# CS2030S

## Programming Methodology II

### Recitation 10

# Q1

# Q1

Basic

## Basic

### Slow Computation

```
A incr() {
  // SLOW!
  return new A(this.x + 1);
}

A decr() {
  // SLOW!
  if (x < 0) {
    throw new IllegalStateException();
  }
  return new A(this.x - 1);
}
```

### Parallelise

```
static A foo(A a) {

  return a.incr()
          .decr();

}
```

# Q1



## Q1A

SupplyAsync

Parallelise

```
static A foo(A a) {

  return a.incr()
          .decr();

}
```

# Q1



## Q1A

### SupplyAsync

```
static CompletableFuture<A> foo(A a) {
  return CompletableFuture.supplyAsync(
            () -> a.incr()
                  .decr()
         );
}
```

### Parallelise

```
static A foo(A a) {

  return a.incr()
         .decr();

}
```

**Note**

- `incr()` and `decr()` will be in the same thread

# Q1

Basic
Q1A
- *SupplyAsync*
- *ThenApply*

## Q1A

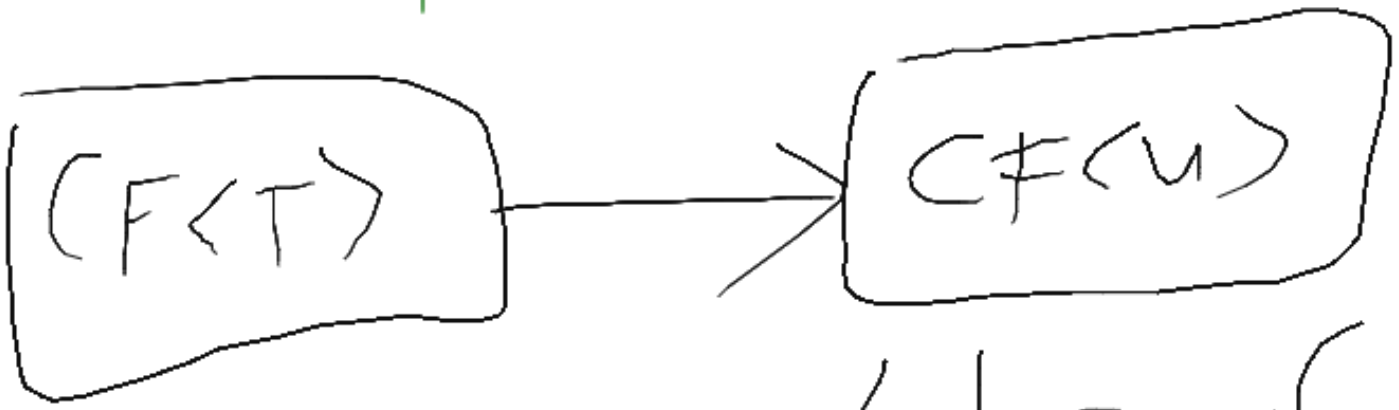
ThenApply

decr(
incr(a)
)

Parallelise

```
static A foo(A a) {

  return a.incr()
         .decr();

}
```

T → U




map / transform

# Q1



## Q1A

### ThenApply

```
static CompletableFuture<A> foo(A a) {
  return CompletableFuture.supplyAsync(
            () -> a.incr()
          ).thenApply(x -> x.decr());

}
```

### Parallelise

```
static A foo(A a) {

  return a.incr()
           .decr();

}
```

**Note**

- `incr()` and `decr()` will be in the same thread

# Q1



## Q1A

ThenApplyAsync

Parallelise

```
static A foo(A a) {

  return a.incr()
          .decr();

}
```

# Q1



## Q1A

### ThenApplyAsync

```
static CompletableFuture<A> foo(A a) {
  return CompletableFuture.supplyAsync(
            () -> a.incr()
          ).ThenApplyAsync(x -> x.decr());

}
```

