Computer Science II

(**version by Ilkka Kokkarinen, April 29, 2020**)

This document describes the author's idea for a good and solid second course in computer science and programming, using Java as the programming language. It is based on the author's long experience in teaching the course *CCPS 209 Computer Science II* for the Chang School of Continuing Education, Ryerson University. However, this document does not describe the policies of CCPS 209 as it is currently offered in Ryerson University and the Chang School that uses a different grading scheme than the one given below as a suggestion for the readers interested to learn this material by self study and practice. Always consult the official documents for the course CCPS 209 for the current implementation of the course.

Outline

This is a second course on computer programming, taught using Java 8 programming language. This course is intended for students who have completed the first university-level course on computer programming. In this course material, that first language is assumed to be Python, the current introductory programming language used in Ryerson computer science. However, that first language could also be some other language that allows students to solve problems with conditions, loops, lists and other basic imperative programming structures, assuming that these problems are small enough and do not need to maintain any additional state that persists in program between the separate function calls to remember the past to the extent that it affects the future. Even though we make occasional comparisons between these two major languages, everything in this course is done in Java, and no Python knowledge is necessary for completing any of the problems in this course.

The course topics are broken down into eleven separate modules, with one more module for spillover and the last module for submission of the course lab work. The numbers listed in parentheses after the title of each module are the lab problems that completing the modules up to that point enables the student to solve in principle.

1. Imperative Java for Pythonistas (0A, 0B, 0C, 0F, 0H)
2. Classes and objects (1, 2, 29)
3. Inheritance and polymorphism (4)
4. Java Collection Framework (0D, 0E, 0G, 3, 11, 12, 15, 16, 17, 24, 25, 30)
5. GUI programming with Swing (5, 9, 13, 14, 22, 23, 27, 28)
6. Exception handling
7. I/O streams (6, 8, 26)
8. Generics
9. Concurrency fundamentals (7)

10. Concurrency controls
11. Computation streams (10)

Materials

The public GitHub repository `ikokkari/JavaExamples` always contains the latest versions of lecture notes (as PDF) and all example programs (as Java source) of this course.

The Java language itself, same as the BlueJ software needed in this course, is totally free to download and use. As programming environments go, BlueJ is excellent for beginners in its simplicity, but students who wish to work in Eclipse or similar full-fledged environments can also complete all their work there, since the Java language and its standard libraries are identical everywhere.

Students should also take an occasional look at the Oracle Java Tutorial to complement these lecture notes and examples. The most appropriate individual trails for this course would be "Learning the Java Language", "Essential Classes" and "Collections". When you are writing code for the labs, it is good to keep the Java 8 API Reference open in another browser tab to quickly look up classes and methods as needed.

Students looking for additional material and alternative points of view into these topics might also want to check out the MIT OpenCourseWare offering of 6.005 Elements of Software Construction that expands on many issues that were merely touched in this course. There are two versions of its lecture notes, a lighter and better formatted set from 2016, and a deeper set from 2018.

Labs

The public GitHub repository `ikokkari/CCPS209Labs` always contains the latest versions of the lab problems for students to sharpen their claws and fangs of the Java language and its structure, along with the latest versions of the JUnit test classes. Some of these problems are assigned as weekly problems, whereas the remaining problems are for the students to solve on their own time. There are quite a few lab problems available for the students to choose from, so no student should get stuck with any one lab for too long, and should initially skip the labs that do not fill them with enthusiasm. Both this course and life in general are too short to get mired with meaningless problems when there are countless worthwhile problems available!

The lab specifications are given in the file "CCPS 209 Labs". The numbering of these labs makes the distinction between the "zero labs" that do not require any object-oriented thinking but essentially use Java write Python-style functions that do not need to remember the past, as opposed to the "actual labs", whose numbering counts up from one, that require the use of various object-oriented programming techniques taught in this course. The numbering of these labs is not consistent with the numbering of the topics in this course, since these lab problems

are a result of long evolution of past lab problems that came and went and begat other problems. (That, and the fact that this numbering stems from some random quirks of the past that at the time felt inconsequential anyway, also sort of illustrates how all our categorizations for things are fundamentally arbitrary and impermanent.)

Do not submit your labs one by one as you complete them. All labs are due in one bunch at the end of the course, and should at that time be submitted as one BlueJ labs project folder that contains all these solutions along with the proudly included JUnit test classes. This way, the grading personnel can access and test your labs in a swift and prompt fashion with just a couple of clicks of the mouse.

Grading

The grade for this course is solely determined by the number of lab problems that the student completes successfully before the end of the course. There are no other factors involved in grading. To count as a successful submission, every required method in the lab must be implemented correctly so that it cleanly passes the JUnit test for that lab with a green checkmark. **No partial marks are given for any of the labs.** For the GUI programming labs that have no JUnit test classes, the lab must look and feel as specified to earn the mark.

For the purposes of grading, all labs have the exact same value, regardless of whether the numbering of that lab falls in the series of "zero labs" or "actual labs".

To pass this course with a grade of 50%, which corresponds to the Ryerson letter grade of D minus, the student **must complete and submit any five labs of their choice**. After the first five labs that carry the student over the threshold of passing the course, **every additional lab adds two points to their course grade.**

Another way to look at this grading scheme is to think of it as every five additional labs incrementing the course grade by one letter. For example, completing a total of twelve labs would give the student the grade of 64%, which corresponds to the letter grade of C in Ryerson. Five more labs would up this grade of 74%, which corresponds to B. To achieve the highest possible Ryerson letter grade of A+ for 90%, the student must complete and submit twenty-five labs, thus running at the average pace of a bit more than two labs each week.

Module 1: Imperative Java for Pythonistas

Introduction

All mainstream languages from the imperative programming paradigm offer pretty much the same common core of elementary statements that define the control flow inside a function.

There is always the if-else two-way decision, a general while-loop, and some kind of limited for-loop for iterating through a sequence of values that has a known beginning and end. This course assumes that a student has already taken an introductory programming course in which no problem is larger than one function that solves it, and the functions and data objects do not need to contain any state to remember the past, but every problem is small enough to be solved with one function. In Ryerson this introductory course is given in Python, a language that we will then occasionally compare and contrast the Java language with, but otherwise we do not assume any knowledge of Python.

This introductory lecture shows how to use Java in the classic imperative manner without needing to know anything about object-oriented programming and thinking in such terms. The encapsulated approach, in this course done in an object-oriented manner, is necessary in dealing with problems that are too simply too big and complex to be written and solved as one function. The basic syntax and structure of simple imperative Java programs is demonstrated with several examples that highlight different aspects of solving problems in imperative fashion.

