L24: Quiz 2 Review

Material from Quiz 1

Quiz 2 is cumulative; it will focus on new material since Quiz 1, but all the class material so far is in scope. The lecture on version control right after Quiz 1 won’t be tested, nor will we test you explicitly on the material from the Laziness/Streams or Program Verification lectures.

Here's one take on the big-picture idea map for the first half of the class. See the Quiz 1 review notes for more detail.

- How should we write individual methods?
  - **Static checking**: Types specify the contract of a method, in a way that is easy for the compiler to check for us.
  - **Testing**: Write test cases to exercise aspects of behavior that go beyond what types express.
  - **Specifications**: Write semi-formal English language descriptions of what a method expects (precondition), what it changes (frame condition), and what it guarantees to callers (postcondition).
    - Examples of specifications shared across many classes include the contract for equals() and hashCode() in Java.

- How should we think about the behavior of classes?
  - **State machines**: A notation for understanding the behavior of sequences of method calls on mutable objects
  - **Abstract data types**: An approach to separating interface (what does the class promise to its users?) and implementation (how is this promise achieved?)
  - **Recursive datatypes**: An alternative to state machines: represent values immutably and compute with them in an algebraic style.

- **Design patterns**: common techniques for writing effective code
  - **Regular expressions & grammars**: notations for processing text in a variety of languages, with associated implementation techniques
  - **Interpreter pattern**: one interface containing several useful operations, each implemented for each class that implements the interface
  - **Visitor pattern**: one minimal interface, containing a few methods sufficient to implement many different functions by case analysis over the different classes that implement the interface
The new material since Quiz 1 divides neatly into two categories: 
concurrency and functional programming.

Concurrency

What is it?
Multiple virtual processors running at the same time, sharing some resources, 
like memory. We often use the term processes for virtual processors that do 
not share memory, and the term threads for virtual processors that do share 
memory.

Two popular modes of concurrency

• **Shared memory:** threads interact with each other by mutating a 
common memory, which can be a Java “heap” like the object diagrams 
we have been drawing all along.

• **Message passing:** threads interact by sending messages over a 
“network,” which could be purely local or could be the actual Internet 
as in PS3 and PS4.

How do we implement concurrency?
Even if there is only a single physical processor, we can still run multiple 
threads by **interleaving**, where we alternate between running computation 
steps in the different threads. True parallelism, with multiple physical 
processors, can be modeled in a very similar way. Thus, much of the 
challenge of reasoning about concurrent programs comes from thinking hard 
about all possible interleavings.

What can go wrong?
Oh so many things! Here a few that are common enough to have names:

• **Race condition:** when the correctness of the program depends on the 
relative timing of events in concurrent computations [Example: bank 
account balance manipulation on L15]

• **Deadlock:** a group of threads are all paused waiting for the others to 
do something, like release a lock [Example: transferring money 
randomly across bank accounts around L16]

Why are concurrent programs hard to debug?

• **Large state space:** interleaving creates many possible control flow 
paths through a single program; hard to build test cases that push us 
along all paths.

• **Nondeterminism:** program is allowed to follow different paths on 
different runs; one test case may trigger many different paths on 
different runs, only one of which exhibits a bug.

How can we avoid bugs in concurrent programs?
We studied four main techniques:
• **Confinement:** make state *thread-local*, so that obviously other threads can't interact with it in confusing interleavings.

• **Immutability:** confusing interleavings of modifications to state aren't possible if the state isn't allowed to change! (But be careful with naïve ideas of when concurrent implementations are suitably immutable.)

• **Using threadsafe datatypes:** often the tricky issues of concurrency can be encapsulated inside abstract datatypes, which is a good reason to use library classes of this kind. It is even possible to use *wrapper* classes that “add thread safety” to your own simpler classes (in some cases, and often with a performance penalty).

• **Explicit synchronization:** use *locking* in your code, either explicitly with lock objects or taking advantage of the Java *synchronized* keyword. This one is important enough to deserve its own section.

**Locking**

It is possible to program with explicit *locks* in Java:

```java
Lock lock = new SomeLockClass();
lock.lock();
/* do stuff with objects considered “protected by” the lock */
lock.unlock();
```

However, one has to be very careful with this pattern of coding, because an *exception* raised may cause execution to jump past the *unlock*. A safe idiom would be:

```java
Lock lock = new SomeLockClass();
lock.lock();
try {
   /* do stuff */
} finally {
   lock.unlock();
}
```

That is, we ensure that the *unlock* runs as part of any path out of this code, even if exceptions are raised.

This sort of careful coding is so important that Java includes a keyword that implements it automatically. Furthermore, every Java object includes a built-in lock. So we can often just write:

```java
synchronized (this) {
   /* do stuff */
}
```

Such code implicit takes and releases the lock associated with *this*. 
Locking can be done at many different granularities. Coarse-grained locking might associate a lock with all accesses to objects in a particular subsystem of your program, while fine-grained locking might associate locks with individual objects within that subsystem. The important rule to follow is whenever there is a rep invariant relating a number of objects, they must all be protected by the same lock.

