Preventing Instantiation

- Default (zero-argument) constructor
  - Provided only if there is no explicit constructor
- Declare a single explicit private constructor
  - Result: No other class can instantiate
  - Note: Including constructor prevents construction!
  - Document the private constructor
- Side effect: Class can not be extended
  - Subclass must call parent’s constructor
  - So, parent’s constructor must be visible
- Use: Utility classes
  - Collection of static members
  - See java.lang.Math, java.util.Arrays
  - Note: including constructor prevents construction!

Example: Non-instantiability

```java
// Non-instantiable utility class
public class UtilityClass {

    // Suppress default constructor
    private UtilityClass() {
        // Constructor never invoked
    }

    // Other parts of class
}
```

Singleton Pattern

- A singleton is a class that is instantiated exactly once
  - Eg Window manager, file system
- Basic recipe
  - Private constructor
  - (One) instance reference in private field
  - Static factory method
- Optimization: Lazy initialization
  - Instantiate only if requested

Example Singleton

```java
// Singleton with static factory
public class Manager {
    private static final Manager INSTANCE = new Manager();

    // Suppress default constructor
    private Manager() {
        // Constructor never invoked
    }

    public static Manager getInstance() {
        return INSTANCE;
    }

    // Other parts of class
}
```

Example Lazy Singleton

```java
// Singleton with static factory and lazy init
public class Manager {

    private static Manager INSTANCE = null;

    // Suppress default constructor
    private Manager() {
        // Constructor never invoked
    }

    public synchronized static Manager getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Manager();
        }
        return INSTANCE;
    }

    // Other parts of class
}
```
Many Subtle Problems

- Multiple threads
  - Static factory must be synchronized
- Multiple classloaders
  - Each classloader has a different instance
- Serialization
  - Saving singleton to disk then re-reading results in new instance

Potpourri: Memory Leaks and Random

Memory Management

- Java (generally) manages memory for you
- Every call to “new” creates a new instance
- Memory allocated to hold instance
- When is this memory released?
  - Answer: when there are no references to this instance
  - eg End of scope
    void someMethod() {
      someClass x = new someClass();
      ...
    } //x goes out of scope
  - (Beware of aliases of course)

Example “Memory Leak”

```java
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack (int initialCapacity) {
        elements = new Object[initialCapacity];
    }

    public void push (Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop () {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
}
```

Example Continued

```java
//class Stack continued...

private void ensureCapacity() {
    if (elements.length == size) {
        Object[] oldElements = elements;
        elements = new Object[2*elements.length + 1];
        System.arraycopy(oldElements, 0,
                         elements, 0, size);
    }
}
```

Example Repaired

```java
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;
    return result;
}
```
Memory Leak: Problem and Solution

- Problem: Keeping obsolete references
  - Stack has array of reference that will never be dereferenced
- Solution: explicitly null-out reference
  - `someReference = null;`
- But, do not do this needlessly
  - Clumsy and complicates code
- When is it needed?
  - Classes that manage their own memory
  - Classes that keep caches
    - WeakHashMap discards entries when key no longer accessible

Know The Libraries: Random

- Generating uniform random [0..bound)
  ```java
  import java.util.Random;
  Random rnd = new Random(); // time seed
  int x = rnd.nextInt(bound);
  ```
- Do not scale using 0-argument version
  ```java
  int x = Math.abs(rnd.nextInt()) % bound;
  ```
- Problems
  - No abs for Integer.MIN_VALUE
  - Short repetition period for bounds small power of 2
  - Uneven distribution for some bounds

To Ponder

```java
static Random rnd = new Random();
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
public static void main(String args[]) {
    int b = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i=0; i < 1000000; i++)
        if (random(b) < b/2)
            low++;
    System.out.println(low); // prints ~666,666
}
```

Summary

- Singleton
  - Instantiated at most once
  - Private constructor ensures no default constructor
  - Static factory returns existing reference
  - Lazy initialization defers instantiation
- Memory Leaks
  - Problem: indefinitely retaining obsolete reference
  - Solution: explicit null-out (only when necessary!)
- Random
  - Use 1-argument (bounded) nextInt method