Week 7 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- Gantt charts
- Detailed design
- Design patterns
  - Composite
  - Command
  - Decorator
  - Observer
  - Factory method
  - Abstract factory
  - Singleton
  - Strategy
  - Adapter

Questions?

Construction Techniques

Bought and customized systems

- It's not always necessary to build a system from scratch
- A **bought and customized** system is one with several bought subsystems that have been customized and integrated into a product that satisfies requirements
- These systems come in a number of overlapping categories:
  - **Commercial off-the-shelf (COTS) systems** are generic products (like SAP, SalesForce, or Blackboard) that need significant customization for a particular client
  - **Component-based systems** are constructed from individual objects that use standard interfaces, like Java Beans and .NET
  - **Service-oriented systems** are like component-based systems except that the connection between components is over the network, and the services are provided by servers

Pros and cons of bought and customized systems

- Pros:
  - Widely used components are usually reliable
  - Good documentation and standards exist for using such components
  - Constructing these systems is usually faster, and costs are easier to predict
- Cons:
  - Increased dependency on outside organizations and their support
  - Lowered flexibility
  - Software engineers have less creative control, potentially reducing job satisfaction (boohoo)

Built systems

- On the other hand, you can build a system from scratch (as we're doing in this class)
- Built systems revolve around three activities:
  - Designing algorithms
  - Designing data structures
  - Programming

Designing algorithms

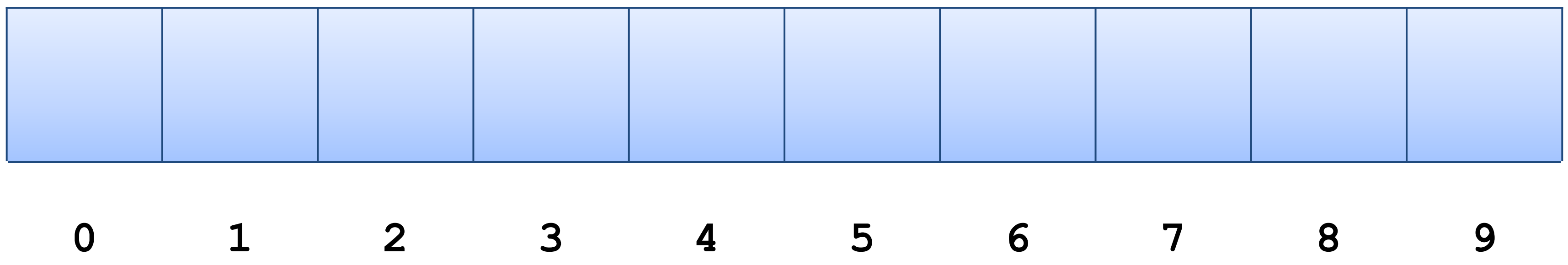
- An **algorithm** is a finite sequence of steps for solving a problem
  - A finite recipe for an infinite number of answers
- There are also **heuristics**, which are not guaranteed to solve the problem but can give answers that are good enough
- Some simple algorithms were discussed in COMP 1600:
  - Bubble sort
- More complex algorithms were discussed in COMP 2100:
  - Merge sort
- Even more complex algorithm types are discussed in COMP 4500:
  - Greedy, divide-and-conquer, dynamic programming, and more
- Algorithm design is challenging, so it's good to consult the literature from a specific area to see if someone has already come up with good ideas

Designing data structures

- A **data structure** is a way to store and organize values in computer memory
- COMP 2100 is supposed to introduce you to many useful kinds of data structures, many of which fall into two categories
  - Contiguous data structures
  - Linked data structures
- There is no such thing as the best data structure for everything: use the right tool for the right job