Coarse-grained locking reduces parallelism and performance, but it is much easier to reason about than fine-grained locking, so we recommend it as the right default. The monitor pattern is an approach to coarse-grained locking that Java supports well. We merely annotate methods as synchronized when they might otherwise be involved in concurrency bugs. All such methods implicitly lock and unlock the current object when called.

When fine-grained locking is necessary, it is important to think hard about avoiding deadlock. Recall the example of wizards friending and defriending each other in L18. Each such wizard will take a lock on himself and then on his new friend. If two wizards are trying to friend each other simultaneously, deadlock can result. Our solution was to impose a lock ordering, where a thread requesting multiple locks must always request them in a well-defined order, which avoids cycles in the graph of who is waiting for whom to release a lock.

**Thread Safety Arguments**

Testing can sometimes find concurrency bugs, but it takes a lot of patience, due to the nondeterminism of interleaved thread execution. A better approach is to write a thread safety argument as a comment in your code, so anyone doubting your claim can read it, much as this person could run your tests.

Such an argument considers all threads in the program and all the data objects they might access. Each data object needs an argument naming one or more of the techniques for avoiding concurrency bugs, along with sufficient detail to convince an informed reader that you have applied the technique(s) correctly.

**Graphical User Interfaces**

We lumped this topic in with concurrency, though it has some independent aspects, too. The main take-aways:

- **View tree** as a unifying abstraction for how to lay out GUI elements. We can combine individual widgets into composite widgets that are then treated in the same first-class way. Each widget in the tree has its own bounding box in space, within which further layout can take place.

- **Event listeners** allow code to register its interest in events (e.g., mouse clicks) that happen to widgets (e.g., buttons). Registration involves passing off an object with a method that will be called with event data when the event fires. Event handlers open up many exciting possibilities for bugs when mixed with mutable data structures for widgets.
• **Background processing in GUIs** requires some care. For instance, the underlying data structures of Java Swing are not thread-safe, so it is important to use `SwingUtilities.invokeLater()` to queue code to run in the single designated GUI thread.

• The **model-view-controller pattern** suggests a separation of concerns in GUI app implementation. The naïve version says that a *model* consists of your application’s underlying data, a *view* presents (part of) that data as GUI elements, and a *controller* handles user input to modify the model and thus change what the view displays.

### Functional Programming

Our treatment of concurrency is all about understanding complex processes that manipulate mutable data. At the other end of the spectrum is *functional programming*, where most code avoids mutation entirely. We touched on functional programming in the first half of the lectures, in our discussion of recursive datatypes, the visitor pattern, and so on.

The second time around, our focus is on two notions of **first-class** pieces of programs that functional programming supports especially well. These are **first-class functions** and **recursive datatypes with the primitive-composition-abstraction pattern** (PCAP).

We also used examples in a few other languages (Python, Scala, OCaml) beside Java to illustrate functional programming ideas with less syntactic clutter. We won't quiz you on the details of syntax, type-checking, etc. in any of these languages.

### Functions as Data

Writing useful numeric programs would be hard if we couldn't pass, say, integers around as function arguments and return values. Because we can do these things, we call integers in Java (and most other languages) **first-class**. Functions are also useful when they get first-class treatment, and such treatment is one of the distinguishing characteristics of functional programming.

This first-class support enables programming with **higher-order functions**, which are functions that take functions as input, return functions as output, or both. Some of the most popular examples are a trio that we focused on in L20:

- **map(f)**, which takes in a list $a_1, \ldots, a_n$ and outputs $f(a_1), \ldots, f(a_n)$
- **filter(f)**, which takes in a list and outputs only those elements that pass the Boolean test $f$
- **reduce(f)**, which takes in a list $a_1, \ldots, a_n$ and outputs $a_1 f \ldots f \ldots a_n$, where we write $f$ like an infix operator

With these powerful building blocks, we can write many useful list transformations, without any mutation and often even without defining any local variables to store intermediate results.
Primitive-Composition-Abstraction Pattern

First-class functions are great, but there are only so many things code can do with a function. The only option is to call it with an argument. Sometimes we would like to look inside a function-like value, to compute properties that don't follow just from any finite input-output behavior. The Primitive-Composition-Abstraction Pattern from L21 suggests an approach for such cases.

- Come up with an explicit language of “programs” represented with a recursive datatype. (E.g., music type from L21)
- Include some primitives, which are constructors of your datatype that do not take arguments in the same datatype. (E.g., “play a note” and “rest”)
- Include some composition constructors, which take other instances of your datatype as arguments. (E.g., “play this music followed by this music”)
- Implement some basic operations over this datatype, as recursive functions that descend through the structure of a value. (E.g., duration function for music)
- Build some abstractions on top of your datatype, which are functions taking in and/or returning values in that type. (E.g., counterpoint for music, which takes in a piece of music and returns a piece that plays several variants of the original at once)

Sometimes the abstractions may be higher-order functions, but sometimes they just manipulate first-class values in your recursive datatype. Either way, the advantages are similar to what we see with map/filter/reduce. We can write some very concise code that accomplishes complex tasks without tedious manipulation of local variables holding intermediate values.