# Structured Programming

# COMP1110/6710

Australian National University

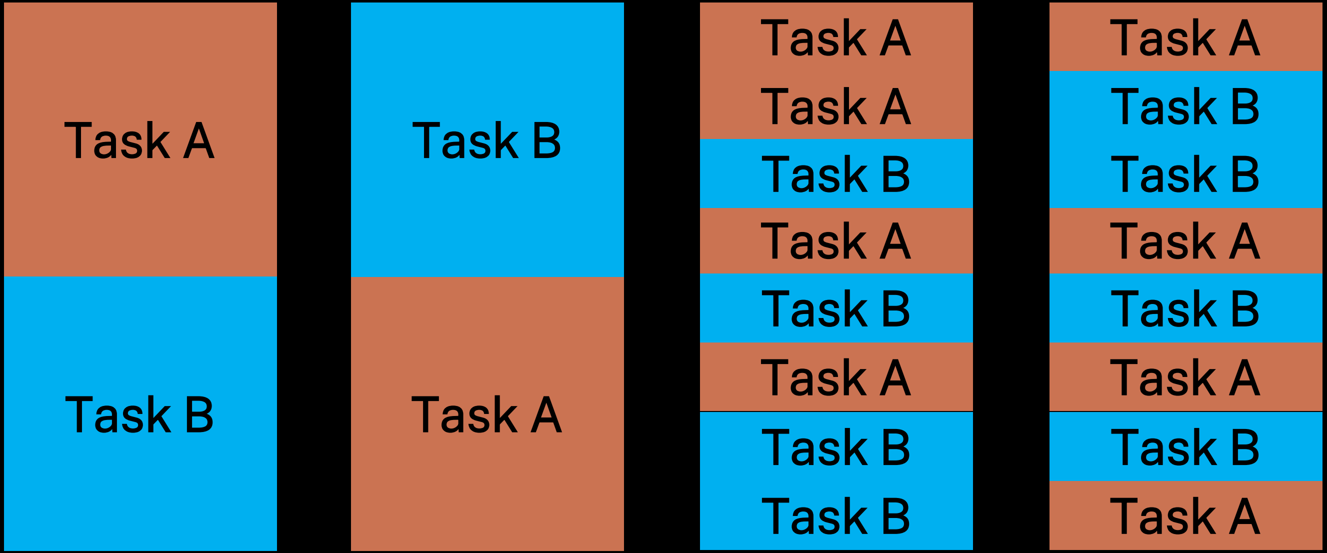
Image Courtesy NASA/JPL-Caltech.

# Concurrency

Computers are not perfect at multitasking, either

# Doing Things Concurrently

Means: not necessarily doing them in a fixed order, or in order at all.



Usually, there needs to be some order, so within task A and task B, the order of operations stays the same.

# We See That All The Time

Your computer is likely running several programs concurrently right now.

Your phone is likely running several programs concurrently right now.

Your watch/TV/dishwasher/... may be running several programs concurrently right now.

In fact, they may be running those programs in parallel.

# Concurrency vs. Parallelism

Concurrency … doing things not necessarily in order, but independently of each other. E.g. also time-slicing.

Parallelism … doing things actually at the same time, say with multiple processors/cores.

Most problems that need solving already come in with concurrency; parallelism just makes some problems more acute (e.g. without parallelism, you can assume that only one “basic” operation can happen at the same time, but that’s usually not very helpful).

## Concurrent Processes

Process … a separate program executed on your computer. The task of the operating system is to ensure that all processes can work independently while sharing hardware resources and files.

[Demo – two Java process read and write a file]

# Concurrency within Programs

A single process may have multiple “threads”. Each thread has its own stack, but all threads share the same heap.

(similarly, each process has its own memory (stack(s) + heap), but they all share the same file system)

[Demo: A Java program with multiple threads]

## Creating a new Thread

```
interface Runnable {
   void run();
}
```

```
Runnable r = …;
Thread t = new Thread(r);
t.start();


//Optional:
//wait for t to finish
t.join();
```

## Problem:

There are lots of little steps in evaluating expressions.

