Thread safety and local variables

Asked 15 years, 9 months ago Modified 15 years, 8 months ago Viewed 9k times

8

If I have a local variable like so:

```
Increment()
{
    int i = getFromDb(); // get count for a customer from db
};
```

And this is an instance class which gets incremented (each time a customer - an instance object - makes a purchase), is this variable thread safe? I hear that local variables are thread safe because each thread gets its own stack, etc etc.

Also, am I right in thinking that this variable is shared state? What I'm lacking in the thinking dept is that this variable will be working with different customer objects (e.g. John, Paul, etc) so is thread safe but this is flawed thinking and a bit of inexperience in concurrent programming. This sounds very naive, but then I don't have a lot of experience in concurrent coding like I do in general, synchronous coding.

EDIT: Also, the function call getFromDb() isn't part of the question and I don't expect anyone to guess on its thread safety as it's just a call to indicate the value is assigned from a function which gets data from the db. :)

EDIT 2: Also, getFromDb's thread safety is guaranteed as it only performs read operations.

c#

Share Improve this question Follow

edited Mar 24, 2009 at 22:29

asked Mar 24, 2009 at 22:05
GurdeepS
67.1k ● 112 ● 257 ● 394

7 Answers

Sorted by: Highest score (default)

34

`i` is declared as a local (method) variable, so it only **normally** exists in the stack-frame of `Increment()` - so yes, `i` is thread safe... (although I can't comment on `getFromDb`).

except if:

- `Increment` is an iterator block (i.e. uses `yield return` or `yield break`)
- `i` is used in an anonymous method (`delegate { i = i + 1;}`) or lambda (`foo => {i=i+foo;}`

In the above two scenarios, there are some cases when it can be exposed outside the stack. But I doubt you are doing either.

Note that *fields* (variables on the class) are **not** thread-safe, as they are trivially exposed to other threads. This is even more noticeable with `static` fields, since all threads automatically share the same field (except for thread-static fields).

Share Improve this answer Follow

answered Mar 24, 2009 at 22:10
Marc Gravell
1.1m ● 273 ● 2.6k ● 3k

+1 for completeness (discussion of anon methods and iterator blocks. Static classes and thread safety is the easiest concept to grasp, personally. :) – GurdeepS Mar 24, 2009 at 22:31

IMHO, it is important to clarify that even in the listed exceptions, the local variable is *still* unique to each call to the method. That is, while they can be accessed outside the context of the method, even potentially in a non-thread-safe manner, they will still be thread-safe relative to the *same* local variable in the context of a *different* call to the same method. – Peter Duniho Feb 17, 2017 at 7:24

5

Your statement has two separate parts - a function call, and an assignment.

The assignment is thread safe, because the variable is local. Every different invocation of this method will get its own version of the local variable, each stored in a different stack frame in a different place in memory.

The call to getFromDb() may or may not be threadsafe - depending on its implementation.

Share Improve this answer Follow

answered Mar 24, 2009 at 22:09
Bevan
44.3k ● 10 ● 84 ● 133

3

As long as the variable is local to the method it is thread-safe. If it was a static variable, then it wouldn't be by default.

```
class Example
{
  static int var1; //not thread-safe

  public void Method1()
   { int var2; //thread-safe
   }
}
```

Share Improve this answer Follow

answered Mar 24, 2009 at 22:10

Mark Cidade
99.8k ● 33 ● 229 ● 237

1

While your int i is thread safe, your whole case is case is probably not thread safe. Your int i is, as you said, thread safe because each thread has its own stack trace and therefore each thread has his own i. However, your thread all share the same database, therefore your database accesses are not thread safe. You need to properly synchronize your database accesses to make sure that each thread will see the database only at the correct moment.

As usual with concurrency and multithreading, you do not need to synchronize on your DB if you only read information. You do need to synchronize as soon as two thread will try to read/write the same set of informations from your DB.

Share Improve this answer Follow

answered Mar 24, 2009 at 22:12

LordOfThePigs
11.3k ● 7 ● 47 ● 72

1

i will be "thread safe" as each thread will have it's own copy of i on the stack as you suggest. The real question will be are the contents of getFromDb() thread safe?

Share Improve this answer Follow

answered Mar 24, 2009 at 22:12

ScottS
8,543 ● 5 ● 31 ● 50

---

i is a local variable so it's not shared state.

1

If your getFromDb() is reading from, say, an oracle sequence or a sql server autoincrement field then the db is taking care of synchronization (in most scenarios, excluding replication/distributed DBs) so you can probably safely return the result to any calling thread. That is, the DB is guaranteeing that every getFromDB() call will get a different value.

Thread-safety is usually a little bit of work - changing the type of a variable will rarely get you thread safety since it depends on how your threads will access the data. You can save yourself some headache by reworking your algorithm so that it uses a queue that all consumers synchronize against instead of trying to orchestrate a series of locks/monitors. Or better yet make the algorithm lock-free if possible.

Share Improve this answer Follow

answered Mar 24, 2009 at 22:28

Shea
11.2k ● 2 ● 20 ● 21

---

i is thread safe syntactically. But when you assign instance variable's values,instance method return value to i, then the shared data is manipulated by multiple threads.

1

Share Improve this answer Follow

answered Apr 21, 2009 at 8:06

javadev