# COMP 322: Parallel and Concurrent Programming

## Lecture 7: Futures

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

# Lazy memo implementation (simplified)

```
public class Lazy<T> {
  private T contents;
  private Supplier<T> supplier;

  private Lazy(Supplier<T> supplier) {
    contents = null;
    this.supplier = supplier;
  }

  public T get() {
    if (contents != null) {
       return contents;
    }

    if (supplier != null) {
      contents = supplier.get();
      supplier = null;
    }

    return contents;
  }
```

Private constructor (as usual) plus a factory method (Lazy.*of*)

If we've already computed the answer, return it.

Call the lambda (once), get the result, forget the lambda.

# From laziness to parallelism

```
public class Lazy<T> {
 private T contents;
 private Supplier<T> supplier;

 private Lazy(Supplier<T> supplier) {
  contents = null;
  this.supplier = supplier;
 }

 public T get() {
  if (contents != null) {
    return contents;
  }

  if (supplier != null) {
   contents = supplier.get();
   supplier = null;
  }

  return contents;
 }
```

Why lazy?

- Maybe it's expensive to compute the *contents*
- Maybe you won't actually need it

The idea: defer computation of a value until (and if) you need

What if?

- I ***know*** that I ***will*** need the *contents* eventually
- But ***not right now***
- My framework knows how to execute the *supplier* in parallel on spare resources, so I can go do something else in the meantime

The idea: offload the computation of the *contents* by the *supplier* to spare resources (another core), get the (hopefully already computed) value when you need it

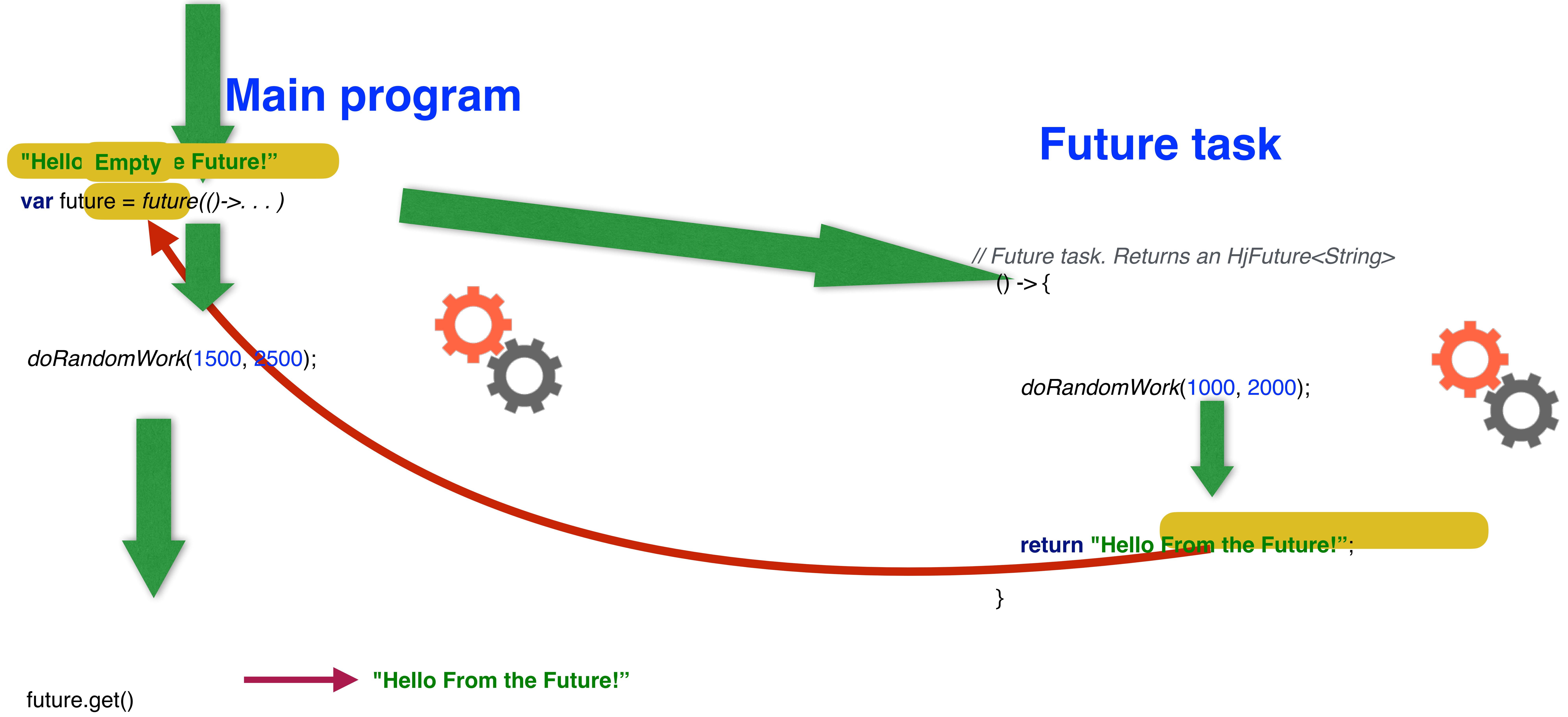
# Futures

```
launchHabaneroApp(() -> {
    var future = future(() -> { // Future task. Returns an HjFuture<String>
        doRandomWork(1000, 2000);
        System.out.println("Done with the future task");
        return "Hello From the Future!”;
    });
    doRandomWork(1500, 2500);
    System.out.println("Done with the main task");
    System.out.println("The future task returned the value " + future.get());
});
```

```
Done with the main task
Done with the future task
The future task returned the value Hello From the Future!

Done with the future task
Done with the main task
The future task returned the value Hello From the Future!
```