Topics

1. Basic syntax and structure of Java, along with the standard naming and code structuring conventions. Turning a class into a standalone program with a `main` function.
2. The distinction between compile time and runtime. The advantages of voluntarily putting ourselves in the restrictive leash of explicit typing to reveal errors.
3. The fundamental imperative programming structures of `if-else`, `while` and `for` used inside static methods, which are the closest Java equivalent of Python functions.
4. Structure and syntax of a Java class that contains only `static` methods that do not have internal state to remember the past.
5. The eight primitive data types of Java and their limitations. The useful methods in the wrapper classes that exist for each one of these primitive types.
6. Homogeneous low-level arrays in Java and their basic operations as instance methods in the array objects themselves and as `static` utility methods in the `Arrays` utility class.
7. Strings in Java and their basic operations, as compared and contrasted to both Python strings and Java character arrays.

Learning objectives

After this module, students know the basic syntax and structure of Java without any object-oriented programming concepts. They can write and use the classic control statements that are equivalents of the Python structures `if-else`, `for`, `while` and `return`. They can name the eight primitive types as "elementary particles" from which higher data types are built from. They know Java arrays as low-level sequences of homogeneous elements whose length is fixed at creation, and can point out differences from the more flexible and powerful Python lists. They can solve computational problems that are small enough to be expressed as one function, in the same manner that they did in Python during the first course.

Readings

1. GitHub repository `ikokkari/JavaExamples`: The example programs `PythonToJava`, `DataDemo`, `ConditionsAndLoops`, and `ArrayDemo`.
2. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 1-1: Notation as a tool of thought", "Java 1-2: Elementary byte-icles", "Java 1-3: A stark data structure", "Java 1-4: For your i's only", "Java 1-5: Do it for a while", "Java 1-6: Uniformity is our strength", "Java 1-7: That inter-notional rag".

Summary

The features of Java language introduced in this module enable programmers to solve any problem small enough to be understood and modeled as one function and does not need any additional state to remember the past activations of that function.

Activities

1. If you don't already have a Java environment you want to work in, download and install BlueJ on your computer. (This will also automatically download and install a recent version of Java.)
2. Download the repository `ikokkari/JavaExamples` as a zip archive on your own computer, and unzip it into whatever location in your hard drive you usually place your data files.
3. Start up BlueJ, and choose "Open Non-BlueJ" to turn your `JavaExamples` folder into a BlueJ project with the project info and other files.
4. Play around a little bit with the BlueJ interactive environment to inspect and edit classes and call their methods interactively. Right-click some of the example classes `PythonToJava`, `DataDemo`, `ConditionsAndLoops`, and `ArrayDemo` to interactively call their methods and learn how to enter arguments to a method and how to examine the results.
5. The BlueJ method argument dialog box always expects a legal Java expression as an argument. When calling a method that expects a string argument, you must enter that string literal into the BlueJ argument text field with the double quotes around it, such as `"Hello world"`. To pass an entire array object as such an argument, list its elements inside **curly braces**, such as `{42, 99, -17}` for a three-element array of `int`s.
6. Realize that if you want to try out your method with some larger array, it is better to write a separate test method with that array hardcoded inside the method.
7. Download the GitHub repository `ikokkari/CCPS209Labs` as a zip archive. **However, do not turn this folder into a BlueJ project.** We will be accessing these files on an individual basis as needed. Instead, create a brand new empty BlueJ project named "`209 Labs`". **You will be writing all your labs inside this project for the rest of this course.** Whenever you are working on some particular lab, add the corresponding JUnit test

class into this project from the `CCPS209Labs` folder. This keeps the project simple and clean, instead of filling it with tons of files you don't need.

8. Read the specification for **Lab 0(A), "Arrays and Arithmetic"** in the lab specifications document "CCPS 209 Labs". To start working with this lab, create a new class `P2J1` in your BlueJ labs project. Double click the class to open it inside the editor window, and delete all the default class boilerplate "helpfully" offered by BlueJ between the curly braces of the class body.
9. For each of the four methods specified for this lab, create a **do-nothing stub version** of that method in your class `P2J1` that consists of nothing but one return statement that returns a zero, `null` or some other suitable placeholder result of the declared return type of that method.
10. Add the JUnit test file `P2J1Test` into your BlueJ project from the `CCPS209Labs` folder. Keep correcting the signatures of your method stubs in the class `P2J1` until they are exactly as specified so that the test class `P2J1` compiles cleanly and is displayed as a solid green box in your BlueJ labs project.
11. You are now ready to start solving the lab problems piecemeal by replacing your method stubs with actual working code. After you complete each method, run the JUnit test class to verify that your method returns the expected answers for the series of test cases pseudo-randomly generated by this test class. All future labs in this course will work the same way, except for the labs about GUI programming that have no automated testers but will be eyeballed instead to check that they do what is expected.

Lab problems

Complete **Lab 0(A), "Arrays and Arithmetic"** from the document "CCPS 209 Labs". Write all your method code into the same source code file `P2J1` in your BlueJ labs project that was set up in the above activities.

All labs that you complete in this course are to be submitted together as a single submission at the end of the course. When that glorious time finally arrives, you will zip up and submit your entire BlueJ project folder with all your individual lab classes inside the one and the same project folder. See the document "CCPS 209 Labs" for a screenshot that your project is supposed to resemble if not in style, at least in spirit.

Module 2: Classes and objects

Introduction

The structured programming approach, suitable for small problems that we sure have seen plenty during the first programming course, will take us only so far in expressing computations in a manner that we can mentally manage and modify in a controlled fashion. To keep large

systems understandable, they must be conceptually broken into smaller pieces so that each piece can be understood and tested on their own. These pieces can then be put together in various ways to create more complex software, even in various different ways analogous to how the exact same Lego pieces can be put together in different ways to create a spaceship or a medieval castle.

Some particularly useful pieces can even be reused in multiple projects. The very finest pieces whose usefulness is universal and is not tied to any particular application or problem domain will eventually become part of the standard library of the language. The implementation details of these pieces are hidden away to prevent the user code from depending on such ephemeral issues, which keeps the user code compatible with the future versions of these pieces.

Topics

1. Java language mechanism to define new data types as classes.
2. The principle of encapsulation that separates the public methods that the data type offers to its outside users from how the said functionality is implemented as a "black box" of private methods and data fields.
3. Representing the state of an object with instance fields that are used to store information about the past to the extent that it affects our future decisions and actions.
4. How local variables, instance fields, static fields and parameters are stored differently during the runtime execution of Java code, and how this different storage affects the behaviour and lifetime of each of these sorts of data.
5. Creating new instances of your class with the operator `new`. Constructors as special methods to initialize the state at new object creation.
6. The fundamental difference between primitive types and classes in the Java type system, as seen in its important consequences for parameter passing and equality comparison.
7. The occasionally counterintuitive but eventually numerous advantages of immutable data types.

