CSE 331 Software Design and Implementation

Handout C1: ***Differences from CSE 143***

Contents:

Introduction

The only prerequisite for CSE 331 is CSE 143. CSE 143 provided an introduction to programming in Java. You will find that foundation very useful. CSE 331 will take you to the next level in terms of programming ability. In order to do so, you will need to change your study habits, the way you think about programming, and re-learn a few technical concepts.

Study habits

CSE 142 and 143 are taken mostly by students who will not become CSE majors. The classes are calibrated so that the average UW student can get by. The average CSE major is much more capable, and CSE classes teach much more; thus, CSE classes are much more challeging. We consistently hear from students - even very good ones - who are taken by surprise by the workload in every 300-level CSE class, including 331. Don't be discouraged, and don't feel that you are alone in feeling this way! We will help you get through the class.

We strongly encourage you to seek out help from the course staff and from other students. You can discuss any aspect of the class, but must do and submit your own work (see the collaboration policy). In CSE 143, some of the better students never need to ask for help. Don't be shy about asking for help in CSE 331.

You will be expected to show more independence in CSE 331. Don't expect all the answers to be pre-packaged for you. For example, there are no "cheat sheets" with a list of all the methods that you might need - rather, you are expected to refer to the Java documentation, just as professional programmers do. Or, do a web search, which is one of the best developer tools you will ever have - not for copying answers, of course, but for diagnosing error messages, learning about an API you didn't know, etc. As another example, when you ask a question, we expect you to explain what you have already tried. Although we expect you to make an honest effort to answer your own questions, we don't want you to get stuck. If you are no longer making progress, seek help! We will be happy to get you on the right track.

Attitude toward programming

A common refrain in CSE 142/3 lectures is that to determine what a piece of code does, "the only way to find out is to try it." Experimenting with your code is a great tool for building insight, and you should do this. Furthermore, in small programs, it's remarkably effective. But it isn't the only way to figure out what your code does, and should never be the first thing you try, especially for more substantial programs. We expect you to develop and use a variety of techniques for understanding and debugging your programs. If you already have a hypothesis regarding the result when you try an experiment, you will learn from that experiment. By contrast, if you just randomly try things, you may never learn why some work and some do not, and you will never become an effective programmer.

CSE 142, especially, treats object-oriented code and data abstraction with suspicion. Object-orientation is the world's dominant programming paradigm, for good reason: it offers powerful mechanisms for effective system-building and for solving challenging design problems. In certain cases other paradigms are better - Java and OO certainly are not perfect for every problem. But you should embrace and understand them, because doing so will make you a better programmer whether you are working in an OO, procedural, functional, logic, or other language. Most of the lessons of CSE 331 will carry over to these other domains.

As a related point, you should only rarely use `static` methods and (especially) variables.

Technical concepts

Data structures

CSE 143 places a great deal of emphasis on tree data structures such as `TreeMap`. Learning to implement this is a fine exercise, but in practice you will rarely use it, because `HashMap` is usually a better choice. Java programmers use `HashMap` more than 4 times as often as `TreeMap`.

Use of `HashMap` and `HashSet` require correct definition of `equals()` and `hashCode()`, just as correct use of `TreeMap` and `TreeSet` require correct definition of `compareTo()` or a `Comparator`. It is important that you understand all of these methods.

You have gotten a lot of practice with manipulating arrays and with re-implementing existing data structures. If you are lucky, you won't ever have to do those again! Lists are usually a better choice than arrays, and the data structure implementations in the standard library (the JDK) are much better than anything you are likely to code up. Besides, it's much more satisfying to solve new problems than to re-solve old ones.

Loops

Previous classes encourage you to write for loops with an explicit index, such as `for (int i=0; i<max; i++) { Object elt = mylist.get(i); ... }`. Even when you are working with arrays it is clearer and less error-prone to use a "foreach" loop: `for (Object elt : mylist) { ... }`. It's poor style to use an explicit index when you do not need it. You will need explicit indices only occasionally.

Infinite looping is a property of program behavior. It is not a syntactic property. A loop of the form

```
while (true) {
  ...
  break;
}
```

is an infinite loop if it never executes the break statement at run time. It is not an infinite loop if it executes the break statement. An accurate name for a loop whose number of iterations is not predictable ahead of time is an *indefinite* loop.

Parsing input

You should never call both `next` and `nextLine` on a single `Scanner` object -- if you do, the `Scanner` may misbehave. For this reason, and also because most input operations are line-oriented rather than word- or token-oriented, use of `Scanner` is uncommon in practice. It's more common to read an entire line, and then if necessary parse it - possibly using a `Scanner`, or `String.split`, or some other mechanism.

Method call semantics

Java supports only call-by-value. It never uses call-by-reference, nor "reference semantics" (whatever that term means). Every Java value is either a primitive (of type `int`, `bool`, etc.) or a reference. These values are passed by value. A consequence of this is that if the method body assigns to the formal parameter, as in `void foo(Date d) { d = new Date(); }`, then no change is visible to the caller. However, when a procedure receives a reference as an argument, the procedure is able to mutate the referent (the pointed-to object) via the formal parameter, as in `void foo(Date d) { d.setYear(2013); }`.

Back to Conceptual Info
Back to the CSE 331 home page.