
{
  return new Equipment(this);
}
```


Generics & Collection

- How does this fall short?

```
public static <T> void copyFromTo(List<T> from, List<T> to)
{
  for (T element : from)
  {
    to.add(element);
  }
}
```

Does not Support Covariance

```
    List<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());

    List<Animal> animals = new ArrayList<Animal>();

    copyFromTo(dogs, animals);
  }
}
```

copyFromTo (List<Dog>, java.util.List<Dog>) in Sample cannot be applied
to (List<Dog>, java.util.List<Animal>)

Ensure Type Compatibility

```
public static <T> void copyFromTo(
  List<? extends T> from, List<T> to)
{
  for (T element : from)
  {
    to.add(element);
  }
}
```

Array

- What's going on here?

```
Object[] values = new Integer[3];
values[0] = 11;
```


Runtime Exception

```
Exception in thread "main" java.lang.ArrayStoreException:
```

- Arrays are Covariant and as a result not type-safe
- You inserted a 1L not a 11 (eleven)
- Can result in Runtime ArrayStoreException
- Generic Lists do not have this problem. By default they are not covariant. So, they can eliminate these kinds of problems at compile time.
- It is better to use List than Array, much safer

Use List

```
List<Integer> values = new ArrayList<Integer>();

values.add(11);
```

add (java.lang.Integer) in List cannot be applied to (long)

Static in Generics

- What's the output of this code?

```
class MyList<T>
{
  public static int count;
  public MyList() { count++; }
  public int getCount() { return count; }
}
public class Sample
{
  public static void main(String[] args)
  {
    MyList<Integer> list1 = new MyList<Integer>();
    MyList<Integer> list2 = new MyList<Integer>();
    MyList<Double> list3 = new MyList<Double>();

    System.out.println(list1.getCount());
    System.out.println(list2.getCount());
    System.out.println(list3.getCount());
  }
}
```

Not What You Desire

3
3
3

- Type erasure erases the type information
- So, there is really no MyList<Integer> or MyList<Double> under the covers
- You only have MyList
- So, static is common across all MyList "types"
- Use Extreme Caution when using static in generics

References

You can download examples and slides from http://www.agiledeveloper.com - download

Thank You!

Please fill in your session evaluations

You can download examples and slides from
http://www.agiledeveloper.com - download