Learning objectives

After completing this module, students know the general principle of how to model the concepts ("nouns") of a problem domain that exists outside their program as classes that exist inside their program, so that instances of those classes have capabilities (“verbs”) relevant to the problem that they are trying to solve. They know what classes, objects and methods are in the Java syntax and in actuality. They can explain how instance methods and fields are fundamentally different from static methods and fields, and can choose between these structures based on the present needs of the situation and expected lifetime of data. Students can write constructors in their classes to perform object initialization. They are aware of the distinction between primitive types and classes in Java, and can explain the nature and purpose of four of the primitive types `int`, `double`, `char` and `boolean`.

Readings

1. "CCPS 209 Course Notes", sections 2.1, 2.2, 2.3, 2.4, 2.6, 2.7, 2.8, 2.9.
2. GitHub repository `ikokkari/JavaExamples`: The example programs `Length` and `Fraction`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 2-1: Encapsulated essences", "Java 2-2: State and main", "Java 2-3: The education of little object", "Java 2-4: As above the line, so below".

Summary

Even though different languages have different foundational philosophies according to which they express computations and organization of data, the shared nature of all computations is inherent to the machine itself and not dependent on which made-up artificial language we use to express these computations. High level concepts and techniques can still be used to express low-level ideas.

Activities

1. Open the BlueJ project `JavaExamples`, and use BlueJ to create three separate objects of type `Length` with some suitable initial values. Inspect these three objects in the BlueJ object workbench to convince yourself of how they are separate entities.
2. Right-click one of your objects and set its length to 10 centimeters. Call the methods `getCentimeters` and `getInches` of that object. Are the results as you expected?
3. Right-click another one of your three objects and set its length to 10 inches. Call the methods `getCentimeters` and `getInches` of that object. Are the results as you expected? Discuss the reason for the asymmetry that you observe between this and the previous step.
4. Call the same methods again for all three objects until you become convinced that modifying the internal state of one object does not affect the internal state of other objects of the same type, but that these objects are separate entities that exist independently of each other in the object heap memory. These vases are all cast from the same mold, but they contain different stuff inside them, which makes them different.

Lab problems

This week, instead of writing our own data type as a class, we continue to drill in the imperative Java syntax. The aim is to push this syntax so deep inside your brain stem and fingertips that when the time comes to solve some real problems with Java, the language will be fighting on your side against the problem, instead of you having to fight both the problem and the language in a hopeless task of tug-of-war that would be impossible for even the mightiest of men.

Since this week's lab deals with strings and arrays, you should again read through the example method `stringDemo` in the `DataDemo` example, and the example method `arrayFirstDemo` in the `ArrayDemo` example, to refresh the syntax of dealing with Java strings and arrays. As an example of how to use the mutable `StringBuilder` to build up a result string efficiently from small pieces, you can study the `countAndSay` method in the `PythonToJava` example.

Armed with the above knowledge, you should work to complete **Lab 0(B)**, "**Putting Details Together, Part I**". All four methods must again be written into one class `P2J2` inside the very same BlueJ labs project inside which you wrote the previous lab `P2J1` from the first week. Again, you should start by implementing each method as a do-nothing method stub until the JUnit test class `P2J2Test` compiles, after which you start filling in the method bodies, one method at the time, with the actual statements that solve the problem.

Module 3: Inheritance and Polymorphism

Introduction

The highly useful encapsulation of the implementation details of some particular high level data type is these days considered basic software engineering. However, as long as every class is an island so that we have no way of expressing within the language that the two types `Eagle` and `Sparrow` are somehow related to each other in an important way that the types `Eagle` and `Planet` are not, we cannot also write methods that utilize such relationships, at least not in any kind of type-safe manner that the compiler could ensure to be safe to execute. As a result, we end up repeating ourselves by writing separate versions of the same method with nearly identical bodies but expecting different parameter types from the caller, just to keep the compiler type checking happy.

The ability to organize our classes into inheritance hierarchies where supertypes are extended into an unlimited variety of branching subtypes that all have the same public interface but vary on their implementation details turns out to be an extremely powerful mechanism. Not only can these subtypes inherit the method implementations from their supertypes so that we don't need to reuse code by copy-pasting it, but this mechanism allows us to write polymorphic methods such as `public void pluck(Bird bird)` that operate on arbitrary members of the `Bird` inheritance hierarchy so that one method can pluck a `Chicken`, a `Partridge` or for that matter, any kind of bird that anybody will ever create. Unlike in Python where functions just crash if you give them arguments that they can't handle, the explicit typing checked at the compile time allows the compiler to ensure that this method will never have to deal with an argument that is a `Planet` or a `BankAccount`.

Topics

1. Modelling the concepts in the problem domain as separate types inside our program so that nouns become classes or data fields, and verbs become their methods.
2. Organizing classes into inheritance hierarchies that represent useful commonalities between some types that correspond to mutually related concepts in the problem domain.
3. Inheritance relationships between supertypes and their potential multitude of subtypes that differ from each other in how some public behaviour that they all have in common has been implemented.
4. Abstract supertypes to model concepts for which some methods cannot be given a meaningful implementation at the superclass level, even though we want these methods to exist throughout the entire class hierarchy.
5. Polymorphic methods that operate on parameters whose types are abstract supertypes (such as `Bird`) and are yet compatible with any concrete subtype (`Hawk`, `Albatross`, `Hummingbird`) that anybody will ever think of creating in the future. **This is by far the single most important thing in this course, from which everything else then follows.**
6. Dynamic binding of method calls at runtime to enable polymorphic methods to be compiled and executed in a type safe fashion.
7. The universal superclass `Object` and its useful methods guaranteed to exist in all classes.
8. Proper implementation of equality comparison between instances of the same and other types, even between apples and oranges that, despite the famous proverbs, can be compared for equality just fine.
9. Decorators that dynamically modify the behaviours of objects from a given class hierarchy.

Learning objectives

Students know the basic Java language mechanism used to create subclasses from an existing class. They understand which parts can and which parts cannot be modified in the subclass while staying within the constraints of the Java type system. They have an understanding of the motivation why the inheritance mechanism and dynamic binding of method calls exist in the first place, that they allow the existence of polymorphic methods that are 100% type-safe at the runtime and yet compatible with the unforeseeable future. Students know how inheritance and method overriding are used to express compile-time variation in behavior between different subtypes of the same concept. They can explain how the decorator mechanism is used to express more dynamic kinds of variation within the same type hierarchy dynamically with polymorphic object decorator.

Readings

1. "CCPS 209 Course Notes", sections 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11.

