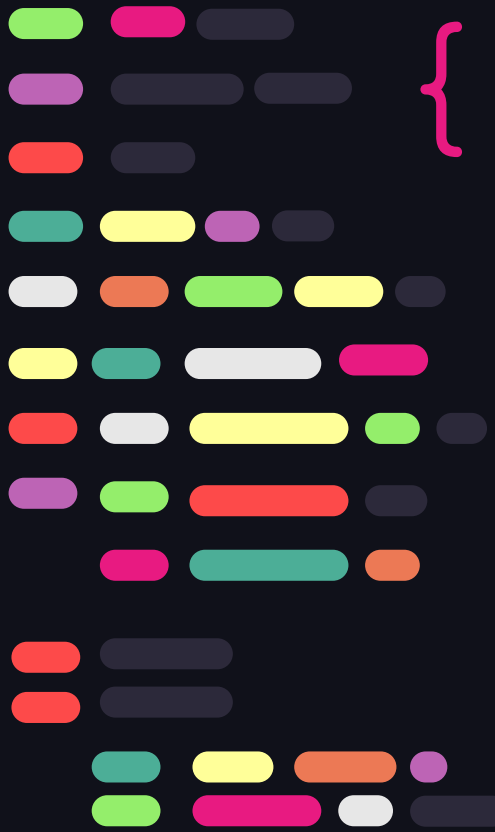


CS2030
Lab 1

AY24/25 Sem 2, Week 3

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Pang Yang Yi <pang.yy@u.nus.edu>

Introduction

Name: Fadhil
Year & Major: Y4 Chemistry
Hobbies: Photography, Grand Strategy Games

Introduction

Name: Yang Yi
Year & Major: Y2 Information Security
Hobbies: Reading

Session Overview

- Admin Stuff
- Recap (Imperative vs Declarative, Streams)
- Lab Task Overview

Admin Stuff

Log in to the lab device

Username: nusstu\exxxxxxxx (e.g. nusstu\e1234567)
Password: <your canvas password>

Make sure that you are logged into **your** account and not someone else's, or you will be marked absent!

What does CS2030 teach you?

Java?

Vim/Micro?

UNIX?

Terminal?

What does CS2030 teach you?

~~Java?~~

~~Vim/Micro?~~

~~UNIX?~~

~~Terminal?~~

Program Design

Programming Paradigms

What does CS2030 teach you?

