# Welcome to CS2030 Lab 7!

25 March 2022 [10J]

Only scan if you're physically present!
LOGIN TO PLAB NOW!

# PA2 Alternative Arrangements

- PA2 falls on the Friday of Week 12 (**08/04/2022**)
- Please let us know if you have to take tests immediately **before** or **after** your CS2030 lab (e.g. if you have lab from 1000-1200 and your test is at 1400 then it's not counted)
- We will book venues for you to take your test **if you need them** (PA2 will end around 10-15 minutes before the end of your lab slot so you can go elsewhere to take the test if you prefer)

# Bake an imperative 🍰!

1. Preheat oven to 175 degrees C. Grease and flour 2 – 8 inch round pans. In a small bowl, whisk together flour, baking soda and salt; set aside.

2. In a large bowl, cream butter, white sugar and brown sugar until light and fluffy. Beat in eggs, one at a time. Mix in the bananas. Add flour mixture alternately with the buttermilk to the creamed mixture. Stir in chopped walnuts. Pour batter into the prepared pans.

3. Bake in the preheated oven for 30 minutes. Remove from oven, and place on a damp tea towel to cool.

# Bake a functional 🍰!

1. A cake is a hot cake that has been cooled on a damp tea towel, where a hot cake is a prepared cake that has been baked in a preheated oven for 30 minutes.
2. A preheated oven is an oven that has been heated to 175 degrees C.
3. A prepared cake is batter that has been poured into prepared pans, where batter is mixture that has chopped walnuts stirred in. Where mixture is butter, white sugar and brown sugar that has been creamed in a large bowl until light and fluffy…

```
cake = cooled(removed_from_oven(added_to_oven(30min, poured( greased(floured(pan)),
stirred(chopped(walnuts), alternating_mixed(buttermilk, whisked(flour, baking soda, salt),
mixed(bananas, beat_mixed(eggs, creamed_until(fluffy, butter, white sugar, brown sugar)))))))
```

# Learning Objectives

**Idea Behind this Week's Stream Exercises:**

1. Understand the `IntStream` and `Stream` APIs
2. Apply what we have learnt with regards to stream operations

**Rules:**

1. No Recursion
2. No For Loops, No While Loops

**What you are allowed:**

1. Helper Classes, If Necessary
2. Helper Methods

With a stream, you specify what you want to have done, not how to do it. You leave scheduling of operations to the implementation.

The stream library can optimize the computation, e.g. using multiple threads for computing sums and counts and combining the results.

Instead of using imperative programming,

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

we want to use declarative programming.

```
IntStream.range(0, n).
    forEach(System.out::println)
```

```
static IntStream range(int startInclusive, int endExclusive)
static IntStream rangeClosed(int startInclusive, int endInclusive)
```

Generate elements from `m` to `n-1`

```
jshell> IntStream.range(1,5).toArray()
$1 ==> int[4] { 1, 2, 3, 4 }
```

Generate elements from `m` to `n`

```
jshell> IntStream.rangeClosed(1,5).toArray()
$2 ==> int[5] { 1, 2, 3, 4, 5 }
```

```
static <T> Stream<T> of(T... values)
```

Create a stream whose elements are the given value(s).

This method has a varargs parameter, so you can construct a stream from any number of arguments.

```
jshell> Stream.of("gently", "down", "the", "stream").
    forEach(e → System.out.print(e + " "))
gently down the stream
```

```
Stream<T> limit(long maxSize)
```

yields a stream with up to `maxSize` of the initial elements from this stream.

```
jshell> Stream.of("gently", "down", "the", "stream").
    limit(2).
    forEach(e → System.out.print(e + " "))
gently down
```

```
Stream<T> distinct()
```

yields a stream of the distinct elements of this stream.

```
jshell> Stream.of("merrily", "merrily", "merrily", "gently").
    distinct().toArray()
$5 ==> Object[2] { "merrily", "gently" }
```

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Return a stream containing the results of applying `mapper` to the elements of this stream.

```
jshell> Stream.of("gently", "down", "the", "stream").
    map(String::toUpperCase).toArray()
$6 ⟹ Object[4] { "GENTLY", "DOWN", "THE", "STREAM" }
```

```
<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
```

🤯 Returns a stream obtained by concatenating the results of applying mapper to the elements of this stream. (Note that each result is a stream.)

```
jshell> IntStream.range(1, 3).
    flatMap(x → IntStream.range(1, 4)).toArray()
$7 ⟹ int[6] { 1, 2, 3, 1, 2, 3 }

// Intermediate Operations
// (1, 2).flatMap(x → (1, 2, 3))
// flatMap((1, 2, 3), (1, 2, 3))
// (1, 2, 3, 1, 2, 3)
```

# Reductions (Terminal Operations)

Reduce the stream into a nonstream value.

## Simple

```
long count()

Optional<T> max(
    Comparator<? super T> comparator)

Optional<T> findFirst()

boolean anyMatch(
    Predicate<? super T> predicate)
```

## General 🤯

```
Optional<T> reduce(
    BinaryOperator<T> accumulator)

T reduce(
    T identity,
    BinaryOperator<T> accumulator)

<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

# Reduction Operations

```
jshell> IntStream.of(1,2,3,4).reduce((x, y) → x + y)
$8 ⟹ OptionalInt[10]
```

adds a **partial result** x with the **next value** y to yield a **new partial result**.

```
jshell> IntStream.of(1,2,3,4).reduce(5, (x, y) → x + y)
$9 ⟹ 15
```

Suppose you have a stream of objects and want to form the sum of some property, such as ***lengths in a stream of strings***.

Can you still use these 2 methods?

The `BinaryOperator<T>` is by definition `(T, T) → T` but now we have two types: The stream elements are `String`, and the accumulated result is an integer.

```
int result = 0;
for (String e: stream)
    result = accumulator.apply(result, e);
return result;
```

# Reductions (Terminal Operations)

Reduce the stream into a nonstream value.

## Simple

```
long count()

Optional<T> max(
    Comparator<? super T> comparator)

Optional<T> findFirst()

boolean anyMatch(
    Predicate<? super T> predicate)
```

## General 🤯

```
Optional<T> reduce(
    BinaryOperator<T> accumulator)

T reduce(
    T identity,
    BinaryOperator<T> accumulator)

<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

```
jshell> Stream.of("gently", "down", "the", "stream").
    reduce(0,
           (total, word) -> total + word.length(),
           (total1, total2) -> total1 + total2)
$10 ==> 19
```

`reduce` is not constrained to execute sequentially, i.e. can be applied to parallel streams.

If using parallel streams, the operation in `accumulator` must be associative. Otherwise, the results would be inconsistent.