2. GitHub repository `ikokkari/JavaExamples`: The example program `Barnyard`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 3-1: Future is already here", "Java 3-2: All hierarchies great and small", "Java 3-3: Please do not address my client directly", "Java 3-4: I, Object".

Summary

Problem domain concepts are not islands that exist independently of each other, but some concepts are subtypes of some more general concepts, such as `Chicken` and `Partridge` as subtypes of `Bird`. Inheritance and polymorphism allow our programming language to express the logic of its operations at the superclass level so that it automatically works for all the past, present and future subtypes of that concept.

Activities

Study the example class `Barnyard` from the lectures, and the toy example inheritance hierarchy defined there. (Again, pay no attention today to the fact that all these classes are static nested classes, as this does not affect the point of this example.) This entire example of `Animal` is admittedly a bit clichéd in the current year, but then again, all clichés became clichés in the first place by being so excellent that everyone and their brother wanted to use them!

To see the point of using toy domains such as a cartoon barnyard to illustrate concepts, think up some real world concept hierarchy that you deal with in your everyday life and work so that your concept hierarchy is **isomorphic** to the animal hierarchy in that the concepts that you are thinking of are in the same relationship to each other as the toy concepts modelled in the `Animal` class hierarchy. What would the two methods `getSound` and `fly` be named inside your problem domain instead of these verbs?

After this thought exercise, to drill in the syntax for Java class inheritance, edit the example class `BarnYard` to create yourself some additional subtypes of animals by extending either abstract superclass `Animal` or `Bird`, whichever is appropriate for the new type of animal you have in mind.

Finally, to practice using decorators, write a decorator subclass `AngryAnimal` that can turn any existing animal object into an angrier version of itself. The method `getSound` of this decorator should return the sound made by the underlying animal with the strings `"grrr"` and `"hisss"` around the original sound so that, for example, an angry chicken would return the sound `"grrr cluck hisss"`. The species of such animal should be the original species with the word `"angry"` tacked in front of it, such as `"angry chicken"`.

Something to ponder: Are these three decorators `AngryAnimal`, `MirroredAnimal` and `LoudAnimal` **commutative** so that the order in which they are applied to some existing animal is irrelevant, since any application order will always produce the same observable behaviour when

viewed from the outside? Argue that this is indeed so, or demonstrate the falsity of this claim with a simple counterexample. Next, which of these three decorators are **idempotent** so that applying the decorator twice is result-equivalent to applying it only once?

Lab problems

This week, we still continue drilling in the Java language and its basic structured programming concepts, with strings and arrays again giving our loops things to work on. This week we work on **Lab 0(C), "Putting Details Together, Part II"** from the document "CCPS 209 Labs". Write the code for the four required methods again into one and the same source code file `P2J3` in your BlueJ labs project, initially again as stubs that you fill in the meat in one at the time, and use the JUnit test class `P2J3` to verify that each method is working correctly before moving into the next one.

After you have completed the labs 0(A), 0(B) and 0(C) during these first three weeks, you have hopefully become sufficiently proficient in the core Java language to enable you to move on to solve more advanced lab problems coming up in the remaining modules.

Module 4: Java Collection Framework

Introduction

In this day and age, there is rarely a good reason for anybody to reinvent the wheels of sorting and other classic algorithms and data structures that they operate on. It is always preferable to use as much as you can from the standard library whose classes and methods have passed the test of time despite being mercilessly subjected to constant pummeling of different argument combinations by all those countless millions of lines of user code that rely on these methods to solve all sorts of problems. The Java Collection Framework organizes the concepts of data aggregation into one inheritance hierarchy whose various subtypes exhibit different behaviours behind the same public interface of useful methods.

Topics

1. Interfaces as fully abstract supertypes inside Java language, and the advantages and extra freedoms that follow from knowing for sure that some superclass has no data fields or concrete method implementations.
2. The general utility of dynamic sets and lists in classic imperative programming and problem solving, as already seen in Python with its `set` and `list` to keep track of things that we have seen and done so that we don't have to do those things again.
3. Use of generic types from the outside so that the same generic type gives birth to type instantiations that deal with different data types left open to be decided by the user code.

4. Commonly used concrete subclasses of the `Set<E>` and `List<E>` interfaces. Example situations where each concrete subclass would be the most appropriate.
5. Polymorphic methods that can safely operate on arbitrary subtypes of collections, sets and lists, instead of having to write the same method separately for each subtype.
6. Stepping through the elements of a collection with iterators.
7. Modifying the behaviour of some method dynamically by giving it strategy objects that get consulted in some decisions during the method execution.
8. Sorting and searching inside a collection with respect to various element orderings.

Learning objectives

Students can list and explain the purpose of the two main branches of `Collection<E>` class hierarchy, and can apply its most common concrete implementations to store and remember things that are useful as a part of solving some other computational problems. Students know how to use and apply existing generic classes to create and use objects, with a sufficient understanding of how these type arguments work between different type instantiations. The students know how to use iterator objects to process the elements of a collection, and can appreciate iterators as an example of maximizing the generality of polymorphic methods. They know that those iterators may either be backed by actual collections, or they virtually generate the sequence in purely computational means, but this difference is irrelevant for the code that processes the data coming from these iterators. Students know the general technique of modifying the internal operation of some method with strategy objects that are passed as arguments to them. As a practical application of this technique, they can modify the criteria used to compare the elements during sorting and searching.

Readings

1. "CCPS 209 Course Notes", sections 4.1, 4.2, 4.3, 4.4, 4.5, 4.6. (We will skip section 4.7 that discusses the association map interface `Map<K,V>` in this module, and will cover it later once we have something useful to do with these association maps.)
2. GitHub repository `ikokkari/JavaExamples`: The example programs `CollectionsDemo`, `NatSet` and `ContinuedFraction`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 4-1: Set me up", "Java 4-2: One at the time, please", "Java 4-3: A fundamentally illusory distinction", "Java 4-4: A natural slide", "Java 4-5: An infinite self-referential integer sequence".

Summary

Implementing your own data structures and algorithms is something that had to be routinely done decades ago, but in this day and age, the standard library of every serious programming language comes with the best implementations as classic data structures and algorithms. Object-oriented programming and inheritance allow us to group these data structures so that the same polymorphic method can operate on any subtype of a particular type of collection, and

we don't need to copy-paste and edit the same sorting algorithm whenever somebody comes up with a new implementation of data structure.

Lab problems

In this week's lab, we will actually create our first subclass extended from an existing superclass, and override some methods to do something new that did not exist in the original superclass. The Java Collection Framework provides us the old workhorse `ArrayList<E>` that we shall further extend and customize as described in **Lab 3,** "**Extending an Existing Class**" from the document "CCPS 209 Labs". Inside the very same BlueJ lab project in which you have already created your solutions to the previous lab problems, create the brand new class `AccessCountArrayList<E>` that extends `java.util.ArrayList<E>`. Override its `get` and `set` methods to do the extra thing as specified in the lab document after calling the superclass version of the same method to do the actual data structure modification behind the scenes. Add the JUnit test class `AccessCountArrayListTest.java` into your BlueJ project, and use it to verify that your new class behaves as specified.