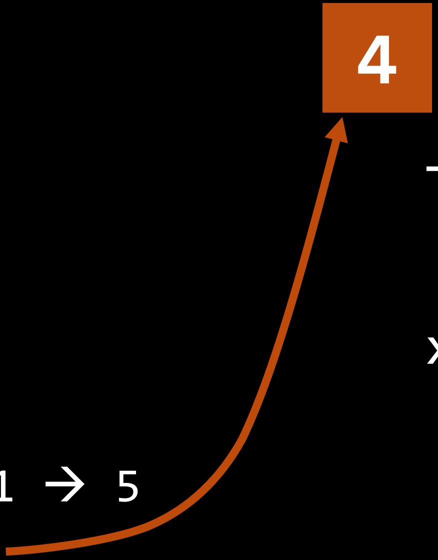
For example, while evaluating

```
x = x + 1;
```

We first calculate x + 1, and then write the result back to x.
But during that calculation, x still has its old value.

## Problem:

Global variable x



Thread 1:

```
x = x + 1;
```

1. Retrieve x → 4
2. Calculate `x + 1 → 5`
5. `Store 5 in x`

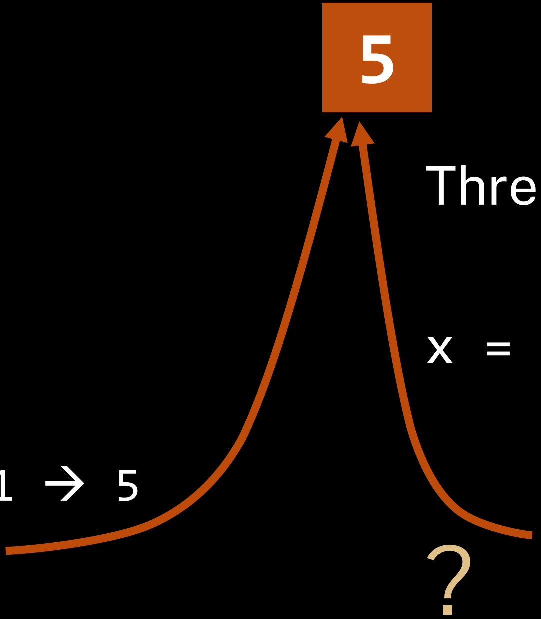
Thread 2:

```
x = x * 2;
```

3. Retrieve x → 4
4. Calculate `x * 2 → 8`

## Problem:

Global variable x



Thread 1:

```
x = x + 1;
```

1. Retrieve x → 4
2. Calculate `x + 1 → 5`
5. `Store 5 in x`

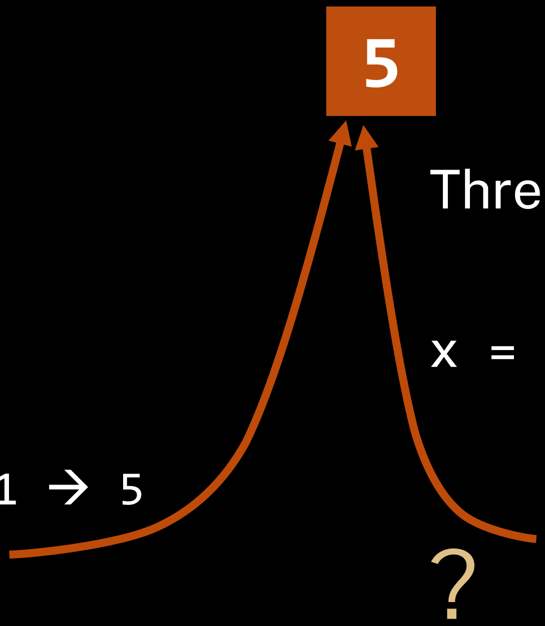
Thread 2:

```
x = x * 2;
```

3. Retrieve x → 4
4. Calculate `x * 2 → 8`
6. `Store 8 in x`

## Problem:

Global variable x



Thread 1:

```
x = x + 1;
```

1. Retrieve x → 4
2. Calculate `x + 1 → 5`
