NWEN303 Concurrent Programming

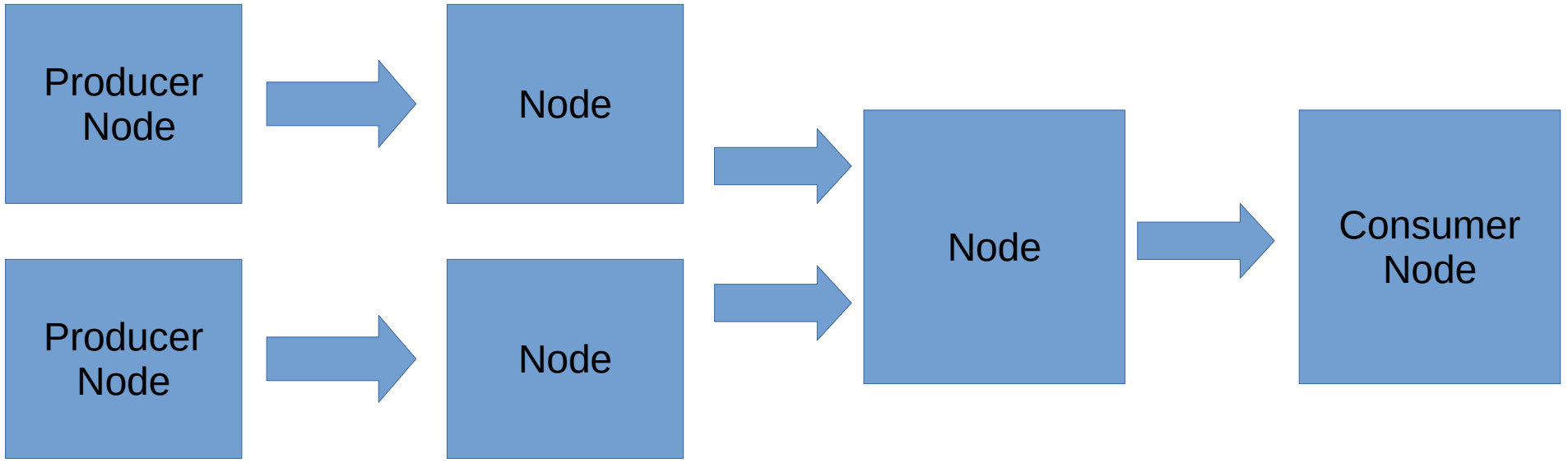
11: Common parallel patterns: Producer Consumer

Marco Servetto
VUW

Producer consumer

- Producer consumer:
  - pipeline instead of fork-join
  - factory metaphor, where objects are produced in steps, and each step can be made in parallel with the others, and specialized workers can act better and faster on a specialized part of the manufacturing.
  - Good for instruction caching
  - Good when manufacturing requires access to limited resources (for example, reading a file)

Producer consumer



Producer/Consumer

- Alice produce wheat,
- Bob produce sugar,
- Charles takes wheat and sugar and produce cakes.
- Finally the little Tim eat the cakes.
- When enough cakes are eaten, we get a `Gift'.
- For now, we assume those tasks need to be done sequentially; for example the oven can only bake 1 cake at a time.
- Alice, Bob, Charles and Tim will communicate list of products .
- Alice and Charles shares list1, Bob and Charles shares list2 and finally Charles and Tim shares list3.
- Lets try to get it right with some code!

Cakes! (attempt 1)

```
public static Gift computeGift(int hunger){
  List<Wheat> ws=new ArrayList<>();
  List<Sugar> ss=new ArrayList<>();
  List<Cake>  cs=new ArrayList<>();
  int[] timHunger=new int[]{hunger};
  Gift[] res=new Gift[]{null};

  Thread alice=new Thread(()->{while(true){ws.add(wheat());}});

  Thread bob=new Thread(()->{while(true){ ss.add(sugar());}});

  Thread charles=new Thread(()->{while(true){
    Sugar s=ss.remove(0);
    Wheat w=ws.remove(0);
    cs.add(cake(s,w));}});

  Thread tim=new Thread(()->{while(true){
    cs.remove(0);
    if(timHunger[0]>0){timHunger[0]-=1; res[0]=gift();}});

  alice.start();bob.start();charles.start();tim.start();

  return res[0];
  }
```