### Parallelise

```
static A foo(A a) {

  return a.incr()
           .decr();

}
```

**Note**

- `incr()` and `decr()` may be in *different* threads

# Q1



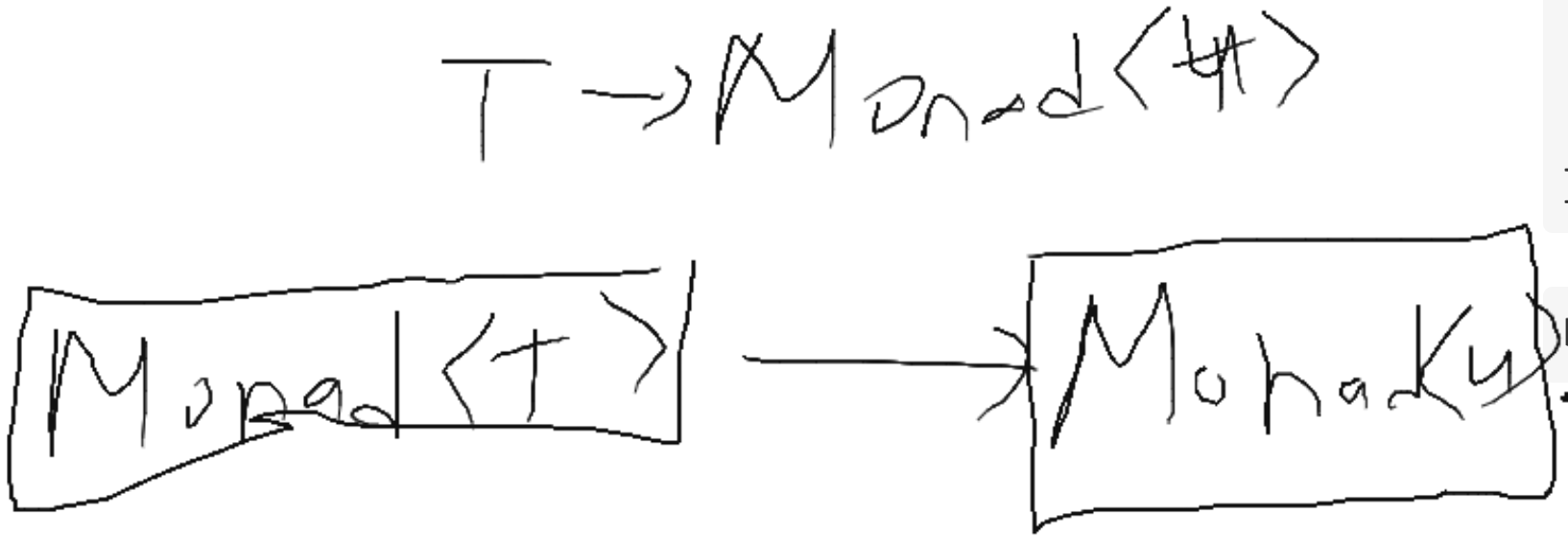
## Q1B

CompletableFuture

Parallelise

```
static A bar(A a) {

  return a.incr();

}
```

```
bar(foo(new A()))
```



# Q1



## Q1B

### CompletableFuture

```
static CompletableFuture<A> bar(A a) {
  return CompletableFuture.supplyAsync(
            () -> a.incr();
          );
}
```

### Parallelise

```
static A bar(A a) {

  return a.incr();

}
```

```
bar(foo(new A()))
```

# Q1



## Q1B

### CompletableFuture

```
static CompletableFuture<A> bar(A a) {
  return CompletableFuture.supplyAsync(
            () -> a.incr();
          );
}
```

```
foo(new A()).thenCompose(x -> bar(x)).join();
```

flatMap

### Parallelise

```
static A bar(A a) {

  return a.incr();

}
```

```
bar(foo(new A()))
```

# Q1



## Q1C

CompletableFuture

Parallelise

```
static A baz(A a,int x) {
  if (x == 0) {

    return new A();

  } else {

    return a.incr()
            .decr();

}
```

# Q1



## Q1C

### CompletableFuture

```
static CompletableFuture<A> baz(A a, int x) {
  if (x == 0) {
    return CompletableFuture.completedFuture(
      new A()
    );
  } else {
    return CompletableFuture.supplyAsync(
            () -> a.incr()
                  .decr()
          );
}
```

### Parallelise

```
static A baz(A a,int x) {
  if (x == 0) {

    return new A();

  } else {

    return a.incr()
            .decr();

}
```