~~Java?~~

~~Vim/Micro?~~

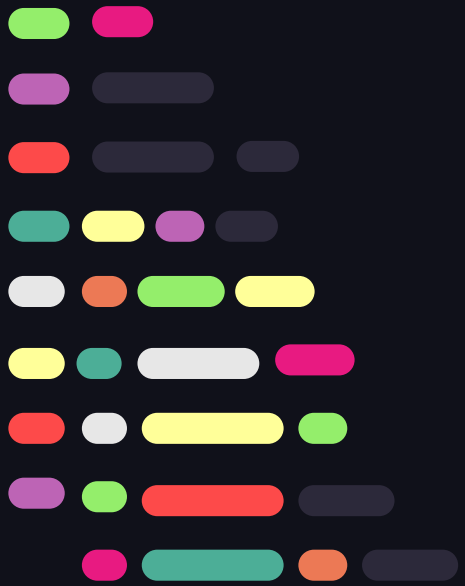
~~UNIX?~~

~~Terminal?~~

Program Design

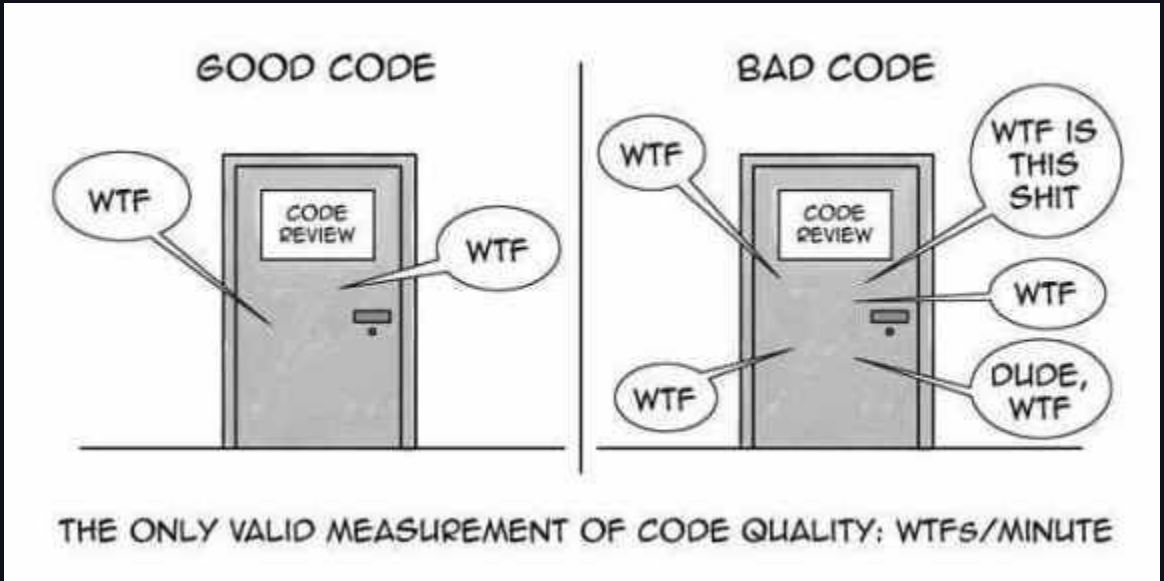
Programming Paradigms

Remember that what you learn here isn't perfect (and that's ok!)



“Programs are meant to be read by humans and only incidentally for computers to execute”

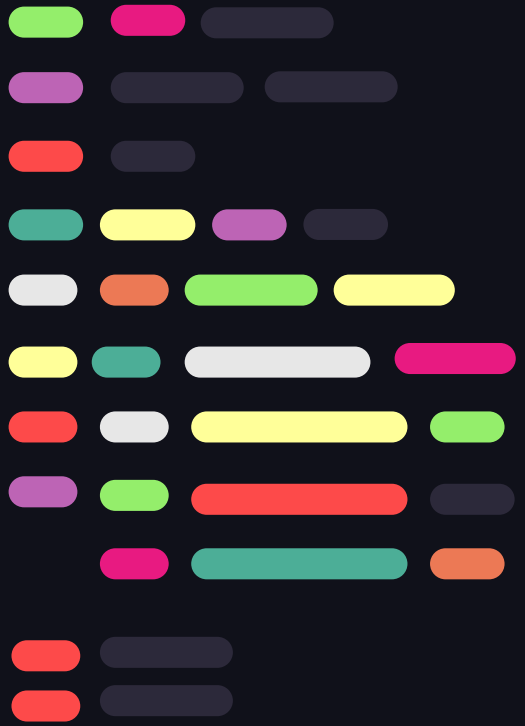
— Harold Abelson and Gerald Jay Sussman


GOOD CODE
BAD CODE
WTF
CODE REVIEW
WTF
WTF
WTF IS THIS SHIT
CODE REVIEW
WTF
WTF
DUDE, WTF
THE ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE

Admin Stuff

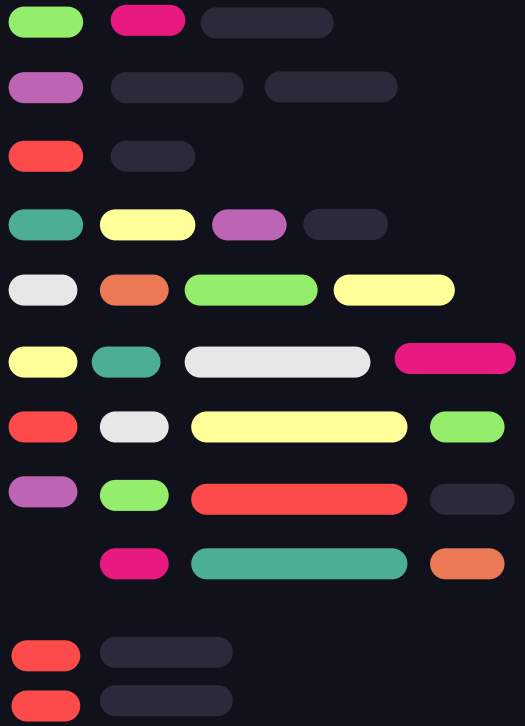
Finals	40%
Individual Project	15%
Practical Assessment 1	15%
Practical Assessment 2	20%
Labs, self-practice exercises, class participation and peer learning activities	10%

