**Data Encapsulation**

One of the approaches in object-oriented programming is to use data encapsulation (sometimes called data hiding) to allow objects to strictly control access to the data within.

Some data is 100% private for internal use. Other data is accessible to the “outside world” as either public instance values or (in the better case) via public access methods (getters and setters). The use of private data also allows you to create effectively immutable objects (simply don’t provide a setter beyond the constructor).

If you are using locks to assure mutual exclusion in how your program accesses an object, it’s even more important that you don’t accidentally provide direct access to information stored inside the object.
**Publication**

If we “publish” information from within a complex object, we are making it directly available to others outside of that object.

Two common examples are:

- Allow a mutable object to be passed in and rather than store and manipulate a copy of that object, only keep a *reference to the object that was passed in*.
- When a getter method needs to return the information stored in some internal mutable object, return a *reference to the internal object* rather than a reference to a copy of the object.

**Escape!**

If we publish a reference to something that we didn’t really mean to do that, we say that reference has escaped!
Publication and Escape of this

In multi-threaded programming, there are also potential issues related to publishing this during an object’s construction.

Examples: If part of the constructor’s job is to…

• …store a reference to the current object somewhere that it can be seen by other threads.
• …start a new thread running.

First, an example with an “obvious” escape of this (publication before you really wanted it out there).

The wind-up...

```java
public class ContrivedGlobalClass {
    static public ContrivedObjectClass
        mostRecentlyCreated = new ContrivedObjectClass();
}

public class ContrivedObjectClass {
    private Date buildTime;
    public ContrivedObjectClass() {
        ContrivedGlobalClass.mostRecentlyCreated = this;
        buildTime = new Date();
    }
    public Date getDate() {
        return buildTime;
    }
}
...and the pitch!

```java
public static void main(String[] args) {
    int errorCount = 0;
    int iterations = 10000;
    Thread T1;

    for (int i=0; i<iterations; i++) {
        T1 = new Thread(new Runnable() {
            public void run() {
                new ContrivedObjectClass();
            }
        });
        T1.start();
        if (ContrivedGlobalClass.mostRecentlyCreated.getDate() == null) {
            errorCount++;
        }
    }
    System.out.println(errorCount);
}
```

What does a constructor actually do, to what, and when?

In the previous example, moving the `buildTime = new Date();` line so that it was before we had the variable reference of `ContrivedGlobalClass.mostRecentlyCreated` being set to `this`, we’d still need to deal with the fact that technically speaking, the constructor wasn’t done yet (it had not returned control back to the thread that was creating the object).

Might we have to deal with the JMM and the “happens-before” rules for data races and that fact that while it might be a one in a million chance, it is still a possible data race…

What about how Java actually deals with the creation of an object. We know that it automatically calls the constructor as part of that process, but what are the specifics?

What if there is a class that extends a class that has something like this in the base constructor?
Based on Java Concurrency in Practice (Page 51)

public class Holder {
    private int value;

    public Holder(int valueIn) {
        value = valueIn;
    }

    public void assertSanity() {
        if (value != value) {
            throw new AssertionError("Value is not itself!");
        }
    }

}

Have a method with a shared reference to a Holder object, starts a thread that creates an object and sets the shared reference to point to it while the method waits for the pointer to be non-null and then calls assertSanity() using that shared reference.

What might an execution trace look like?

```java
static private Holder obj; //shared reference
public static void main(String[] args) {
    obj = new Holder(1);
    Thread T1;

    T1 = new Thread(new Runnable() {
        public void run() {
            obj.assertSanity();
        }
    });
    T1.start();

    obj = new Holder(5);
}

Possible partial execution trace (let OBJ be the actual “5” Holder):
spawn(Main, T1);
write(Main, OBJ.value, 0); //default value assigned by Java
read(Main, valueIn, 5);
write(Main, OBJ.value, 5);
read(T1, obj.value, _);
read(T1, obj.value, _);
```
The problem here is one of visibility...

spawn(Main, T1);
write(Main, OBJ.value, 0);
read(Main, valueln, 5);
write(Main, OBJ.value, 5);
read(T1, obj.value, _);
read(T1, obj.value, _);

The main thread waited around for the object “5” Holder OBJ to be constructed before assigning obj to refer to it. Let’s assume memory writes happen so that T1 sees obj as a reference to the “5” Holder object.

Is there anything in the happens-before rule set that says the write into the value field of that object happens-before either of thread T1’s reads? No, there isn’t!

So, there is a non-zero probability that the second blue write becomes visible to thread T1 BETWEEN the two green reads, meaning the first could get a 0 and the second could get a 5.

Publishing this to an event handler:

```java
double[] getEvents() {
    return (Event[]) list.toArray(new Event[0]);
}
```

Mutable, Immutable, Effectively Immutable

There is a difference between what we might refer to as **effectively immutable** (no setter methods) and **actually immutable** (no setter methods but also **final** data members and a **safe** constructor).

The nuances to being **actually immutable** are important. In fact, all of the data being **final** helps us when writing a safe constructor!

---

What if you want to publish this when you build it?

We can accomplish that goal in a safe manner via the use of the “factory method” design pattern and the use of **final** fields.
**Factory Methods**

The basic idea behind the factory method is that the constructor is not actually available to the outside world.

Because of this, you can’t just create a new object the normal way.

Instead, you invoke a static “factory method” that creates the new object and then it publishes what is effectively “this” to the outside world if you want that ability…

---

**What if you want to publish this when you build it?**

Why might we still need `final` fields?

*Data races!*

Even with the factory method, there is still a non-zero chance that what happened won’t be visible to other threads “yet” when its done.

However, when they were fixing the Java Memory Model around a decade ago, they added a rule that said once a `final` field is set in the constructor, that information had to be pushed out to shared memory by the time the constructor returned its reference.
```
NOTE: This example fixes THREE issues...

```public class FixedContrivedObjectClass {
final private Date buildTime;

    //Make the constructor PRIVATE !
private FixedContrivedObjectClass() {
    buildTime = new Date();
}

public static FixedContrivedObjectClass newInstance() {
    FixedContrivedObjectClass newObj =
    new FixedContrivedObjectClass();
    FixedContrivedGlobalClass.mostRecentlyCreating = newObj;
    return newObj;
}

public Date getDate() {
    return buildTime;

}
```
**Issues with reflection...**

It turns out that the `java.lang.reflect` library actually allows you to access and even change a private field of a class object!

If you want to really avoid publishing something, it seems like you also need to write a `SecurityManager` to restrict what reflection can be done (I’ve only read about this particular issue).

NOTE: There are certain legitimate reasons to look at private fields, like if you want to write an external serializer…