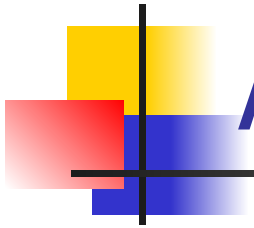
# Effective Java: Creating and Destroying Objects

Last Updated: Fall 2012



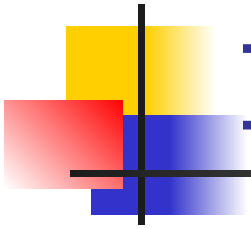
# Agenda

- Material From Joshua Bloch
  - Effective Java: Programming Language Guide
- Cover Items 1-7
- Bottom Line:
  - Understand Java Construction First
  - C++ Creation/Destruction More Complex

# Item 1: Consider Static Factory Methods Instead of Constructors

- Constructor Calls vs Static Factory Method
  - Alternate way to get an object
  - Sometimes replaces constructors
  - Sometimes augments constructors

```
// Simple example from Boolean class
public static Boolean valueOf (boolean b) {
   return b ? Boolean.TRUE : Boolean.FALSE;
}
```



# Item 1: Advantages

- Unlike constructors, static factory methods
  - Can have meaningful names
  - Need not create new instances
  - Can return any subtype of return type
    - Reduces client dependency on specific class
  - Can reduce verbosity of creating parameterized type instances

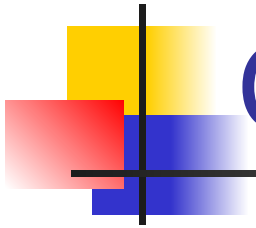
# Advantage 1: Meaningful Names

- Consider the constructor/factory pair:

```
// Constructs a randomly generated positive BigInteger
// that is probably prime, with the specified bitLength
// BigInteger (int bitLength, int certainty, Random rnd)

vs.
// Returns a positive BigInteger that is probably prime,
// with the specified bitLength.

// BigInteger.probablePrime (int bitLength, Random rnd)
```

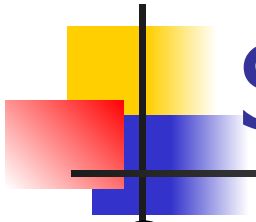
  - Note: The extra constructor argument avoids a clash with another constructor
  - Unique parameter lists on constructors are *really* restrictive

# Advantage 2: Not Required To Create New Object



- Instance-controlled classes can be useful
  - Can avoid creating unnecessary duplicate objects
    - `Boolean.valueOf(boolean)` is an example
  - Can guarantee a "singleton" or "noninstatiable" object
  - Can allow for very fast "equals" test

# Advantage 3: Can Return Subtype of Return Type



- Consider the java.util.Collections class
  - 32 Convenience implementations of Collection *interfaces*
    - All are static factory methods
  - Interface return type vs. actual classes
- Static factory methods can hide multiple implementations
  - `java.util.EnumSet` has two implementations
  - Future release could easily change this
  - Clients neither know nor care about actual type
    - Reduce client dependencies!

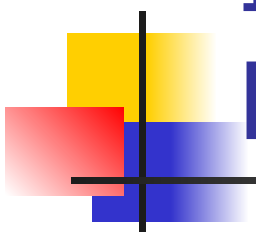
# Service Provider Framework Example

```
// Service interface
public interface Service { ... // Service-specific methods go here }

// Service provider interface
public interface Provider { public Service newService(); }

// Noninstantiable class for service registration and access
public class Services {
   private Services() { } // Prevents instantiation (Item 4)
   private static final Map<String, Provider> providers =
      new ConcurrentHashMap<String, Provider>();

   // static Provider registration API - services may be added long after factory defined
   public static void registerProvider(String name, Provider p){
      providers.put(name, p); }

   // static Service factory API
   public static Service getInstance(String name) {
      Provider p = providers.get(name);
      if (p == null) throw new IllegalArgumentException("No provider named: " + name);
      return p.newService(); }
}
```

# Advantage 4: Reduce Verbosity of Parameterized Type Instances

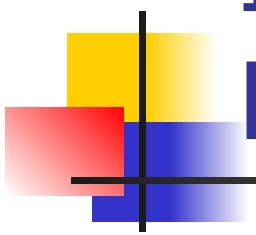
```
// Parameterized type instances
Map<String, List<String>> m = new HashMap<String, List<String>>();

vs.

// Static factory alternative

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

// Now, client code looks like this
// Compiler does type inference!

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

# Item 1: Disadvantages of Static Factory Methods



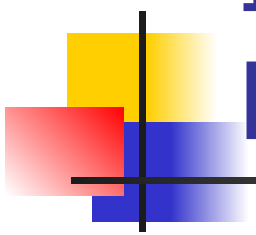
- Subclassing impossible without constructors
  - Arguably a blessing in disguise
- Naming conventions necessary
  - `valueOf` – effectively a type converter (also just `of`)
  - `getInstance` – return instance described by parameters
  - `newInstance` – like `getInstance`, but guarantees distinct object
  - `get`*`Type`* – like `getInstance`, but converts type
  - `new`*`Type`* – like `newInstance`, but converts type

# Item 2: Consider a Builder vs. Many Constructor Parameters



- Static factories and constructors don't scale well to large numbers of optional parameters
- Bloch's examples:
  - NutritionFactsTelescoping.java
  - NutritionFactsBeans.java
  - NutritionFactsBuilder.java
- The last version enjoys significant advantages

# Item 3: Enforce Singleton Property



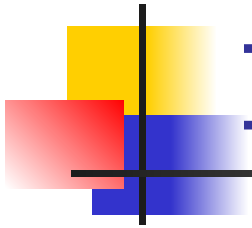
- A *Singleton* is a class that's instantiated exactly once
  - Note: singletons are hard to mock in unit testing
- Two approaches before Enums:
  - Public static member (a constant, of course)
  - Public static factory method
- Enum singleton is now preferred
  - Lots of subtle advantages: security, serialization, etc.

# Item 3: Code Example