Module 5: GUI programming with Swing

Introduction

Historically, GUI programming was the original "killer app" that showcased the benefits of the object-oriented programming techniques of inheritance and polymorphism as the organizing principles for your data and code. This approach avoids needless repetition and imposes a clear discipline that allows arbitrary components to coexist and play together regardless of where and when those components originated. Even though Java has been long dead in the front end, we can use the Swing GUI programming framework as a demonstration and a proof of concept to sell the students a whole bunch of techniques about the power of organizing your concepts into inheritance hierarchies.

Topics

1. Representing the concepts from the specific problem domain of graphical user interfaces as class hierarchies in the original AWT and the later Swing frameworks for GUI programming.
2. Swing GUI components implemented as subtypes of existing GUI component types.
3. The `Graphics2D` engine for the modern and accurate high-resolution rendering of colourful shapes on top of Swing components.
4. Dynamically customizing the behaviour of Swing components with strategy objects.
5. Building up arbitrarily complex GUI layouts by adding simple components inside other components that were designed to hold them.

6. Compare and contrast the factory pattern to the ordinary way of creating objects the usual way with the operator `new`.
7. Inner classes used to represent objects that cannot meaningfully exist on their own, but only in the context of some object of the outer class that they are "inside" of.
8. Writing event listeners to observe and react to the user events that take place with Swing components.

Learning objectives

Students know the central classes of `JComponent`, `Graphics2D` and `Shape` that are used and extended to create graphical GUI components in Swing. They know enough of the most commonly used handful of polymorphic methods defined inside these classes so that they are able to apply this knowledge to create their own subtypes of custom Swing components. Students can explain how nesting a class inside another class is different from writing that class as a top level entity of its own, and can apply this knowledge of nesting in writing event listeners that react to the events that happen to Swing components. They know the idea and motivation behind object factories, and can appreciate the flexibility in which Swing classes and methods allow dynamic customization with tactically selected strategy objects.

Readings

1. "CCPS 209 Course Notes", sections 5.1, 5.2, 5.3, 5.4 and 5.5. (The sections 5.6 and 5.7 about turtle graphics are extra for interested students who wish to see a more interesting "animal" class hierarchy whose different species and subspecies respond in different ways to the same commands and situation.)
2. GitHub repository `ikokkari/JavaExamples`: The example programs `ShapePanel` , `Counter` and `Minesweeper`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 5-1: ...or we will surely Swing separately", "Java 5-2: We wuz framed", "Java 5-3: Rendering the invisible", "Java 5-4: The inner ring", "Java 5-5: Meaningful events". (Students interested in GUI programming can check out extra material of "Java 5-6: Turtles all the way up", "Java 5-7: Blasted from the past" and "5-8: Trading shade for space".)

Summary

Historically, better illustrations of the overall power and usefulness of inheritance and polymorphism than GUI programming have been few and far between. This particular problem domain was originally the flashy "killer app" that closed the sale of these concepts to the wider audience of programmers of all levels. Just like some animals are birds and therefore can fly at least in our local toy model of reality, some GUI components are containers that can contain other components inside them.

Activities

Read through the example classes of `ShapePanel` and `Counter` carefully enough that you could explain the purpose that each individual line serves in them, and the hypothetical effect to the whole if each line were left out.

Lab problems

This week **Lab 5, "Introduction to Swing",** from the document "CCPS 209 Labs" is perhaps best solved by copy-pasting the code from `ShapePanel` and `Counter` demonstration classes and cutting and modifying those individual statements and blocks of code that need to be cut and modified, instead of writing the class from scratch on your own. In this lab, your `Head` component will draw some sort of simple cartoon head, and has an inner `MouseListener` class that makes the head open and close its eyes whenever the mouse cursor moves in and out of that component. A `main` method that creates an entire grid of separate heads again emphasizes the fundamental fact that all these heads are separate objects with their own private states.

Module 6: Exception handling

Introduction

If some method is supposed to extract the integer from a string of digits such as `"421799"`, what is this method supposed to do when some wag gives it the string `"Hello world"` instead? If another method is supposed to read in the data from the given text file intended to contain country data information, what should this method do when the file does not exist? How about when the text inside that file contains a syntax error according to the rules of the language used to encode the data inside it? Or what if the syntactically valid file informs you that the area of some actual country is approximately equal to the surface area of the Sun?

Sometimes the program or a method in it cannot possibly do what it was supposed to do due to no fault of its own, but the outside world did not hold on to its end of the deal. None of these concepts even exist in the level of the programming language, but are semantic properties that we ascribe to the method and the reason why we want that method to exist. We therefore need a whole new exception mechanism in the language to distinguish between a method successfully returning a result, as opposed to that method realizing that it has been placed in a logically impossible situation so that the only way out is to put its hands in the air and give up.

Topics

1. Preconditions and postconditions that describe in natural language at semantic level what purpose the method achieves, as opposed to how statements in Java language describe at an operational level what the method does.
2. The Liskov substitution principle that always tells us whether some method has been properly overridden in a subclass.
3. What "something went wrong" means in different levels of abstraction of a computer program and its execution.
4. The Java exception machinery and its associated keywords `throw`, `catch` and `finally` in the language.
5. The class hierarchy of `Throwable` errors and exceptions in Java. The distinction between checked and unchecked exceptions to divide the blame in case things don't work out.
6. The need for `finally`-blocks to ensure that allocated resources will get properly released regardless of which way the execution leaves the `try`-block.
7. Using assertions as tripwires to quickly pinpoint where exactly your reasoning about the program behaviour has been inaccurate, and this way uncover logic errors in your code.

Learning objectives

After completing this module, students think about their methods in terms of pre- and postconditions instead of their current implementation as a particular series of statements, and how the rest of their program depends on those postconditions being fulfilled every time. They are aware of the purpose and goal of the Liskov Substitution Principle as expressed as its "bumper sticker" version of how our polymorphic methods should never have to worry about the exact subtype of their arguments, and how following the principle automatically achieves that ideal. Students learn to deal with the exception mechanism in the Java language when they call some Java standard library methods that can potentially fail and throw an exception instead of successfully returning a result, and can make their own methods fail and throw exceptions when the situation warrants. They know what assertions are and how they make your reasoning about code postconditions explicit and automatically verified during the program execution.