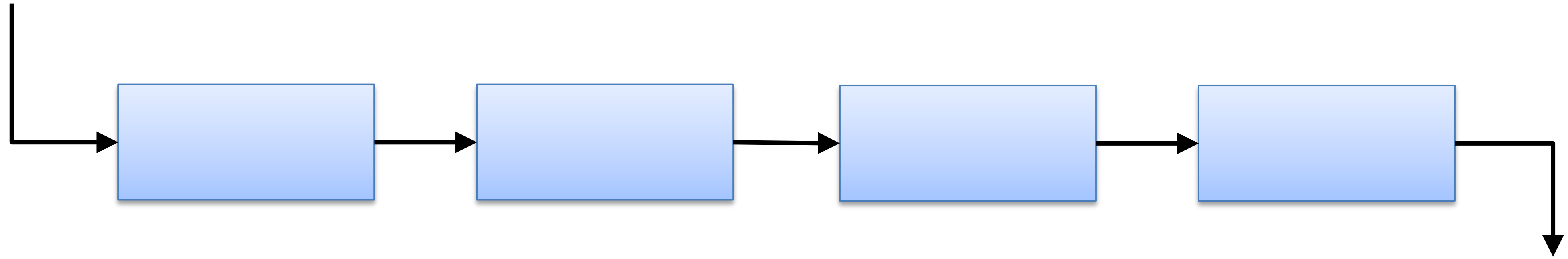
Contiguous data structures

- Contiguous data structures are built around array-like primitives
- Examples: arrays, **`ArrayList`**, **`HashSet`**, **`HashMap`**, **`Vector`**, **`ArrayDeque`**
- Pros:
  - Arbitrary elements can be jumped to in constant time
  - Iteration through elements is fast
  - Better locality of reference (elements are close together in memory)
- Cons:
  - Space is usually wasted, sometimes almost half
  - Resizing is often expensive



Linked data structures

- Linked data structures are built around nodes linked together
- Examples: linked lists, trees, **`LinkedList`**, **`TreeSet`**, **`TreeMap`**
- Pros:
  - Space is only allocated for actual elements
  - Adding or removing elements can take constant time
- Cons:
  - Reaching arbitrary elements requires visiting other nodes
  - Iteration through elements is slower
  - Elements can be spread throughout memory, worsening caching
  - Each node has the overhead of additional pointers in addition to data



Programming

- **Programming** is **sigh* creating a description of algorithms and data structures that can be executed on a computer
- High-level programming languages are human-readable but not directly executable
  - Some languages like C and Rust are **compiled** into **machine language**
  - Some languages like Python and PHP are **interpreted** and run on the fly
  - Yet others like Java and C# run in a **virtual machine**, which combines elements of both interpretation and compilation
- The **syntax** of a language is the **lexicon** (words or symbols used) and its **grammar** (the ways words and symbols can be combined)
- **Semantics** describe the meaning of syntactically correct expressions
- **Pragmatics** describe how to use a language to get things done

Programming language paradigms

- If you've taken COMP 3200, you know that there are different flavors of programming languages called **paradigms**
- Paradigms
  - **Imperative**
  - **Data-driven**
  - **Declarative**
- Because it maps most closely to what the machine is doing, imperative languages have long been popular
- It still pays to know how to think about other languages which can be useful in specific situations
- Pick the language that's right for the product and the client, not necessarily the one you're most comfortable with

Imperative languages

- Imperative languages manipulate values in memory locations
- If you can turn your solution into a list of instructions executed in order, imperative languages are a good fit
- C and Pascal are quintessentially imperative
- Most of the Java we do is imperative, but Java can be written in a functional style and in an event-driven style (though it's awkward)
- Object-orientation is a layer that is often applied to imperative languages but shows up in other paradigms too

Sample C/C++

```
double mean(double a, double b)
{
   double total;
   total = a + b;
   return (total / 2.0);
}
```

Data-driven languages

- Data-driven languages give rules for manipulating data
- The rules specify what happens the program runs into data formatted a certain way
- Examples:
  - XSLT is a language for converting one XML document into another
  - AWK and sed are Unix utilities for processing text
- If you do a lot of processing of data files, you might need to use one

Sample XSLT

```
<xsl:template match="volume">
      Vol. <xsl:value-of select="." />,
</xsl:template>
```

Declarative languages

- Declarative languages cover a lot of ground
- Logic languages like Prolog give rules that state goals and ways to achieve them as well as facts that are goals that have already been achieved
  - Traditionally used for AI
- Functional languages like Haskell express everything in terms of functions that return values (but don't actually change the state of memory)
  - Other examples: Erlang/Elixir, Clojure, F#
  - JavaScript allows for functional programming
  - Scala is multi-paradigm with functional ideas

Sample Prolog

```
domesticated(X) :- cow(X).
cow(bossy).
? domesticated(bossy).
```