5. `Store 5 in x`

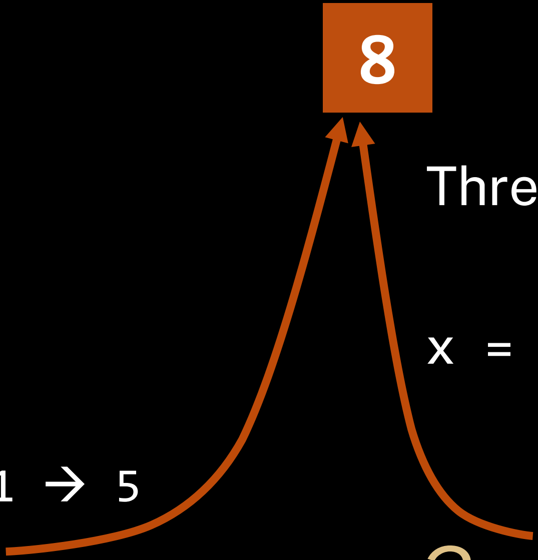
Thread 2:

```
x = x * 2;
```

3. Retrieve x → 4
4. Calculate `x * 2 → 8`
6. `Store 8 in x`

# Problem:

Global variable x



Thread 1:

```
x = x + 1;
```

1. Retrieve x → 4
2. Calculate `x + 1 → 5`
5. Store `5 in x`

"Lost update"

Thread 2:

```
x = x * 2;
```

3. Retrieve x → 4
4. Calculate `x * 2 → 8`
6. Store `8 in x`

?

# Solution: Mutual Exclusion

In Java, via synchronized-blocks:

```
synchronized(sharedLockObj) {
    …
}
```

Reference to some object on the heap

Only one thread can be in a synchronized-block (also called a “critical section” for the same heap object at the same time. Others have to wait.

Shorthand:

```
synchronized [Type] foo(…) {
    …
}
~
[Type] foo(…) {
    synchronized(this) {
        …
}}
```