# Q1



## Q1D

### AllOf

Java SE 17 & JDK 17

**Module** java.base
**Package** java.util.concurrent

### Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

**Type Parameters:**

T - The result type returned by this future's `join` and `get` methods

**All Implemented Interfaces:**

CompletionStage<T>, Future<T>

---

```
public class CompletableFuture<T>
extends Object
```

# Q1



## Q1D

### AllOf

```
CompletableFuture<Void> all = CompletableFuture.allOf(
  foo(new A()),
  bar(new A()),
  baz(new A(), 1)
);
all.join();
System.out.println("done!");
```

# Q1



## Q1E

Handle

Java SE 17 & JDK 17

**Module** java.base
**Package** java.util.concurrent

### Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

**Type Parameters:**

T - The result type returned by this future's `join` and `get` methods

**All Implemented Interfaces:**

CompletionStage<T>, Future<T>

---

```
public class CompletableFuture<T>
extends Object
```

# Q1



## Q1E

Handle

```
CompletableFuture<A> exc = CompletableFuture
  .supplyAsync(() -> new A().decr().decr())
  .handle((res,exc) -> {
    if (exc != null) {
      System.out.println("ERROR: " + exc);
      return new A();
    } else {
      return res;
    }
  });
System.out.println(exc.join());
```

# Q2

# Q2

Q2A
*- Dependency*

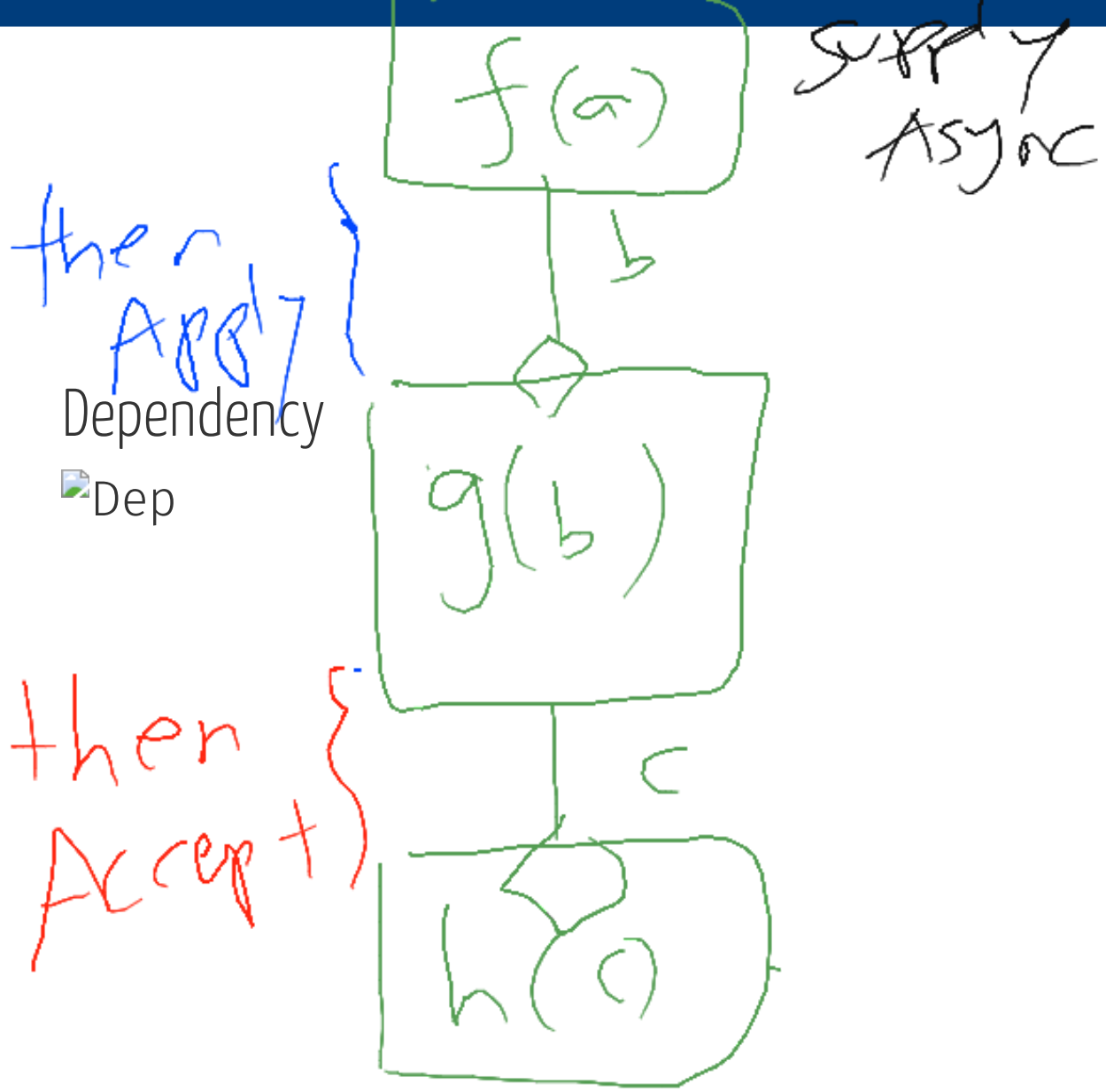
## Q2A

Code

```
B b = f(a);
C c = g(b);

D d = h(c);
```

Dependency

Dep



# Q2

Q2A
- *Dependency*
- *Future*

## Q2A

### Code

```
B b = f(a);
C c = g(b);

D d = h(c);
```

### CompletableFuture

```
CompletableFuture<D> cf = CompletableFuture
  .supplyAsync(() -> f(a))
  .thenApply  (b  -> g(b))
  .thenApply  (c  -> h(c));
D d = cf.join();
```

# Q2



## Q2B

### Code

```
B b = f(a);
C c = g(b);
h(c); // no return
```

### Dependency

Dep

# Q2



## Q2B

### Code

```
B b = f(a);
C c = g(b);
D d = h(c);
```

### CompletableFuture

```
CompletableFuture<Void> cf = CompletableFuture
  .supplyAsync(() -> f(a))
  .thenApply  (b  -> g(b))
  .thenAccept (c  -> h(c));
cf.join();
```