Setting Expectations



- Attendance is graded
- Attendance is automatically taken upon login within first hour
- Considered absent if late by more than an hour

Setting Expectations



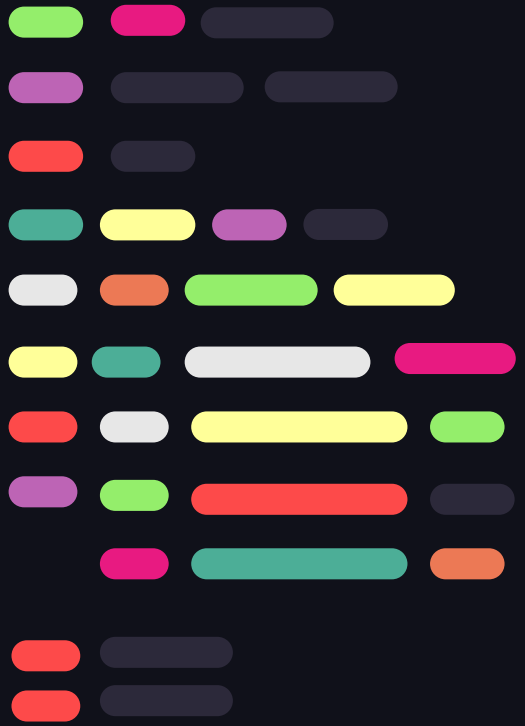
What your TAs are professionally obliged to do:

- Run lab sessions
- Provide help during lab sessions
- Provide feedback

What your TAs are **NOT** expected to do:

- Be on call 24/7 to answer your queries
- Be your personal debugger/encyclopedia for all things CS2030 related

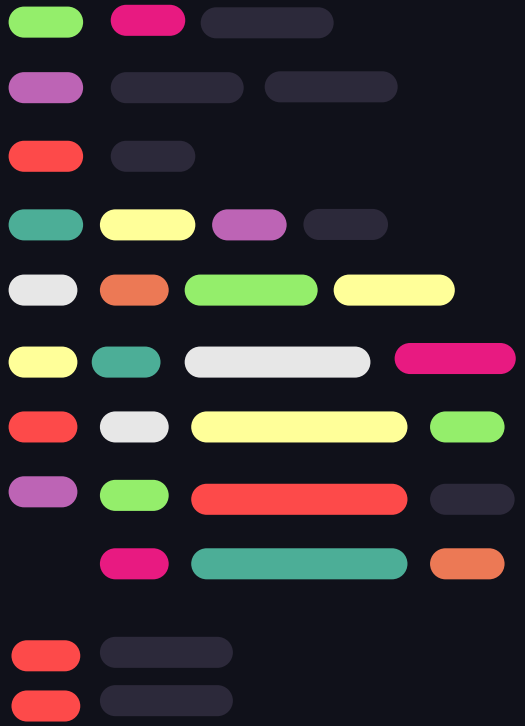
Setting Expectations



What your TAs will (*try to*) do for you:

- Help you to bridge the learning curve
- Give you good/constructive/candid comments about your work and progress
- Answer your CS2030-related (*or non-related*) queries outside of lab sessions

Setting Expectations



What we **cannot** tolerate:

Any form of academic dishonesty

This includes any form of code plagiarism, cheating, copying, etc.

Plagiarism

Plagiarism is a VERY serious academic offense.

