Lazy Initialization

By: Jack Ashworth, Brandon Cash, Kyle Downing

Introduction

- Java is heavily know for its object-oriented programming
  - By default Java allows two ways of object initialization
    - Eager and Lazy
- Eager
  - The object creation works at compile-time (the ‘usual’ way)
    - Not efficient for large programs or many object initializations
- Lazy
  - The object is only instantiated when it is needed

What is Lazy Initialization?

- Overall Definition
  - … is the pattern of putting off the creation of an object or process until it is needed.
  - The idea behind this pattern is that you may never need an object
    - So you will have saved the initialization costs
- When to use
  - Sometimes optimizations can actually decrease performance even when trying to improve it
  - The use of lazy initialization depends on how often we can avoid initialization of the object and how expensive that it will be
  - Sometimes the use of lazy initialization can increase the level complexity

First method

- The synchronized method
  - This method is very simple
  - It determines if the field is initialized
    - If not, it initializes it
    - If it is, then it just returns it
- The synchronized keyword allows us to use this method with concurrent threads
  - Although at the cost of some performance

```
private FieldType field;

private synchronized FieldType getField() {
  if (field == null) {
    field = computeFieldValue();
  }
  return field;
}
```

Second Method

- Lazy initialization holder class idiom
  -The class gets initialized once the object reads file.holder.field

  -Is useful due to not having to synchronize
  -Class doesn't get initialized until it gets called

```
private static class FieldHolder {
  static final FieldType field = computeFieldValue();
}

private static FieldType getField() {
  return FieldHolder.field;
}
```

Third Method

- Double check idiom
  -This first checks to see if it needs to get initialized
  -Then initializes if needed
  -The lock happens then checks again to see if initialization is still needed

  -Doesn't synchronize after the field is initiated
  -The field gets checked twice
  - Purpose of this way is performance reasons

  -One downside is "The code is a bit convoluted"

```
private volatile FieldType field;

private FieldType getField() {
  FieldType result = field;
  if (result == null) {
    synchronized(this) {
      result = field;
      if (result == null) {
        field = result = computeFieldValue();
      }
    }
  }
  return result;
}
```

FINAL METHOD

- Single check idiom
  -This is similar to the double check but drops the check if need

  -Multiple initializations can occur
  -The keyword in both single and double is volatile

```
private volatile FieldType field;

private FieldType getField() {
  FieldType result = field;
  if (result == null) {
    field = result = computeFieldValue();
  }
  return result.
}
```

Pros

- Useful for:
  - Objects that are expensive to initialize
  - Objects that may end up getting unused by the program
- Can be implemented in many different programming languages (including java)

Cons

- Very difficult to use in threaded applications without causing problems
  - This is due to the possibility of race conditions occurring, which are expensive to avoid
- Can causes slowdowns in an application if it has to wait for when the lazy object is needed

CITES

Effective Java: Use Lazy Initialization Judiciously | by Kyle Carter | CodeX | Medium

Lazy vs. eager instantiation in Java: Which is better? | by Rafael del Nero | InfoWorld