# Q2



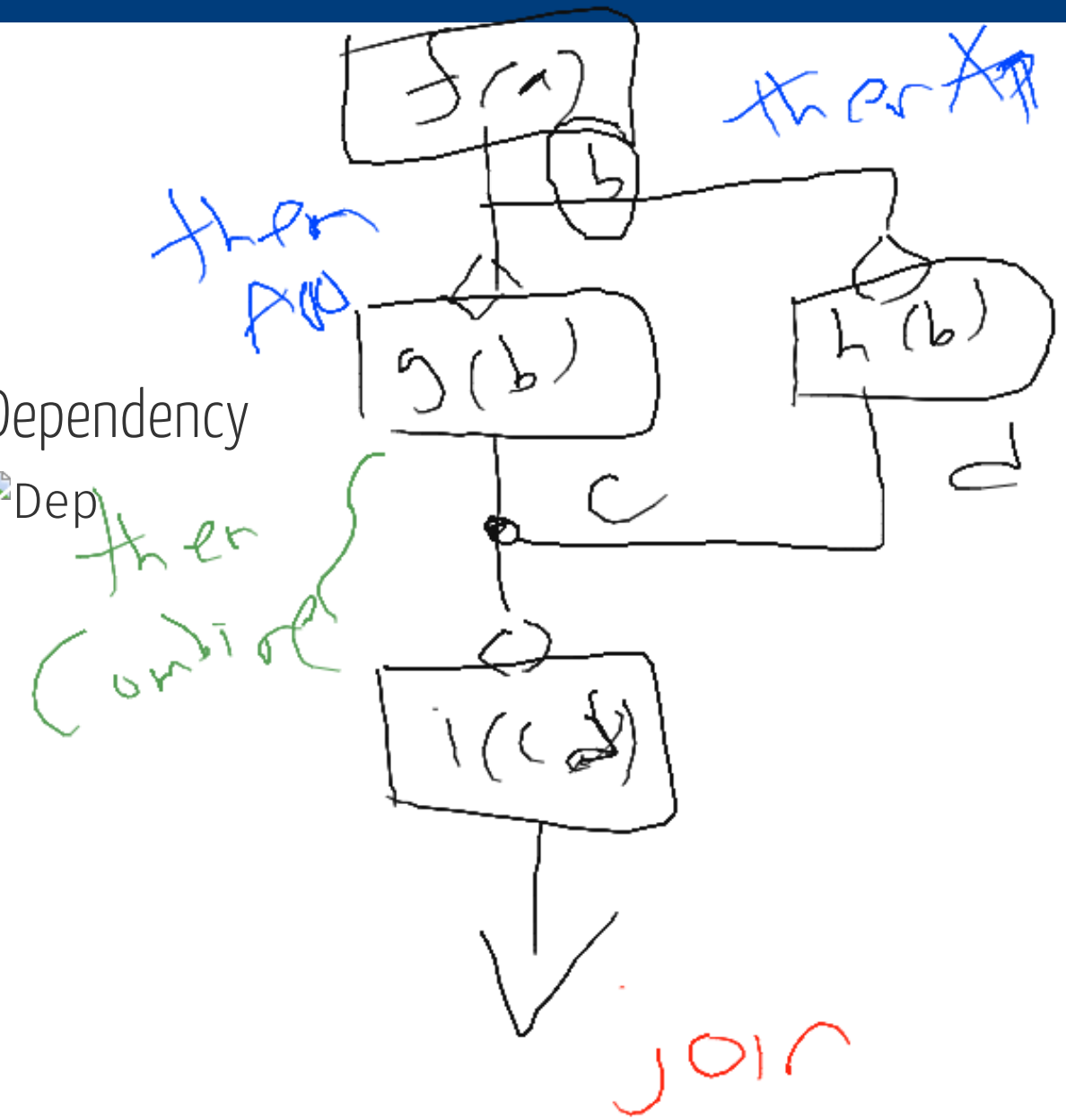
## Q2C

### Code

```
B b = f(a);

C c = g(b);

D d = h(c);


E e = i(c, d);
```

### Dependency

Dep



# Q2



## Q2C

### Code

```
B b = f(a);

C c = g(b);

D d = h(c);


E e = i(c, d);
```

### CompletableFuture

```
CompletableFuture<B> cfb = CompletableFuture
  .supplyAsync(() -> f(a));
CompletableFuture<C> cfc = cfb
  .thenApply(b -> g(b));
CompletableFuture<D> cfd = cfb
  .thenApply(b -> h(b));
CompletableFuture<E> cfe = cfc
  .thenCombine(cfd, (c,d) -> i(c,d));
E e = cfe.join();
```

# Q4

# Q4

Fibonacci

## Fibonacci

```
static int fib(int n) {
  if (n <= 1) {
    return n;
  } else {
    int f1 = fib(n - 1);
    int f2 = fib(n - 2);


    int r2 = f2;
    int r1 = f1;
    return r1 + r2;
  }
}
```

1

2

Invoke:

1: Pass by value

2: jump/fork

# Q4

Fibonacci

class Fib extends RecursiveTask

## Fibonacci

```
static int fib(int n) {
  if (n <= 1) {
    return n;
  } else {
    int f1 = fib(n - 1);
    int f2 = fib(n - 2);



    int r2 = f2;
    int r1 = f1;
    return r1 + r2;
  }
}
```

@Override

```
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);
    f1.fork();
    f2.fork();
    int r2 = f2.join();
    int r1 = f1.join();
    return r1 + r2;
  }
}
```

# Q4

Fibonacci Variants

## Variants

```java
static int fib(int n) {
  if (n <= 1) {
    return n;
  } else {
    int f1 = fib(n - 1);
    int f2 = fib(n - 2);


    int r2 = f2;
    int r1 = f1;
    return r1 + r2;
  }
}
```

=

```java
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);


    int r2 = f2.compute();
    int r1 = f1.compute();
    return r1 + r2;
  }
}
```

# Q4

Fibonacci Variants

## Variants

```java
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);

    f2.fork();
    int r2 = f2.join();
    int r1 = f1.compute();
    return r1 + r2;
  }
}
```

```java
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);
    f1.fork();

    int r2 = f2.compute();
    int r1 = f1.join();
    return r1 + r2;
  }
}
```

# Q4

Fibonacci Variants

## Variants

```
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);
    f1.fork();
    f2.fork();
    int r2 = f2.join();
    int r1 = f1.join();
    return r1 + r2;
  }
}
```

```
protected Integer compute() {
  if (n <= 1) {
    return n;
  } else {
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);
    f1.fork();
    f2.fork();
    int r1 = f1.join();
    int r2 = f2.join();
    return r1 + r2;
  }
}
```

```
jshell> /exit
|    Goodbye
```