OO Program Design in Java

Corky Cartwright
Department of Computer Science
Rice University

Functional Programming

- Functional programming in Java
  - immutable data
  - algebraic (inductive) data represented using the composite pattern
  - functional methods defined using the interpreter pattern
  - essentially the same design recipe as functional programming in Scheme
  - classes with a single instance: singleton pattern
  - functions as data represented by singleton classes with a single `apply` mehod (really should be called the first-class function pattern)
  - strategy/command pattern supports passing first-class functions to methods
  - closures (with refs to method vars) can be passed as anonymous classes
  - extending composite pattern adds hooks supports the visitor pattern
  - visitors are first class functions with union pattern structure so inheritance is possible
  - static methods are oblivious to class instances (not supported in LL)
  - static final fields are program constants
  - exceptions and exception-handling using catch

The Design Recipe for Java

How should one go about writing programs?

- Analyze problem, which includes:
  - defining any classes `C` for data types that are not primitive;
  - determining what visible methods should appear in each class.
- For each visible method **`m`** in each class `C` :
  - Write the header and contract (HTDP: purpose) for **`m`**.
  - Create a test class for `C` (or the set of tightly coupled classes including `C` if it does not already exist) and write a test method for **`m`** that checks it behavior on representative inputs.
  - Select and instantiate a template for the method body and primary argument (**`this`**). Define auxiliary helper methods as needed, and add them to the class **C** containing **`m`**.
  - Code the method by filling in the template
  - Run the tests and confirm that they succeed.

Functional (LL) Java Restrictions

- No mutation (re-assignment) ESSENTIAL
- No loops (can't do much if no re-assignment)
- No synchronization ESSENTIAL
- No visibility modifiers
- No explicit constructors
- Only exception to above is private constructor for singleton pattern
- No static methods
- No overloading
- Static final fields
- Only inner classes are anonymous classes
- Exceptions but no catch

Generalizing FP to Include State

Motivations

- Modeling objects that change over time
- Modeling naturally cyclic structures
- Caching computed results (benign optimization of functional semantics; invisible to client)
  - Lazy evaluation (suspensions are replaced by values on demand; supports infinite streams)
  - Memoization (a table of suspensions mapping function inputs to outputs)
- Efficient algorithms often use mutable state.

Characteristics

- Mutable object fields
- Mutable method variables
- State pattern: an object's principal field is mutable with union/composite type; the possible states are the variants of the union/composite
- OO style dictates the disciplined use of mutation
- Never modify fields in other objects directly.
- Support high level mutation via mutating methods.

Full Java

- Static methods
- methods can be overloaded
- constructors must be explicit, except for 0-ary constructor
- visibility must be specified explicitly
- generic classes, polymorphic methods
- equality must be defined explicitly (and hashcode overridden if class is immutable)
- accessors must be defined explicitly
- catch clauses
- mutation allowed
- loops
- Inner classes (static and dynamic)
- synchronization; wait/notify
- Static initialization blocks

Static Type Checking

- Compiler checks declared type information for consistency.
- Generally intuitive except for lack of covariant generic subtyping, List<String> ! <: List<Object>
- Important gotchas:
  - Static types govern overloading
  - Static types govern field selection.

Generics in a Nutshell

- A generic class/interface is a class/interface parameterized by one or more type variables `T`, `U`, …, *e.g.* `List<T>`
- Within a generic class, type variables can (almost) be used like conventional types.
- Prohibitions: **new T()**, `new T[()]`, … `instanceof T`, … `instanceof Foo<T>`, generic exception classes.
- Warnings (type safety is lost) `(T) …`, `(Foo<T>)` ...
- Outside a generic class, clients always refer to instantiations of the class, *e.g.* `List<Integer>, List<E>` where `E` is any type formed from contants and type variables in scope.
- Inside the scope of a generic class, the type variables of all enclosing classes are in scope.
- Note: a static inner class is NOT within the scope of the enclosing class.
- Generic subtyping is *non-variant* (*invariant*): `C<S> <:` (is a subtype of) `C<T>` Iff `S = T`. But it respects erased (ignoring parameterization) class subtyping: `A<S> <: B<T>` Iff `S = T` and `A <: B`

Common Algorithms and Data Structures

- Generally imperative (probably too imperative) because they evolved in the context of RAM instruction sets and higher level language supporting obvious abstractions of patterns of machine instructions.
- Important to recognize portions of algorithms that can be made functional without asymptotic performance loss.
- Important forms of data: machine primitives, algebraic data types including lists, arrays, tables (finite sets, finite functions), first-class functions (!), first-class set formulations other than lists (advanced).
- Sorting lists is critical (fastest technique for int keys: radix).
- Fast table searching is critical. Hashing is often the best solution.