NUS Plagiarism Policy states that the minimum penalty for cases of plagiarism and cheating in tests/examinations/graded assignments that have been assessed to be of ‘Moderate’ severity would be that of a `Fail` grade for the affected course.

https://myportal.nus.edu.sg/studentportal/student-discipline/all/docs/NUS-Plagiarism-Policy.pdf

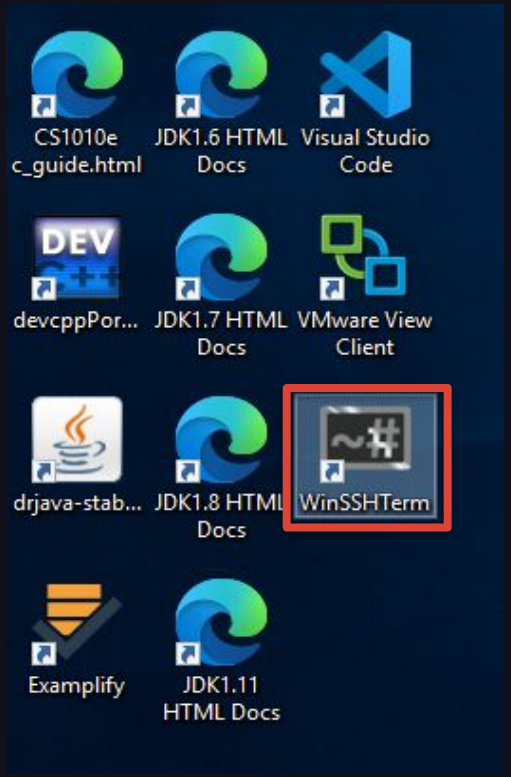
Plagiarism

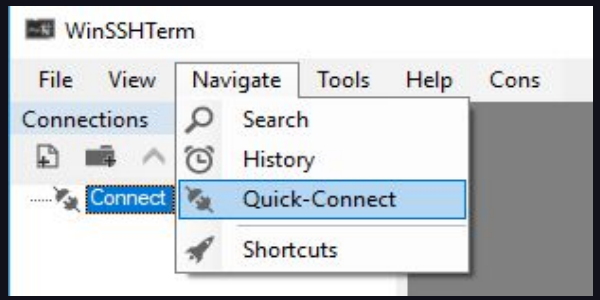
Your lab submissions should always be done independently.

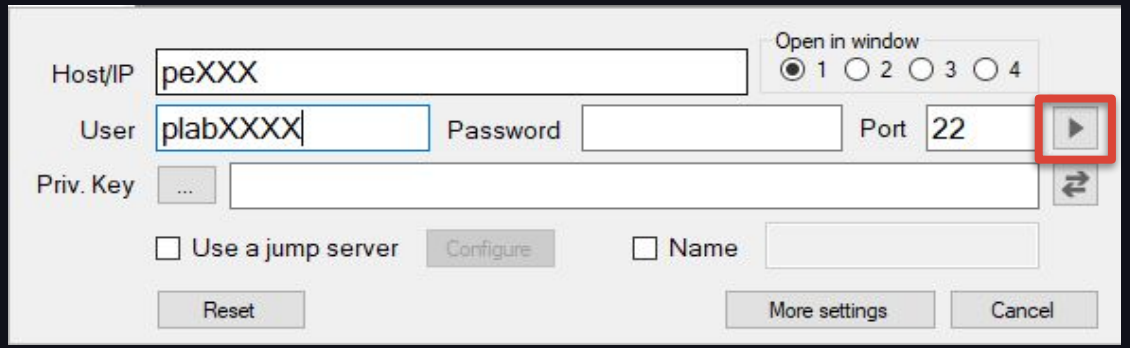
Do **NOT** share your code with others - Discussions are fine, but you will never know if they blatantly copy-paste your code and you become complicit to the plagiarism offence.