Readings

1. "CCPS 209 Course Notes", sections 6.1, 6.2, 6.3, 6.4, 6.5, 6.6.
2. GitHub repository `ikokkari/JavaExamples`: The example program ExceptionDemo.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 6-1: We will fix it in the post", "Java 6-2: Let's all do a SOLID for Babs", "Java 6-3: The impossible takes a little longer", "Java 6-4: Every exit well covered", "Java 6-5: Dividing the blame".

Summary

The computer processor executes very simple low-level instructions one instruction at the time, without any higher-level semantic view about what these instructions exist for and what it means for the entire program execution to fail at the semantic level above the mechanical execution of these instructions. In the majority of modern programming languages from the past three decades, exceptions are a separate mechanism from return values to force the program execution to backtrack to the most recent alternative and continue from there once the current execution path has become untenable.

Lab problems

Since throwing an exception is never the correct answer in any of our graded labs, our lab problem collection does not feature any problems about using exceptions. Instead, we continue to practice the very important topic of Java Collection Framework and the useful data structures that it gives up. This week we shall focus on the `List<E>` branch whose subclasses represent sequences of elements of the generic placeholder type `E` so that the size of this sequence can dynamically grow and shrink. (Note that all Java `Collection` classes have a `size()`, not `length()`, even when these collections happen to be `List<E>` linear sequences without gaps, so that they could be treated as sequences whose elements are accessed based on their positions with the methods `get` and `set`.)

This week we shall complete **Lab 0(D): "Lists of Integers**" from the document "CCPS 209 Labs", with the JUnit test class `P2J4`. Since BlueJ makes it rather painful to enter even a simple `List<Integer>` instance with a handful of elements as a Java expression into the textfield inside the BlueJ method argument dialog, you probably want to write a separate test method alongside the actual four methods that you write in this lab. The list that you want to use as a test case is then hardcoded into the test method as a local variable. This way you can just call the test method every time you want to repeat the test.

Of course, for those students inclined to learn to use JUnit in writing tests, are quite welcome to do so, since this allows for the clean separation between the methods to solve problems and the methods to test them.

Module 7: I/O Streams

Introduction

Everything inside a computer is made of bytes that have no knowledge about what these bytes together currently represent. Especially when grouped along with some other nearby bytes to

collectively represent integers, strings, birds, raster images or entire executable programs, this meaning is imposed on these bytes from the outside by the way that the program deals with these bytes. The bytes themselves only contain small integers made of bits that represent zeros and ones, and are not affected by the meaning that the users of these bytes happen to ascribe to them.

A mechanism that allows us to move this bytes back and forth turns out to be extremely amenable to thinking about using inheritance and polymorphism, with the abstract superclasses representing the public interface of the mechanism, and concrete implementations of the subtypes then deal with the details of the actual transmission mechanisms in copying these bytes in memory, file system or the Internet connection. Our code can then depend only on this high-level abstraction that is guaranteed never to change, instead of being fundamentally tied to some particular flavour of the month concrete mechanism that is going to change in the future many times over. Our polymorphic methods that operate on the concepts expressed in the supertype level are ready, willing and able to handle whatever the future may bring!

Topics

1. Dependencies between classes, and why it is a good thing to depend on high-level abstractions instead of concrete low-level details.
2. The `InputStream` and `OutputStream` class hierarchies for reading and writing raw unsigned bytes.
3. The `Reader` and `Writer` class hierarchies for reading and writing Unicode characters.
4. Dynamically modifying the behaviour of various streams with decorators.
5. Processing input that consists of lines of text separated into fields.
6. The handy facade class `Scanner` to unify text processing operations regardless of the source of that text.
7. Using a `Map` with integer values associated with keys to keep track of how many times we have seen something during the execution of our program.

Learning objectives

In their own coding, students who have completed this module are aware of the principle of why it is generally better to depend on high-level abstractions than low-level details. They are able to use concrete implementations streaming interfaces of bytes and Unicode characters in their own code to move data from various points A to points B, and can also dynamically modify the behaviour of such streams by decorating them with data compressors from the Java standard library. They gain appreciation of the fact that inside a computer, all data is raw bytes to which we impose a meaning from the outside. Students are aware of the Unicode standard that encodes every character to the same unique integer codepoint across all systems. Students can use the `Scanner` utility to process textual input in their own programs.

Readings

1. "CCPS 209 Course Notes", sections 7.1, 7.2, 7.3, 4.7. (Sections 7.4 and 7.5 are extra for the students interested in the special topics of reflection and object streams.)
2. GitHub repository `ikokkari/JavaExamples`: The example programs `GZip`, `WordFrequency`, and `SomeUnixCommands`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 7-1: Rawbyte express", "Java 7-2: There is always room for one more who uses Rexona", "Java 7-3: Question of character", "Java 7-4: A scanner lightly", "Java 7-5: God is the same everywhere".

Summary

Separating the mechanism from policy is a standard software engineering principle. Making your dependencies as high-level as possible so that you don't depend on any ephemeral details of the present keeps your code compatible with future changes and developments. This is illustrated by having our method not care about the mechanism of how its high-level data is encoded and transmitted as bytes.

Lab problems

The lab this week combines a whole bunch of ideas that have been discussed in this course so far. This week we shall do **Lab 6:** "**Processing text files, Part I**" from the document "CCPS 209 Labs". This lab will first have the students create an abstract superclass `FileProcessor<R>` whose methods define an algorithm of three abstract individual stages. Concrete subclasses will then provide different implementations for these three stages to implement different algorithms from the same algorithm template.

For example, the subclass `WordCount` should emulate the basic behaviour of the classic Unix command line tool `wc` that counts how many characters, words and lines the given text contains. The next week's lab will then have the student create another subclass of `FileProcessor<R>` with these three methods implemented very differently so that they now implement a very different Unix command line tool.

(These exercises do not require any knowledge or use of the Unix command line, since everything happens inside Java and its virtual machine. We merely talk here about Unix and command line tools as a helpful analogy that further clarifies to the readers who know those tools what these classes are supposed to achieve.)

Module 8: Generics

Introduction

Since its verbose and restrictive beginnings back in the early nineties, Java has evolved towards a more dynamic and functional nature. Back in the year 2005 already in the past distant enough to practically be a different country, the Java 5 language update was major enough to skip directly from version 2 to 5 with the introduction of generic types and boxing that fundamentally changed the spirit of the entire language. However, since these important features were not part of Java from day one, their incorporation to the language while guaranteeing backwards compatibility with all existing codebase has some counterintuitive consequences on how these features can be used.

Regardless of how these generics are implemented inside the language and its runtime execution mechanism, the relationship between generic types and our desire to write polymorphic methods that should never have to worry about the exact type of the objects they receive as arguments is quite different from what most people might intuitively assume. The correct way to make these things work together requires the concept of artificially bounded generic types that make for much better parameter types for polymorphic methods than ordinary generic types.