Concurrency and Parallelism

- Concurrency and parallelism both involve the notion of independent computations (threads) that potentially run concurrently.
- In concurrent programming, these concurrent tasks are not directly related to one another but the often incidentally share data structures.
- Concurrency arises from:
  - asynchronous tasks (computations) created by a GUI
  - Multiple input agents (different GUI buttons, different remote computers), e.g an airline reservation system. Creating applications for a network naturally forces them to support concurrency
  - Improved performance is not the fundamental concern
- Parallelism involves dividing what is in principle a single computation into multiple tasks (threads) to improve performance. The client of a parallel application does not know whether the computation is being performed using explicit parallelism or not. (Note: computer hardware includes lots of internal parallelism to improve performance.)

GUI Programming

- GUI programming is the most common form of concurrency.
- A simple GUI involves two threads but the do not access any shared data structures and their execution only overlaps incidentally.
- Well-written GUI programs can typically be decomposed into three components and two or more (often varying as the program executes) threads.
- The main thread (the thread that runs when the program is started) executes the component called the *controller*, which creates and initializes the other two components: the *view*, or the GUI interface, and the *model*, or the core application program, which in simple cases is not even aware that the GUI and controller exist. The application supports a programming interface (API) which only knows/understands the values passed across that API.
- After creating and initializing the view and the model, the controller starts an event thread associated with the view (written as part of the GUI library) and immediately dies. The only period when the two threads overlap is the interval between the main thread starting the GUI thread and dying. And this overlap is utterly inconsequential because the main thread does not interact with the GUI during this period.

Canonical Simple GUI program

- Study the ClickCounter program in the course notes on OO Design.
- It follows the simple scenario described on the previous slide.
- Note that the key part of the code is the controller code that installs listeners linking the model and view so that the processing of view events performs appropriate operations on the model.
- Another potential example: the Laundry Program In HW10.
- BUT, I designed this program in the fall of 1996 when I was just learning OO design. I did not look carefully at the design of the program carefully again until this year because after 1998 we limited it to the console interface.
-

Rethinking the Laundry Program

The driving loop for the program is the simulate function in the Student class. If the input to the program is a simple text stream, then this design works well. The simulate method repeatedly asks for Input and blocks if no input is available yet.

The input text processor (the surrogate GUI for the text version of the program) reads ASCII characters from the input stream and aggregates them into command objects. The simulate method sees a stream of commands.

But a good GUI interface allows the user to stipulate in the middle of an event stream that input should be taken from a file, so the GUI has to accumulate input commands and buffer them delivering them to the simulator (the simulate method in the Student class) on demand.

If the simulate operation simply executed a single command (passed to the simulate method) then the simulate method would be significantly simpler because it would only need to process a single command (using a case split on the command),

More Challenging Examples

- The best known solutions to many standard computational problems can be formulated as the memoization of naïve solutions.
- Memoized algorithms correspond to a problem solving technique called *dynamic programming*.
- Examples:
  - parsing CFGs (CYK algorithm),
  - optimizing the multiplication of a chain of matrices
  - shortest path between two nodes in a graph, …
  - many string algorithms
- Lots of information on the web on dynamic programming

Rethinking the Laundry Program

The driving loop for the program is the simulate function in the Student class. If the input to the program is a simple text stream, then this design works well. The simulate method repeatedly asks for Input and blocks

- The input text processor (the surrogate GUI for the text version of the program) reads ASCII characters from the input stream and aggregates them into command objects. The simulate method sees a stream of commands.
- But a good GUI interface allows the user to stipulate in the middle of an event stream that input should be taken from a file, so the GUI has to accumulate input commands and buffer them delivering them to the simulator (the simulate method in the Student class) on demand.
- If the simulate operation simply executed a single command (passed to the simulate method) then the simulate method would be significantly simpler because it would only need to process a single command (using a case split on the command),

For Next Class

- Exam II over OO material will be given at scheduled site on Friday, April 30.
- Parallel sudoku homework is due today at 11:59pm. Enjoy.

Anonymous Inner classes in Java (Lecture 31, slide 9)

```
public void start(final double rate)
{
      ActionListener adder = new
            ActionListener()
           { // anonymous inner class that implements ActionListener interface
               public void actionPerformed(ActionEvent evt)
               {
                      double interest = balance * rate / 100;
                      balance += interest;
               }
           };
      Timer t = new Timer(1000, adder);
      t.start();
       . . .
}
```

- This is saying, construct a new object of a class that implements the ActionListener interface, where the one required method (actionPerformed) is defined inside the brackets.

Java’s Callable Interface (Lecture 31, slide 14)

- Introduced in J2SE 5.0 in java.util.concurrent package (remember to “import java.util.concurrent;”)

```
public interface Callable<V> {
     /**
      * Computes a result, or throws an exception.
      *
      * @return computed result
      * @throws Exception if unable to compute a result
      */
     V call( ) throws Exception;
}
```

Task Decomposition using Callable (Lecture 31, slide 15)