Lab Admin

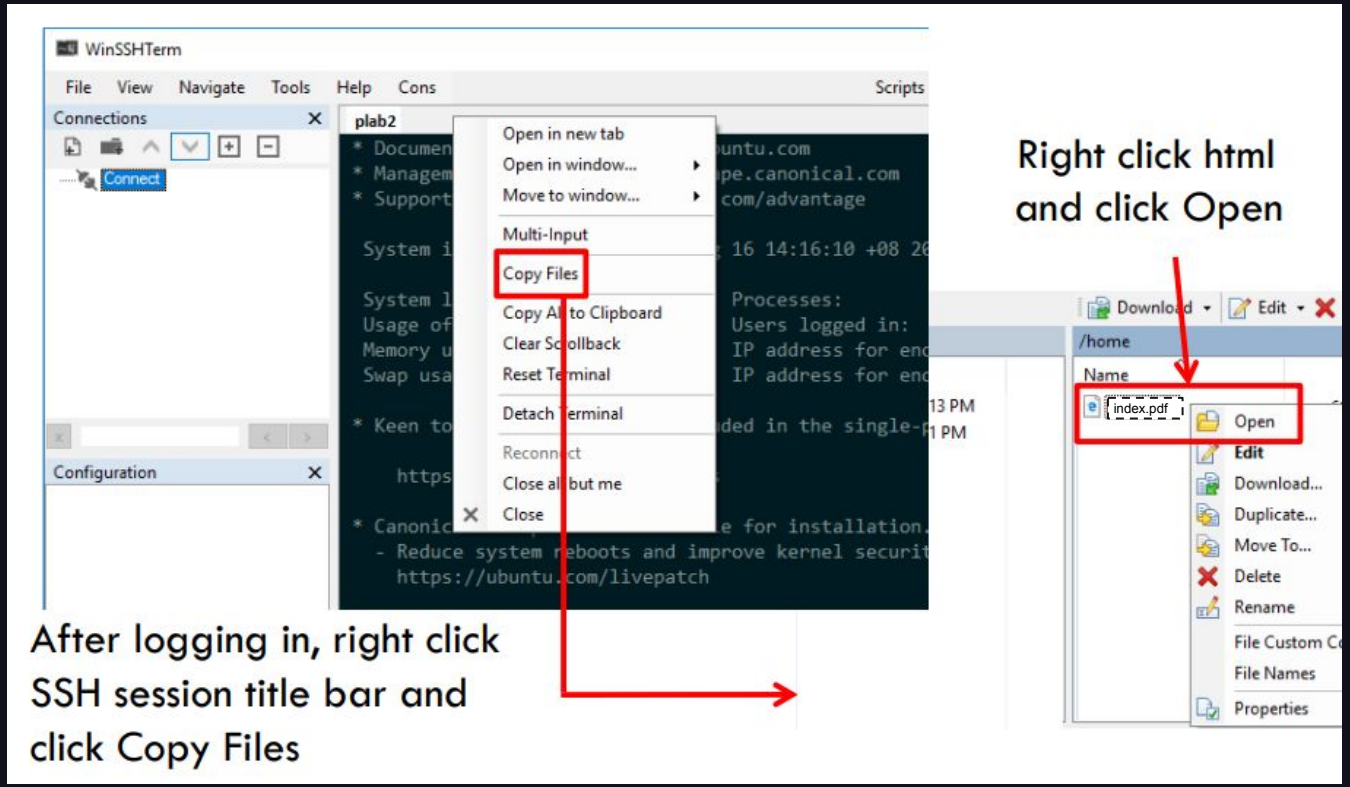
Connecting to PE Node







Viewing Questions



Questions are in the file labelled **index.pdf**

Lab Sessions

During the lab:

- You are highly advised to use vim/micro on the lab device
- Do not submit any code to CodeCrunch
- Compile your code frequently throughout the session
- When done, save your files, quit your editor, ensure your files are all intact before quitting SSH

Lab Sessions

During Practical Assessments, only code written in the PE node (via WinSSHTerm) will be saved. As such, **you should practice** with vim/micro during labs.

If you are not able to get anything to work during the PAs due to a lack of practice, the teaching team is not responsible for any consequences.