Or special “class” object for static methods

## Synchronized

Global variable x

4

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```

Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

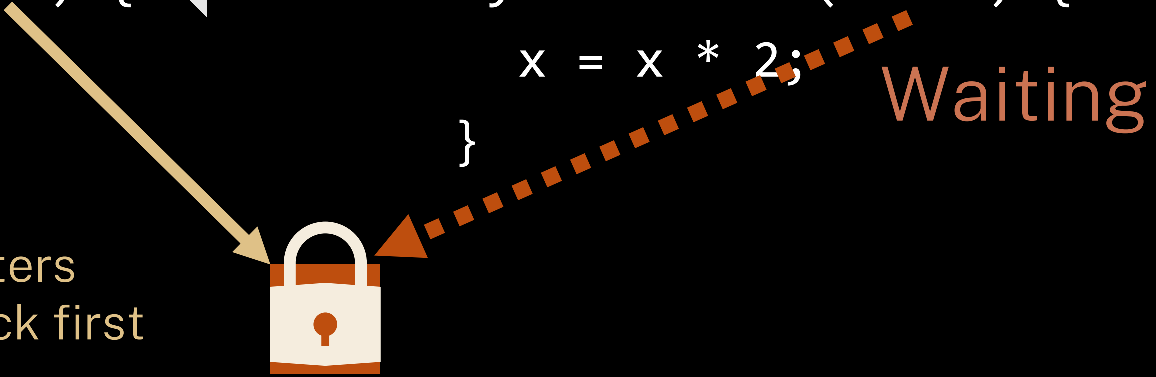
4

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```



Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

5

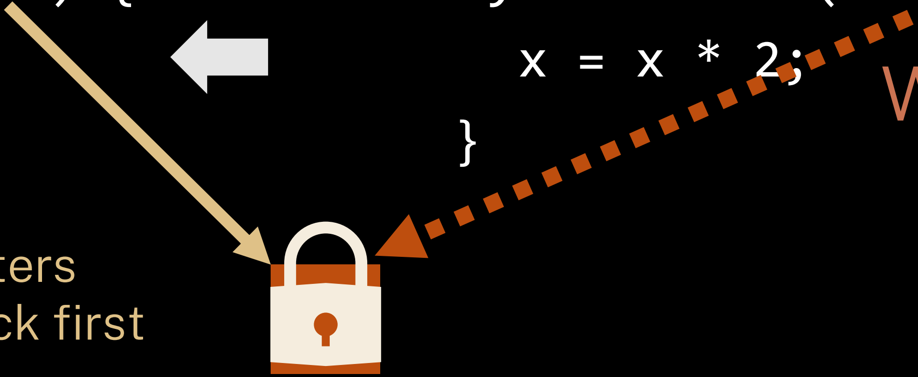
Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```

Waiting



Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

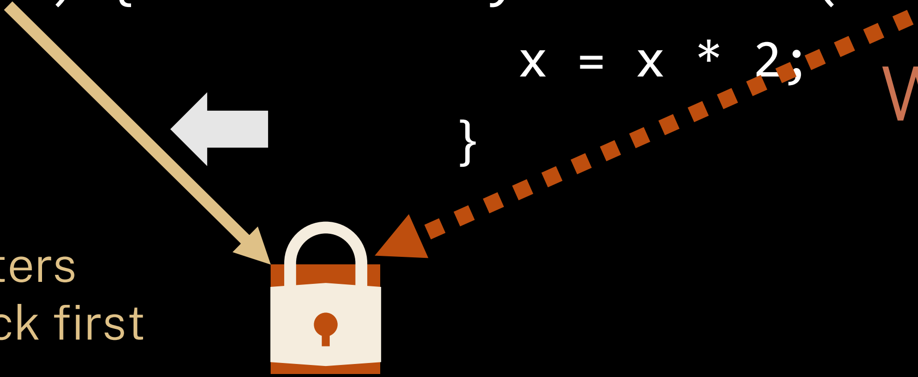
5

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```



Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

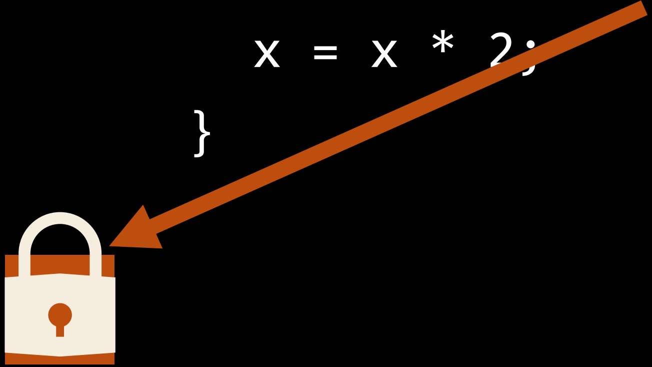
5

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```



Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

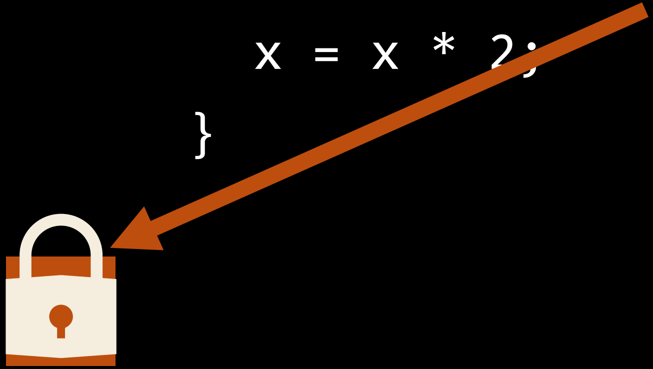
10

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```



Whichever thread enters the synchronized block first gets to finish it; the other thread has to wait