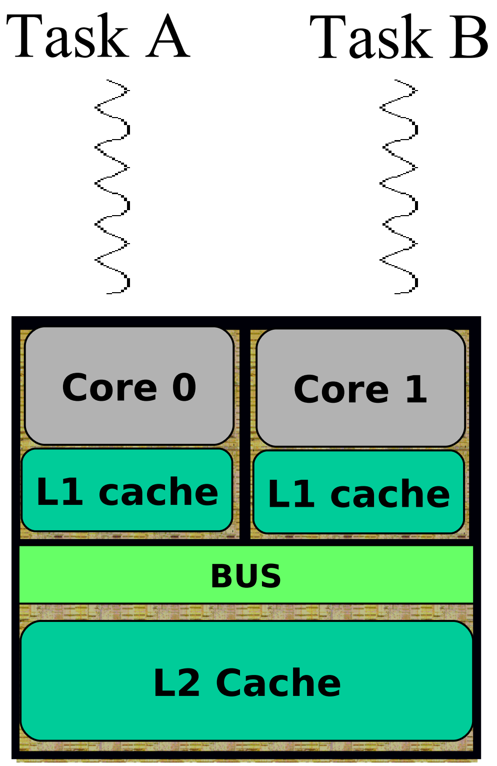
```
// HTML renderer before decomposition
   ImageData image1 = imageInfo.downloadImage(1);
   ImageData image2 = imageInfo.downloadImage(2);
   . . .
   renderImage(image1);
   renderImage(image2);

// HTML renderer after task decomposition
   Callable<ImageData> task1 = new Callable<ImageData>() {
     public ImageData call() {return imageInfo.downloadImage(1);}};
   Callable<ImageData> task2 = new Callable<ImageData>() {
     public ImageData call() {return imageInfo.downloadImage(2);}};
   . . .
   renderImage(task1.call());
   renderImage(task2.call());
```

From Sequential to Parallel Task Decomposition (Lecture 34, slide 18)

Key Observation:

If two *functional* tasks can be executed in any order, they can also be executed *in parallel*



Schematic of a Dual-core Processor

How can we express Task Parallelism in Java?

- Answer: there are many ways, but they all ultimately involve execution on Java *threads*
- The Java main program starts as a single thread
- The code executed by the main thread can create other threads
  - Either explicitly (as in the following slides); or
  - Implicitly via library use:
    - AWT/Swing, Applets, RMI, image loading, Servlets, web services, `Executor` usage (thread pools), …

Executing a Callable task in a parallel Java Thread (Lecture 34, slide 20)

```
// 1. Create a callable closure (lambda)
Callable<ArrayList<Integer>> left_c = …
// 2. Package the closure as a task
final FutureTask<ArrayList<Integer>> task_A =
    new FutureTask<ArrayList<Integer>>(left_c);
// 3. Start executing the task in a parallel thread
new Thread(task_A).start();
// 4. Wait for task to complete, and get its result
left_s = task_A.get();
```

Quicksort with Parallel Tasks (Lectures 34 & 39)

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
     if (a.isEmpty()) return new ArrayList<Integer>();
     final ArrayList<Integer> left = new ArrayList<Integer>();
     final ArrayList<Integer> mid = new ArrayList<Integer>();
     final ArrayList<Integer> right = new ArrayList<Integer>();
     int pivot = a.get(a.size()/2); // Use midpoint element as pivot
     for (Integer i : a)
         if ( i < pivot ) left.add(i); // Use element 0 as pivot
         else if ( i > pivot) right.add(i);
         else mid.add(i)
     // Now, left, mid, right contain the three partitions of
     // array a with respect to pivot
     // Continue on next slide ...
```

Quicksort with Parallel Tasks (contd) (Lectures 34 & 39)

```
FutureTask<ArrayList<Integer>> left_t = // Closure for recursive call
  new FutureTask<ArrayList<Integer>>(
    new Callable<ArrayList<Integer>>() {
      public ArrayList<Integer> call() { return quickSort(left); } } );
FutureTask<ArrayList<Integer>> right_t = // Closure for recursive call
  new FutureTask<ArrayList<Integer>>(
    new Callable<ArrayList<Integer>>() {
      public ArrayList<Integer> call() { return quickSort(right); } } );
// Execute each closure in a parallel thread
new Thread(left_t).start(); new Thread(right_t).start();
// Wait for result of FutureTask's left_t and right_t
ArrayList<Integer> left_s = left_t.get(); // Sorted version of left
ArrayList<Integer> right_s = right_t.get(); // Sorted version of right
return left_s.addAll(mid).addAll(right_s);
} // quickSort
```

Summary of Lecture 39

- Trade-offs in Parallel Programming
  - Overhead
  - Memory
  - Serialization
- Computation Graph, Total Work (T_1), Critical Path Length (T_∞)
- Lower bounds in Computating Graph
  - $T_P \geq T_1/P$
  - $T_P \geq T_\infty$
- Amdahl's Law (for serial fraction, f_S, and parallel fraction, f_P)
  - $T_P >= f_S * T_1 + f_P * T_1 / P$

Life beyond COMP 211

- Computer Science has a lot to offer
    - Help solve major challenges facing the world
        - Energy crisis, cancer prevention, globalization, …
    - Work on intellectually stimulating problems
        - Data mining, dynamics of social/financial networks, computational science, …
    - Vast choice of career options
        - Animation, Design, Finance, Law, Medicine, Software, …
- Talk to your major advisor!