So,

Please take your labs seriously, as practice for your PAs

Code Submission

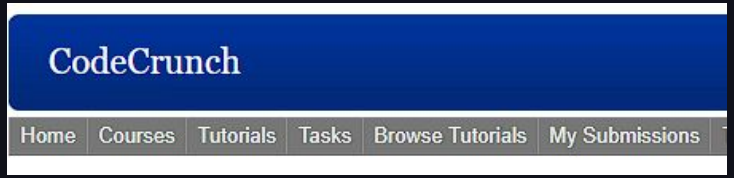
Once done, simply leave the files in the PLAB account

- Remember to save your files and quit VIM/Micro
- Check all your files are there with the correct content with **ls** (or **dir**) and **cat**
- Quit SSH by typing **exit**

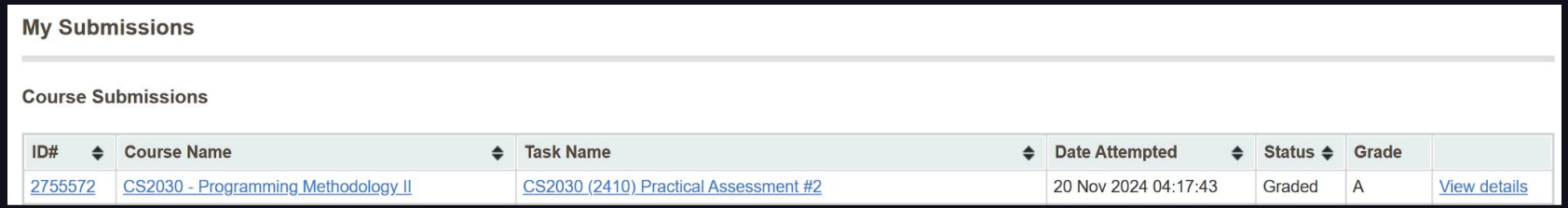
Code will be automatically uploaded to CodeCrunch by the end of the day, download your files from CodeCrunch and carry on.

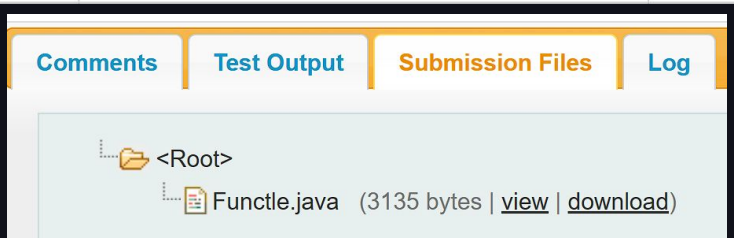
CodeCrunch

https://codecrunch.comp.nus.edu.sg/



Log in
Click ‘My Submissions’
Click ‘View details’ on the task
You will find all your files under ‘Submission Files’





Code Submission (tl;dr)

1. You code in the PE node during lab
2. Your code will be uploaded to CodeCrunch by the end of the day (provided you code in the PE node)
3. Download your files and continue working
4. Submit your solution by the deadline (usually 2359 of the night before the next lab)


Why Java?

Why Java?


Java Setup - Progress
Java
ORACLE
Status: Registering Java Runtime Environment
3 Billion Devices Run Java
Computers, Printers, Routers, Cell Phones, BlackBerry, Kindle, Parking Meters, Public Transportation Passes, ATMs, Credit Cards, Home Security Systems, Cable Boxes, TVs...
ORACLE

Why Java?

Write once, run anywhere

- Unlike programs in other languages, Java is not recompiled for different OSes and architectures
- Java bytecode runs on the JVM (Java Virtual Machine) instead
- Made Java incredibly popular in the 90s as the Internet took off
- **Popularised OOP as a consequence (which is why we're using it here)**

Why Java?