Global variable `xlock (= new Object())`

# Synchronized

Global variable x

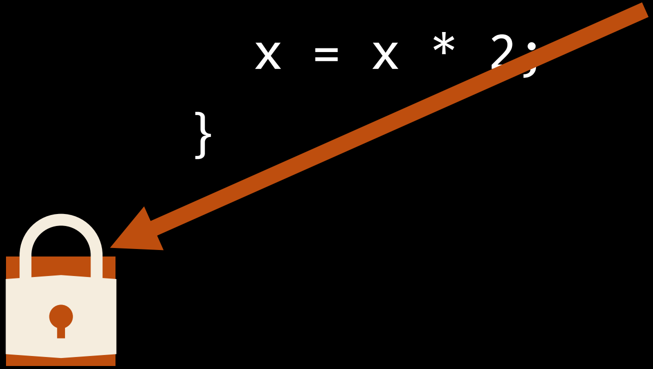
10

Thread 1:

```
synchronized(xlock) {
  x = x + 1;
}
```

Thread 2:

```
synchronized(xlock) {
  x = x * 2;
}
```



Still two possible values for x, but at least not nore than that

Global variable `xlock (= new Object())`

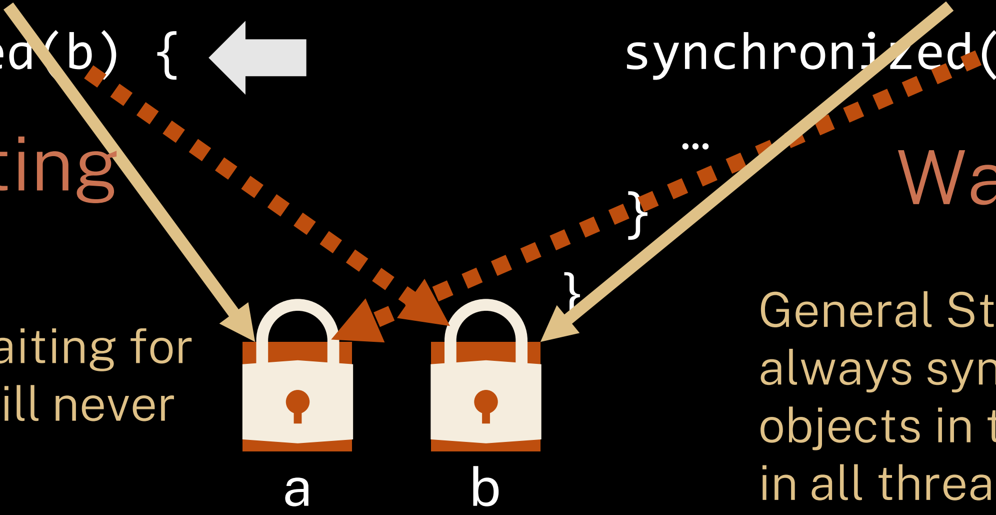
# Beware: Deadlocks

Thread 1:

```
synchronized(a) {
  synchronized(b) {
    …
  }
}
```

Thread 2:

```
synchronized(b) {
  synchronized(a) {
    …
  }
}
```



Threads are waiting for each other – will never continue…

General Strategy: always synchronize on objects in the same order in all threads

## Threads - Applications

- Dividing up work, where N threads can do work ~N times as fast
- Responsive interfaces, running background work separately
- Servers that handle multiple clients

## Threads - Summary

- Concurrency arbitrary interleaves actions from different Threads
- This may produce weird results, or violate invariants
- Use synchronization constructs to limit the possible interleavings
- Beware of over-synchronizing:
  - Danger of deadlocks
  - Loss of valid concurrency

# Structured Programming

The Grand Tour

A ONCE IN A LIFETIME GETAWAY
THE GRAND TOUR
JUPITER / SATURN / URANUS / NEPTUNE
EXPERIENCE THE CHARM OF GRAVITY ASSISTS
EVERY 175 YEARS
NOW BOARDING

Australian National University

## What Are We Here For?

Students will learn to use an industrial-strength object-oriented programming language and form basic mental models of how computer programs execute and interact with their environment. The course focuses on key aspects of solving programming problems: reasoning about a problem description to design appropriate data representations and function/method descriptions, to find examples, to write, test, debug, and otherwise evaluate the relevant code, and to present and defend their approach.