# Futures



```
var future = future(()->. . . )

doRandomWork(1500, 2500);

future.get()
```

```
// Future task. Returns an HjFuture<String>
() -> {

    doRandomWork(1000, 2000);

    return "Hello From the Future!";

}
```

# Futures

## Main program

"Hello Empty e Future!"

```
var future = future(()->. . . )
```

```
doRandomWork(1500, 2500);
```

```
future.get()
```

he Future!"

## Future task

```
// Future task. Returns an HjFuture<String>
() -> {

    doRandomWork(1000, 2000);

    return "Hello From the Future!";

}
```

# What is a Future?

Read-only container (just like a Lazy Memo)

Always Empty on creation (just like a Lazy Memo)

Creation completes immediately (just like a Lazy Memo)

Gets computed and filled-in by a lambda (just like a Lazy Memo)

The user calls get() on it to get the value (just like a Lazy Memo)

The lambda to compute the value may be executed in parallel

- Unlike the Lazy Memo

The user may block on the first get() if the value is not ready

- ***Kind of*** like the Lazy Memo. The first get() on the Lazy Memo always (kind of) blocks: it has to wait for the lambda to execute.

Subsequent get() calls complete immediately (just like a Lazy Memo)

## Futures

Simple, but an extremely powerful concept in parallel and concurrent programming

Very functional in nature

Almost all languages and frameworks that have any kind of parallel and concurrent programming have them

JavaScript (Futures), Kotlin (asyncs), Python (futures in Ray on Python), C++ (std::future), Go (closures and goroutines), Scala (futures), C# (Task<T>), Clojure (future), Ruby (Concurrent::Future), R (future package), Swift (Future)

## Example: Two-way parallel sum using Futures

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) & sum2 (upper half) in parallel
var sum1 = future(() -> { // Future Task T2
    int sum = 0;
    for (int i = 0; i < X.length / 2; i++) sum += X[i];
    return sum;
});
var sum2 = future(() -> { // Future Task T3
    int sum = 0;
    for (int i = X.length / 2; i < X.length; i++) sum += X[i];
    return sum;
});
// Task T1 waits for Tasks T2 and T3 to complete
int total = sum1.get() + sum2.get();
```

# Another way of computing Parallel Sum

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) & sum2 (upper half) in parallel

// Task T2 (the future task) computes the lower half sum
var sum1 = future(() -> { // Future Task T2
    int sum = 0;
    for (int i = 0; i < X.length / 2; i++) sum += X[i];
    return sum;
});

// Task T1 (the main program) computes the upper half sum
int sum2 = 0;
for (int i = X.length / 2; i < X.length; i++) sum2 += X[i];

// Task T1 waits for Task T2 to complete
int total = sum1.get() + sum2;
```

## Does this work?

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) & sum2 (upper half) in parallel

// Task T1 (the main program) computes the upper half sum
int sum2 = 0;
for (int i = X.length / 2; i < X.length; i++) sum2 += X[I];

// Task T2 (the future task) computes the lower half sum
var sum1 = future(() -> { // Future Task T2
   int sum = 0;
   for (int i = 0; i < X.length / 2; i++) sum += X[i];
   return sum;
});

// Task T1 waits for Task T2 to complete
int total = sum1.get() + sum2;
```

# Recursive Array Sum (Sequential version)

Sequential divide-and-conquer pattern:

```
static int computeSum(int[] X, int lo, int hi) {
    if ( lo > hi ) return 0;
    else if ( lo == hi ) return X[lo];
    else {
        int mid = (lo+hi)/2;
        int sum1 =
                computeSum(X, lo, mid);
        int sum2 =
                computeSum(X, mid+1, hi);

       return sum1 + sum2;
    }
} // computeSum

. . .

int sum = computeSum(X, 0, X.length-1); // main
```

## Recursive Array Sum (Future version)

Parallel divide-and-conquer pattern:

```
static int computeSum(int[] X, int lo, int hi) throws SuspendableException {
    if ( lo > hi ) return 0;
    else if ( lo == hi ) return X[lo];
    else {
        int mid = (lo+hi)/2;
        var sum1 = future(() ->
                computeSum(X, lo, mid));
        var sum2 = future(() ->
                computeSum(X, mid+1, hi));
        // Parent now waits for the future values
        return sum1.get() + sum2.get();
    }
} // computeSum

. . .

int sum = computeSum(X, 0, X.length-1); // main
```

## Summary

Futures are a highly functional, structured and disciplined way to express and coordinate concurrent execution

Simple, but powerful concept

Widespread in popular languages and frameworks