With great popularity comes great backlash

- Lots of code being written in Java also meant lots of bad code being written in Java
- Perceived stagnation
- Combined with Java’s verbosity, impressions of Java slowly soured

Why Java?

```
1 class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

Why Java?

Imitation is the sincerest form of flattery

- Running code on an intermediate virtual machine proved to be extremely popular
- Other languages that compiled to Java bytecode were created to address Java's shortcomings
- Yet others built their own virtual machines for their languages





Why Java?

Things have changed

- Over the last decade, Java's maintainers have tried to keep pace with modern developments, especially in FP.
- e.g. Lambda expressions and Streams (Java 8), JShell (Java 9), Records (Java 17), unnamed classes (Java 21)
- In this course we use **Java 21 LTS**

Why Java?

```
1 void main() {
2     System.out.println("Hello World!");
3 }
```

Why Java?

tl;dr

- Java may be old, but it still is a modern language that you are very likely to encounter outside
- Java is an OOP language but also supports FP (not as extensive as Scala/Haskell, but that makes it all the more useful as an educational tool)

Recap


Imperative vs
Declarative

Imperative Programming

Telling the compiler “how to do something”, step by step.

```java
for (int i = 0; i < n; i++) {
    System.out.println(i);
}
```

Here you are telling the compiler:

1 - get the value of i
2 - check if the value of i is less than n
3 - print the value of i out
4 - increment the value of i
5 - repeat

Declarative Programming

Writing code that describes what you want, but not how to get it

You avoid writing code to describe the lower-level implementation of what you want the code to do

In the previous example, you are saying that for range of 0 to n you want to print each value out

Streams

We will be making use of Streams to implement iteration in a declarative manner

This also means no for/while loops (use Streams to achieve this instead!)

Streams

There are three general forms of Stream operations:

1. Data Source Operations (*Where the Stream starts*)
2. Non-Terminal/Intermediate Operations (*What happens in the Stream*)
3. Terminal Operations (*Where the Stream ends*)

IntStreams - Start

A IntStream must start from somewhere:

```
IntStream.range(1, 3); // startInclusive, endExclusive
IntStream.rangeClosed(1, 3); // startInclusive, endInclusive
```

Streams - Start

A Stream must start from somewhere:

```
Stream.<Integer>of(1);
Stream.<Integer>of(1, 2, 3);
Stream.<String>of(“ILoveCS2030”);
Stream.<Integer>iterate(0,
                        x -> x < 10,
                        x -> x + 1);
```

Streams - Non-Terminals

Then you perform your intermediate operations on the Stream

These operations result in another Stream

Think of it as a chain of operations to be applied to the Stream

Streams - map

The map function applies a function to the values inside the Stream

```
map(func);
```

For example:

```
IntStream.range(1, 3).map(x -> x + 1);
// Returns a stream of 2 and 3
```

Streams - filter

The filter function returns a Stream<T> consisting of the elements of the current Stream<T> that match the given predicate

```
filter(predicate);
```

For example:

```
IntStream.range(1, 3).filter(x -> x > 1);
// Returns a stream of 2
```

Streams - filter

```
IntStream.rangeClosed(0, 10)      // 0, 1, .. , 10
         .filter(x -> x > 3)      // x must be more than 3
         .filter(x -> x < 7);     // x must be less than 7
```

This filters out anything that does not match your conditions.

```
// which means it returns an IntStream of 4, 5, 6
```

You can also write it like this:

```
IntStream.rangeClosed(0, 10).filter(x -> x > 3 && x < 7);
```

Streams - Terminals (End)

Once you’re done chaining your operations, you end the chain with a terminal operation.

Only at this step will the Stream pipeline operations start

No other operations can be chained after this one

Streams - Terminals (End)

Reduces the elements into a single return result based on the identity element and the accumulator. (Note that there are several overloaded methods for this)

```
reduce(identity, accumulator)
// the accumulator is a binary func that takes  - a partial result (type T) and the next element (which can be a different type U), and returns a new result of type T.