```
// Option 1: public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() {...}
}
// Option 2: static factory method
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() {...}
    public static Elvis getInstance() { return INSTANCE; }
}
// Option 3: Enum type – now the preferred approach
public enum Elvis {
   INSTANCE;
   ...
}
```

# Item 4: Enforce Noninstantiability With a Private Constructor

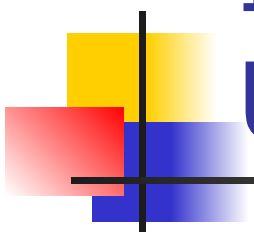
- Some classes just group static methods and/or fields
  - Makes no sense to instantiate such a class
- Trying to enforce noninstantiability by making class abstract doesn't work
  - Subclassing is possible
  - Clients are led to believe subclassing makes sense
- However, a private constructor does the job



# Item 4: Code Example

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    ... // Remainder of class omitted
}

// Note that no subclassing is possible (constructor chaining...)
// Note that client can't call constructor
// Note that if constructor is mistakenly called inside class,
// there is an immediate assertion violation.
```

# Item 5: Avoid Creating Unnecessary Objects



- On the one hand, performance is a secondary concern behind correctness
- On the other, gratuitous object creation is just bad programming

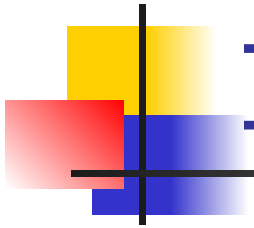
```
// String s = new String("stringette");  // Don't do this!
vs.
// String s = "stringette";  // Let JVM optimize for you

// Also see earlier Boolean.valueOf() static factory example
```

# Item 5: Code Example

```
public class Person {
   private final Date birthDate;
   // Other fields, methods, and constructor omitted
   // 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, 0);
   Date boomStart = gmtCal.getTime();
   gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0, 0);
   Date boomEnd = gmtCal.getTime();
   return birthDate.compareTo(boomStart) >= 0 &&
          birthDate.compareTo(boomEnd)    < 0;
   }
}
```

# Item 5: Code Example Fixed

```
public class Person {
   private final Date birthDate;
   // Other fields, methods, and constructor omitted
   private static final Date BOOM_START;
   private static final Date BOOM_END;

   static {  // Note static block
      Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
      gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0, 0);
      BOOM_START = gmtCal.getTime();
      gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0, 0);
      BOOM_END = gmtCal.getTime();
   }
   public boolean isBabyBoomer() {
      return birthDate.compareTo(BOOM_START) >= 0 &&
             birthDate.compareTo(BOOM_END)    < 0;
   }
}
```



# Item 5: Autoboxing Overhead

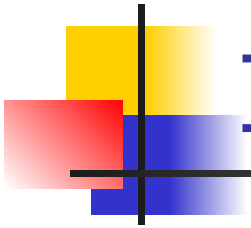
```
//  Hideously slow program!  Can you spot the object creation?
public static void main(String[] args) {
   Long sum = 0L;
   for (long i =0; i < Integer.MAX_VALUE; i++) {
      sum += i;
   }
   System.out.println(sum);
}

// Lessons: 1) prefer primitives to Boxed primitives
//          2) watch for unintentional autoboxing
```

# Item 6: Eliminate Obsolete Object References

- Sometimes, you manage your own memory (leak)
  - Example: Stack.java

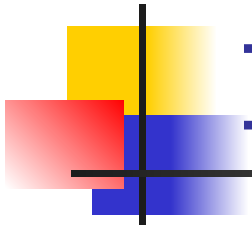
```
public Object pop () {
    if (size == 0) throw new IllegalStateException("Stack.pop");
    Object result = elements[--size];
    elements[size] = null;    // Eliminate obsolete reference
    return result;
}
```

- Also a problem with caches and registration of listeners and callbacks
  - Suggestion: Use *weak* pointers, such as `WeakHashMap`



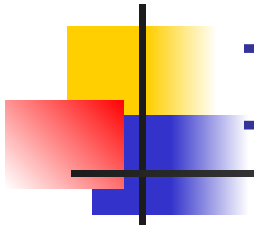
# Item 7: Avoid Finalizers

- `finalize()` is a method in the `Object` class
  - What the garbage collector may call when cleaning up an unused object
- Finalizers: unpredictable, dangerous, unnecessary!
  - They are NOT the analog of C++ destructors
- There is no guarantee a finalizer will ever be called
- Finalizers have severe performance penalties
- Instead, provide explicit termination methods
  - Sometimes requires “finalizer chaining”



# Item 7: Code Example

```
// try-finally block guarantees execution of termination method
// termination method ensures resources are released
// Example resources:  database connections, threads, windows
Foo foo = new Foo();
try {
   // Do what must be done with foo
} finally {
  foo.terminate();  // Explicit termination method in Foo
}
```



# Item 7: Finalizer chaining

```
// Manual finalizer chaining
// Only do this if you *have* to use the finalizer
@Override
protected void finalize() throws Throwable {
  try {
   .. . // Finalize subclass state
  } finally
     super.finalize();  // Note order of finalizer chaining
  }
}

// Also consider "Finalizer Guardian" idiom for public, nonfinal
// classes.  The goal is to ensure that finalization takes place
// even if subclass finalizer fails to invoke super.finalize()
```