Topics

1. Minor Java 5 language improvements of static imports and variable length argument lists.
2. Boxing, the automatic conversion between primitive types and wrapper classes in both directions in Java.
3. Hidden pitfalls of boxing involved in the equality comparison between primitives and wrappers, and iteration over a long sequence of elements.
4. Review of the use of generic classes such as `ArrayList<E>` from the user code.
5. Restrictions to use of generics imposed by the erasure mechanism of Java.
6. Inheritance relationships between generic classes.
7. The counterintuitive and surprising consequences of using a generic type as a parameter type of a polymorphic method.
8. Bounded generic types that allow us to properly implement polymorphic methods that operate on a generic parameter type.

Learning objectives

Students have been dipped into the notion that most people know but few know well enough in their bones to behave as if they really believed this; that programming languages were not handed down to us by God or emerged as laws of nature, but were designed by people as artifacts that, same as all other man-made systems, have tradeoffs in their design and will evolve over time. They learn to rely on the boxing mechanism that allows us to momentarily forget the unfortunate distinction between primitive types and wrapper classes in Java, but remain aware of the pitfalls of this related to equality comparison and mutability.

Before this module, students have already used generic classes from the Java standard library in their code and extended existing generic classes to both generic and non-generic subtypes, but after this module, can create their own generic classes while being aware of the restrictions inherent in this mechanism. Most importantly, they can confidently write polymorphic methods with generic parameter types so that a method can, for example, expect any kind of `List` of any kinds of `Bird` objects as its argument, and will always compile and do the right thing regardless of whether the argument is an instance of `HashSet<Hawk>` or `ArrayList<Albatross>`.

Readings

1. "CCPS 209 Course Notes", sections 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8.
2. GitHub repository `ikokkari/JavaExamples`: The example programs `Java5Demo`, `Pair`, `GenericsDemo`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 8-1: Wrapped to go", "Java 8-2: Types to decide later", "Java 8-3: His raw materials".

Summary

All artificial systems, not just our programming languages, that people actually use to solve real problems in the real world must keep evolving over time to meet the new needs that were not anticipated at the time the system was originally designed. In programming languages, such improvements must be done in a way that does not break any existing user code that was legal before the change. This will eventually prevent the system from changing further, but meanwhile we can enjoy the small conveniences offered by the new features.

Lab problems

Continuing on the theme of Lab 6 from the previous week's module, the exercise for this week is **Lab 8:** "**Processing text files, Part II**" from the document "CCPS 209 Labs", with its associated JUnit test class `TailTest`. Same way as in last week, create a subclass of the existing abstract superclass `FileProcessor<R>`, but the three stages of the abstract algorithm template this time

implemented differently so that together these stages emulate the behaviour of the Unix command line tool `tail` that finds and emits the last `n` lines of the given text.

Module 9: Concurrency Fundamentals

Introduction

Even though we could do so in principle, we don't want to write our programs as single `main` functions thousands of lines in length. Nor do we store all program data inside one giant blob object. Instead, we separate the data and the operations executed on it into smaller pieces that correspond to clearly understood problem domain concepts so that these pieces of program can be similarly understood, modified and reused on their own, the very execution of a large and complex program can be broken into smaller execution threads. In the same spirit as how we don't care which memory location some variable is stored in, we don't care how and in which order the Java Virtual Machine chooses to assign these threads execution time.

When executed in a modern computer with multiple processor cores, we can even have these execution threads running in parallel to make the program finish faster. However, if these execution threads need results from the other threads, or when they try to modify the contents of shared objects, mutual exclusion with special lock objects becomes necessary to have all these threads execute in some proper order to guarantee the correct outcome without unpredictable interference.

Topics

1. Concurrent execution of threads inside the Java Virtual Machine.
2. The discipline of preemptive multitasking of threads to assign the execution time each thread receives, as contrasted to co-operative multitasking as exemplified by Python generators.
3. Living with the inherently non-deterministic nature of the concurrent execution of code.
4. Proper creation and care of execution threads in Java either with explicit thread creation or by assigning that duty to an `ExecutorService` utility.
5. Giving a thread a task to execute so that we can read the future results of that task once it has been completed.
6. Objects and heap are shared between all threads, whereas the local variables stacks are separate inside each individual thread.
7. Race conditions that result from concurrent modification of shared data with non-atomic operations.
8. Using mutual exclusion locks to protect access to the critical sections of the code to ensure atomicity of modifications of objects that are shared between threads.

Learning objectives

Students recognize that execution of the Java statements, so far basically taken for granted as a thing that somehow magically happens underneath it all, is actually a real thing that has an existence on its own, and therefore multiple instances of that thing can simultaneously coexist. They know how to launch any number of their own execution threads, and give these threads `Runnable` tasks to execute so that their code can do something else and come back later to gather the future results of these tasks. Students can explain, with aid of simple scenarios, the incorrect results and behaviour that can be caused by a suitably unfortunate interleaving of execution threads that modify the value of a shared object. If told that some part of their program is a critical section, these students would know how to protect the access to that critical section using mutual exclusion.

Readings

1. "CCPS 209 Course Notes", sections 9.1, 9.2, 9.3, 9.4, 9.5.
2. GitHub repository `ikokkari/JavaExamples`: The example program `ConcurrencyDemo`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 9-1: There is plenty of time at the sides", "Java 9-2: After you, sir, I insist", "Java 9-3: Every thread an island", "Java 9-4: One pie, many fingers", "Java 9-5: Critical solitude".

Summary

Spreading the execution sideways into separate threads that can be executed in parallel is a straightforward way to shorten the total execution time, but it also creates a whole new dimension of difficulties that did not even exist in the purely sequential execution of code done so far. The best approach is to keep it simple and use standard concurrency controls to rein in the complexity while giving it enough leeway to reap the benefits of concurrency and parallel execution in both organizing the code into understandable separate chunks and having it complete in a shorter time.

Lab problems

This week's lab again combines a bunch of ideas covered in this course so far. This week we shall do **Lab 7:** "**Concurrency in Animation**" from the document "CCPS 209 Labs" that also refreshes the technique of extending the existing Swing component classes to create your own component types. However, as part of its construction, this time the Swing component will start a new execution thread whose job is to recompute the state of the world and update its appearance to the user 50 frames per second, each animation frame displaying a snapshot of a swarm of `Particle` objects that buzz randomly around the surface of the component.

Students should take care to ensure that if the window that contains this particle swarm is closed, their component properly shuts down the shop and terminates the animation thread. To convince yourself that this indeed happens, have the `WindowListener` instance output a short termination message on the console.