// in the case of IntStreams, both must be int
```

Streams - reduce

Combines the stream elements into a single result using the given operation on the next **value**, starting with the **identity** as the initial value

Streams - reduce

Using the given operation to combine **value** with **identity**:

```
IntStream.range(1, 4).reduce(1, (x, y) -> x + y);
(1, 2, 3).reduce(1, (x, y) -> x + y);
(2, 3).reduce(1, (1, 1) -> 1 + 1);
(2, 3).reduce(2, (x, y) -> x + y);
(3).reduce(2, (2, 2) -> 2 + 2);
(3).reduce(4, (x, y) -> x + y);
.reduce(4, (4, 3) -> 4 + 3);
7;
```

Streams - reduce

Using the given operation to combine **value** with **identity**:

```
|  Welcome to JShell -- Version 21.0.5
|  For an introduction type: /help intro

jshell> IntStream.range(1, 4).
   ...> reduce(1, (a, b) -> {
   ...>     System.out.println(String.format("adding %d to accumulated value %d", b, a)); // side-effect!
   ...>     return a + b;
   ...> })
adding 1 to accumulated value 1
adding 2 to accumulated value 2
adding 3 to accumulated value 4
$1 ==> 7
```

IntStreams

IntStreams are a sequence of primitive int-valued elements

This means that map functions expect an int to int function

For operations to convert the int into other types, use the mapToObj function instead

IntStreams - examples

Given a list, e.g. myList = {1, 2, 3, 4, 5}

```
IntStream.range(0, myList.size())
         .map(x -> myList.get(x))
         .filter(x -> x > 2);
// filters list elements and keeps only elements > 2
```

IntStreams - Example 1

Given a list, e.g. myList = {1, 2, 3, 4, 5}

```
IntStream.range(0, myList.size())
         .map(x -> myList.get(x))
         .reduce(0, (x, y) -> x + y);
// sums elements of the list
```

IntStreams - Example 2

This also works for a list of Strings, with a small tweak,

e.g. myList = {“a”, “b”, “c”, “d”, “e”}

```
IntStream.range(0, myList.size())
         .mapToObj(x -> myList.get(x))
         .reduce(“”, (x, y) -> x + y);
// the reduce here is essentially string concatenation
```

IntStreams - Example 3

```
List<String> ls = List.of(“a”, “b”, “c”, “d”, “e”)
String toInsert = “z”;
IntStream.range(0, ls.size())
          .mapToObj(x -> x == 3 ? toInsert : ls.get(x));
          .toList();
// this returns an immutable List, just like List.of!
```

IntStreams - Example 4

```
List<String> ls = List.of(“a”, “b”, “c”, “d”, “e”)
String toInsert = “z”;
IntStream.range(0, ls.size())
         .mapToObj(x -> x == 3 ? toInsert : ls.get(x));
         .collect(Collectors.toList());
// this returns a mutable ArrayList and is not allowed
```

Why IntStream?

Tip

When designing solutions with Streams, think of each step in the pipeline as a building block that incrementally brings you closer to the final result.

Focus on breaking down the problem into smaller transformations that can be represented as stream operations

Tip

Some questions to ask yourself:

- How long does the (initial) Stream need to be? //decides your start
- What transformations do I need to do to the elements to achieve the desired results? //decides your intermediate operations
- What is the end result/required output? //decides your terminal

Lab 1

Task Overview

Lab will be **three** different applications of Streams, make use of the aforementioned methods to solve the levels

Avoid Arrays as they are mutable structures

Task 1: Twin Primes

Twin Prime: One of a pair of prime numbers with a difference of 2
- e.g. 3 and 5, 5 and 7, 41 and 43 etc.

Define a `twinPrimes` method that takes in an integer `n` and returns an IntStream comprising of distinct twin primes from 2 to n

Hint: You don't have to write everything in one method

Task 2: Reverse String

Define a `reverse` method that takes in a string and reverses it

You should start by streaming the appropriate indices (indices should go in the correct/normal order, so 0, 1, 2, 3 etc.)

Task 3: Counting Repeats

Define a countRepeats method that takes in a list of integers (from 0 to 9) and returns the number of occurrences of adjacent repeated digits.

Hint: You need only look at every three consecutive digits to decide if a repeat has occurred //ask yourself: “how do I look at consecutive digits?”

Task Overview

Deadline: **6 Feb (Thurs) 2359**