Sample Haskell

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Idioms

- **Idioms** in programming languages are common ways to express ideas
- Example Java idioms:
  - Use `for` loops when you want to repeat a specific number of times
  - Use `while` loops when you don't know how much you're going to repeat
  - Use a three-line swap to exchange values
- It's a good idea to read code in a language you don't know well to figure out the idioms that people use
- Some people use idioms from languages they know better that can be either inefficient or confusing if they're not used in a different language
- **Syntactic sugar** is a kind of formalized idiom
  - An easy-to-use grammatical structure is converted to a harder-to-read one behind the scenes
  - Example: enhanced `for` loops in Java

Programming style

- Each language has stylistic considerations for how to write readable code
  - Many workplaces and open source projects publish style guidelines
- **Naming conventions** cover how to name variables, methods, classes, files, packages, etc.
  - Spelling matters
  - Capitalization is often a matter of convention
  - Being consistent makes everything clearer

Naming

- Most languages encourage either **snake case** or **camel case**
  - Snake case breaks up words with underscores: **`nuclear_silo_radius`**
  - Camel case breaks up words with capitalization: **`nuclearSiloRadius`**
  - Snake case is common in C and Python
  - Camel case is common in Java and C#
  - Very few programming languages allow spaces in variable names
- I prefer variables to be explicit so that it's clear what we're talking about even if we start reading in the middle of unfamiliar code
  - Java tends toward the explicit rather than the abbreviated
- A few other Java naming conventions:
  - Packages are all lowercase
  - Local variables, member variables, and methods start with lowercase letters
  - Classes, enums, and interfaces start with uppercase letters
  - Constants are written in snake case with ALL CAPS

Older naming conventions

- Most languages do not have meaningful limitations on variable name length now, but they used to
- Older C code in particular often leaves out vowels to save space
- Hungarian notation is naming conventions that describe the types of variables with prefixes:
  - `wParam` (word-sized parameter)
  - `pfData` (pointer to a floating-point value of data)
  - `lpszName` (long pointer to a zero-terminated string)
- Hungarian notations can also be used to specify scopes:
  - `g_nGoats` (global integer for number of goats)
  - `m_nBoats` (member variable integer for number of boats)
- These conventions have largely been given up, since IDEs provide tools for keeping track of types and scopes
  - Also, languages likes Java and C# have much stronger type-safety than C and C++, giving compiler errors for misusing types

Layout conventions

- Many languages (with the notable exception of Python) ignore whitespace
- Thus, we have a choice about how to layout our code
- In C-family, curly brace languages, it's common to put the opening brace of an **`if`** statement, method, or loop either on the same line as the header (K&R style) or on the next line (Allman style)
  - K&R is more common for Java, but Allman is more common for C#
- Some people also have strong feelings that indentation should be tabs while others prefer spaces
- A common convention is that lines of code should not exceed 80 characters

K&R style

```
if (raining) {
	System.out.println("I'm wet!");
}
```

Allman style

```
if (raining)
{
	System.out.println("I'm wet!");
}
```

Commenting

- Almost every language allows for comments
- Code that is so easy to understand that it needs no comments is called **self-documenting code**
  - Ideally, all code is self-documenting, but this goal is rarely reached
- Perhaps the other end of the spectrum is **literate programming**, which explains everything in English mixed in with the code, taking the perspective that code is for humans to understand and only incidentally for computers to execute
- Commenting should explain confusing code, especially unusual algorithms

Good commenting

- **Do** use comments to describe the intent of a complicated piece of code
- **Do** use comments to explain the rationale behind a decision so that people can understand in the future
  - Why this way?
  - Why not that other way?
- **Do** use comments to reference relevant outside documents
  - Explanation of an algorithm
  - API documentation page
  - Design document with UML diagrams

Questionable commenting

- **Don't** use comments to repeat the code
- Be careful about using comments for to-do items and future work
  - Especially if it means you don't do the right thing now
- It is possible to over-comment, so consider whether the supplemental information is useful

Bad comments that repeat the code

```
// Increase i by 1
++i;

// Include sales[i] in the total
total = total + sales[i];
```

Quiz

Upcoming

Next time…

- Work day on Friday
- We'll talk about quality assurance in construction on Wednesday
  - Since Monday is break!

Reminders

- Read Chapter 8: Quality Assurance in Construction for Wednesday
- Finish the draft of Project 2
  - **Due Friday!**