# What Are We Here For?

form basic mental models of how computer programs execute and interact with their environment.

- The Command Line
- Command Line Arguments
- Printing / Reading from Standard Input
- File Systems / File Operations
- Compile-Time vs. Run-Time

- Expressions, Statements & Functions
- Control Flow:
    - Left-to-Right, Inside-Out, Top-Down
    - Control Flow Constructs
- Mutable State / Stack vs. Heap

- Testing & Debugging

## Working Code

Ultimately, we want to get a computer to do something for us.

Computer Science is about all aspects of how to best do that.

This requires:

- Attention to details – computers are sticklers for details
- Understanding requirements
- Knowing how to run an test your code

# “It Works” Is Not Enough

Code is not just written for computers to run!

It is also written for people* to read

- When they want to convince themselves that it does what it should
- When they need to add/change something
- When they just want to know how/why it works

* This includes you a week later

## What Are We Here For?

The course focuses on key aspects of solving programming problems: reasoning about a problem description to design appropriate data representations and function/method descriptions, to find examples, to write, test, debug, and otherwise evaluate the relevant code, and to present and defend their approach.

Sound familiar?

# Data Design

***The shape of the data determines the shape of the code!***

How you represent the data that you work with hugely influences how easy or hard certain parts of your desired functionality will be to implement, and other characteristics of your code.

# Data Design - Interpretations

We did this from the start:

Overall purpose of data definition

```
/** A Student is a record that contains key information about an
  * ANU student. Examples:
  * - Lisa Studywoman, u1234567, BAC
  * - Paul Masterson, u7654321, MCOMP
  * @param name – the name of the student, a non-empty String
  * @param uid – a UID, identifying the student
  * @param program – the student’s degree program
  */
record Student(String name, String uid, String program) {}
```

Example values

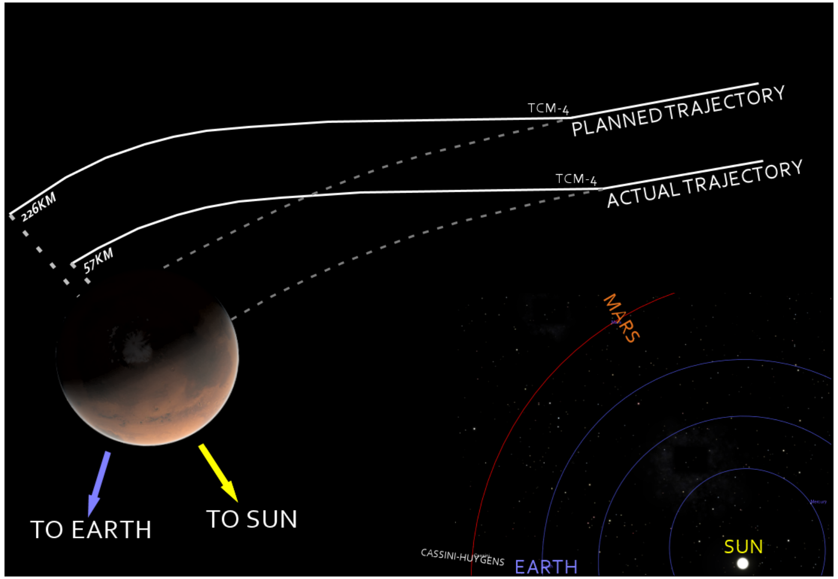
Explanations of fields

javadoc format, useful for later

The fields of the record

# Data Design - Interpretations

Why?
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure



Specifically, software that calculated the total impulse produced by thruster firings produced results in **pound-force seconds.** The trajectory calculation software then used these results – expected to be in **newton-seconds** (incorrect by a factor of 4.45) – to update the predicted position of the spacecraft.