Module 10: Concurrency Controls

Introduction

The ability to spread the execution sideways into multiple parallel execution threads opened up a whole new dimension to how we think about code and its runtime execution. We wish to squeeze out every drop of advantages from this new and exciting possibility, but also wish to avoid introducing the notoriously tricky and inherently unpredictable concurrency bugs into our program. The best way to reach the sufficient certainty about the correct execution of our program in all environments for all possible execution interleavings is to never try to be clever by reinventing the wheel, as this would be akin to trying to perform your own surgery. Instead, we should rely on tried and true high-level concurrency control mechanisms that have passed the test of time and that hide their internal complexities away from our eyes.

Topics

1. Using condition variables to implement blocking until some condition becomes true in some indeterminate future.
2. The concept of fairness in concurrency, usually enforced the same way as in real life by making the competing threads wait for their turn inside some first-in-first-out queue for whatever exclusive resource they need.
3. Giving another thread a friendly nudge to nicely ask it to terminate whenever it is possible to do so in a safe and clean manner.
4. Semaphores as the classic universal control structure for concurrency.
5. A sampler of modern concurrency controls illustrated with the example of searching for humongous prime numbers in parallel, with this concurrency kept in line from race conditions with four different high-level concurrency controls.
6. The separate execution thread of Swing that manages the GUI components while the user code is doing its own thing.

Learning objectives

After this module, students are able to make their Java methods nicely wait in place for something outside their reach to happen, without wasting precious processor cycles running around in a circle like some dog chasing its tail. They understand the non-deterministic nature of concurrent and especially parallel execution of threads. They can design their code so that it

plays along nicely when executed concurrently with an unknown number of other threads, and will close down the shop and clean after itself whenever asked nicely to do so. Students can visualize a semaphore as an integer with atomic increment and decrement operations that block when the value would become negative, and can use the semaphore as a mutual exclusion lock or to implement blocking. They are aware of the `java.util.concurrent` package and can name some of the important concurrency controls inside it.

Readings

1. "CCPS 209 Course Notes", sections 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7.
2. GitHub repository `ikokkari/JavaExamples`: The example programs `MySemaphore`, `BigPrimes`, `Sliders`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 10-1: Waiting for go-do", "Java 10-2: Simplicity is prerequisite for reliability", "Java 10-3: Faraway entities", "Java 10-4: You don't have to go home, but you can't stay here", "Java 10-5: Smooth sliding".

Summary

Concurrency is a gargantuan subject whose surface we can barely scratch in these lectures, as it expands our notions of program execution into a whole another dimension wide open with both possibilities and pitfalls. The best practice in maximizing the gains of this approach is to use well-known and standard concurrency controls that keep everything disciplined while maximizing the gains from concurrency and parallelization.

Lab problems

Concurrency is a big topic that does not fit inside a single lab, and writing automated testers for it would be even more complicated than that. However, to prepare something that we can use as activity next week, and to review the important technique of recursion so that we won't forget it since it has been a while since we last used it during the first programming course, we shall now go all the way back to **Lab 0(F), "Two Branching Recursions**", and solve the two problems there. These branching recursions will then serve as an interesting playground for parallelization during the next week's final module of this course.

Module 11: Functional Programming in Java

Introduction

Your author is writing this on a desktop computer that is eight years old and yet works just fine for everything that he ever wants to do, but the young students might have higher dreams of

solving problems with raw computational power. Now that the famous Moore's law is quietly tapering off after giving us all this exponential growth in computing power available to everybody, we have to find new ways to speed up our computations. Using better data structures and algorithms will always have their place, but there are no great improvements in the horizon in that field as these new data structures and algorithms keep shaving the third decimal place instead of providing speedups of order of magnitude.

The only way to keep the progress going, so that more problems fall under the hammer of computation to smash them open, is to spread our computations sideways in time by breaking them down into separate tasks that are independent of each other so that their order does not matter, and assign multiple computation units to execute these tasks in parallel. However, all problems fall on a continuum depending on how much they resist being broken down to such smaller independent tasks. Especially many inherently sequential problems in which no task can begin without seeing the result of the previous task cannot be parallelized at all this way. At the opposite end of this continuum, the embarrassingly parallel problems can be parallelized without any apparent theoretical limit, and many such problems abound in optimization, machine learning and computer graphics. The Fork-Join framework and computation streams allow us to reap the benefits of parallelism without even having to think about execution threads, the same way as high-level languages such as Java allow us to think about computations without having to think about how they actually break down to the low level details of machine code.

Topics

1. The important laws of Moore and Amdahl, and the hard constraints that these laws impose on our quest to speed up our computations.
2. Using the Fork-Join framework to parallelize the execution of branching recursions.
3. The syntactic sugar of lambda expressions to conveniently define small function objects, especially one-liners, on the spot to be passed to other methods.
4. Computation streams that express computations in the map-reduce framework.
5. Important operations used as the stages inside a computation stream.
6. Collecting the end results coming out of a computation stream.

Learning objectives

Students can define the laws of Moore and Amdahl, and explain the relevance of these laws to how fast computational problems can be solved. They know that computational problems that they will encounter during their careers will vary greatly to what extent they can be parallelized. Given an existing branching recursion, they can find ways to parallelize it using the Fork-Join framework. Students can construct one-liner functions objects as lambda expressions. They can identify the supplier and the collector that bookend some computation stream, and can explain the purpose and behaviour of the intermediate operations of `map`, `filter`, `skip`, `limit` and `flatmap` inside that stream.

Readings

1. "CCPS 209 Course Notes", sections 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7.
2. GitHub repository `ikokkari/JavaExamples`: The example programs `FJMergeSort`, `ImageLoops`, `MapReduceDemo`.
3. YouTube playlist "CCPS 209 Computer Science II" videos: "Java 11-1: When you see a fork in time, take it", "Java 11-2: Laws of nature and man", "Java 11-3: Objects of computation", "Java 11-4: A clear case of black and white", "Java 11-5: Go with the flow", "Java 11-6: Calm and collected", "Java 11-7: Only if you must", "Java 11-8: A billion little nulls".

Summary

Expressing computations in the Fork-Join framework and computation streams in modern Java is not quite as straightforward as using the classic structured programming loops and conditions. However, once you have found a way to express your computation this way, you have hitched your wagon to a mighty steed that will only get faster in the future as the hardware for distributed and parallel execution of code improves.

Activities

This week, we shall take some working solution to the branching recursion lab from the previous module, and together see how its recursions could be parallelized with the Fork-Join framework.

Lab problems

To gain proficiency and confidence in expressing computations as streams instead of familiar structured programming and recursion, **Lab 10: Computation Streams** from the document "CCPS 209 Labs" makes them solve problems with computation streams. These problems have been chosen so that all of the central operations on streams will show up somewhere in them.