According to NASA, the cost of the mission was $327.6 M ($571.41 M in 2023)

## Data Design – The Expression Problem

Functional style:
sealed interfaces + records

- Easy to add new functions
- More effort to add new kinds of data

OO style:
interfaces + classes

- Easy to add new kinds of data
- More effort to add new methods

# Data Design – ADT Implementations

Arrays/ArrayLists:

- Constant-time access
- Linear-time add/insert/remove

Linked Lists:

- Constant-time insertion/deletion
- Linear-time access

In general, trade-offs depend on application!
(In practice, ArrayList is usually the safer choice)

# Data Design - Abstraction

In Functional Java: usually via generalized Itemizations

At the core of Object-Oriented programming:

- Function Abstraction
- Type Abstraction
- Data Abstraction

All in one!

- Interfaces & Abstract Classes
- Encapsulation/Access Modifiers
- Iterators

*The shape of the interface determines the shape of the code!*

# Signature & Purpose Statement

*The shape of the interface determines the shape of the code!*

The "external" interface is often fixed, because other people depend on it.

But how you design your helper functions makes a huge difference.

**Don't repeat yourself!** Consider the possibilities for abstraction:

- Value Abstraction
- Type Abstraction
- Functional Abstraction
- Subtyping/Inheritance

But don't abstract prematurely!

As always, document your choices so other people (including you in the future) understand what's going on!

## Examples

The first key test: do you understand what should be happening here?

Having concrete examples both guides you in what code to write, and how to evaluate it afterward.

Key point: think of corner-cases! Does the function behave uniformly over all inputs, or are there differences? How do they play out?

# Design Strategy

Going forward, this is the least critical part of the Design Recipe.

Its two main purposes were:

- To give you a small menu of options to choose from, making it easier to make decisions
- To limit the overall size and complexity of the functions you write. This should be the main legacy of this step for you: keep the various parts of your code as simple, small, and independent as possible!

## Implementation

See other slides ☺

# Tests

Remember: Working Code

Tests are no guarantee that your code is perfect, but if you do it well, you can find lots of potential bugs (or better: see that they are not there).

Tests also validate your examples, confirming your understanding of the overall task.

# Working with Arbitrary-Sized Data

## At the core of every interesting program

### (Structural) Recursion

- Follows the shape of the data
- Easy to reason about
  - Base Cases
  - If you know the solutions for smallers case, how do you combine them for a bigger case?
- Unwieldy with some stateful data structures

### Iteration

- More natural when working with arrays or generative recursion (when there is no obvious data structure to traverse)
- Less easy to reason about
- Think about termination!

# Mutable State

- Very common, but also common source of bugs
- Know what’s where: the stack or the heap?
- Be clear about the invariants of your data definitions!
- Be clear about the effects your code has on shared state!

# Mutable State – A Journey

Every variable stands for a single value within its scope

Every variable stands for a box whose contents may change, and the things they point to may change, too

We revisited this for:
- Generics
- Closures
- Concurrency

Things in those boxes may change while you are working on them 😮

More fun in later courses!

# Polymorphism / Type Abstraction

Subtyping

- Core to OO
- Abstracts commonalities in interfaces – concrete type does not matter when using interface
- Dynamic Dispatch/Overriding
- Loses static type information

Generics

- Preserves static type information
- For when actual types don’t matter to generic code, but to the user
- In Java: compile-time only

## Iterators

Not that interesting on their own.

But they combine:

- Mutable State
- Iteration
- Multiple Kinds of Abstraction

An excellent way to reinforce all those concepts!

## Abstract Data Types

### Key common pattern for most programs

Really: three difficulty levels

Lists/Maps:

- Usually simple Iteration/Recursion
- Very standardized

Trees:

- More general structure
- Still relatively standard recursion, but also BFS/DFS
- Lots of different shapes

Graphs:

- Yet More general structure
- More complicated traversal
- Possible loops

## Time Complexity

- Standard Interview Question
- Key consideration in large system design
- Important active research areas in Computer Science
- Huge difference in costs!

## SELTs

Plase fill them out (opens next week)!

In giving qualitative feedback, please be specific about issues you had and things you think would improve the course!