Cakes! (attempt 1-failed)

```
public static Gift computeGift(int hunger){
  List<Wheat> ws=new ArrayList<>();
  List<Sugar> ss=new ArrayList<>();
  List<Cake>  cs=new ArrayList<>();
  int[] timHunger=new int[]{hunger};
  Gift[] res=new Gift[]{null};

  Thread alice=new Thread(()->{while(true){ws.add(wheat());}});

  Thread bob=new Thread(()->{while(true){ ss.add(sugar());}});

  Thread charles=new Thread(()->{while(true){
    Sugar s=ss.remove(0);//what if there is no sugar/wheat to remove?
    Wheat w=ws.remove(0);//even if there is the sugar/wheat,
    cs.add(cake(s,w));}});//we have a race condition on ss and ws

  Thread tim=new Thread(()->{while(true){
    cs.remove(0);//same as above
    if(timHunger[0]>0){timHunger[0]-=1; res[0]=gift();}});

  alice.start();bob.start();charles.start();tim.start();
  //the threads are never stopped
  return res[0];//we are not waiting for the result to be there
  }//even if we somehow wait, thanks to caching, we may not see it!
```

Cakes! (attempt 2-failed)

```
List<Wheat> ws=Collections.synchronizedList(new ArrayList<>());
List<Sugar> ss=Collections.synchronizedList(new ArrayList<>());
List<Cake>  cs=Collections.synchronizedList(new ArrayList<>());
//now I can access ws ss and cs in parallel!
Thread alice=new Thread(()->{while(true){ws.add(produceWheat());}});

Thread bob=new Thread(()->{while(true){ ss.add(produceSugar());}});

Thread charles=new Thread(()->{while(true){
  Sugar s=ss.remove(0);//still... what if the list is empty?
  Wheat w=ws.remove(0);
  cs.add(produceCake(s,w));}});
```

How to take the sugar?

```
if (ss.isEmpty()){Thread.sleep(100);}
Sugar s=ss.remove(0);
```

- Wrong! may be is still empty

```
while (ss.isEmpty()){Thread.sleep(100);}
Sugar s=ss.remove(0);
```

- Does it works? yes, but only in the current example, since there is only one thread taking the sugar...
- Would be better to use wait/notify and have a the sugar producer awaking the sugar user!

How to take the sugar?

- When we take the sugar

```
synchronized(ss){
  while(ss.isEmpty()){ss.wait();}
  Sugar s=ss.remove(0);}
```

- When we put the sugar

```
synchronized(ss){
  ss.add(produceSugar());
  ss.notifyAll();}
```

- Correct but unsatisfactory. This defies the purpose of synchronized collections, requires all the producer/consumer to properly follow the pattern and calls notify all much more often that what would be actually needed.
- It seams so much of a reasonably obvious and common thing to do....

It seams so much of a reasonably obvious and common thing to do....

- So, someone have done it already and put it into a decent library. It is so common indeed that that library is even in the standard Java.
- Search for BlockingQueue!
- The main lesson here is that before going head on searching to forge a solution, we should spend at least 4-10 hours searching for a good library that is doing already what we need.

Cakes!

```
BlockingQueue<Wheat> ws=new LinkedBlockingQueue<>();
BlockingQueue<Sugar> ss=new LinkedBlockingQueue<>();
BlockingQueue<Cake> cs=new LinkedBlockingQueue<>();

Thread alice=new Thread(()->{while(true){    // add may work, but...
  try{ws.put(wheat());}catch(InterruptedException e){return;}});
  //thread dies: not "swallowed"

Thread bob=new Thread(()->{while(true){
  try{ss.put(sugar());}catch(InterruptedException e){return;}});

Thread charles=new Thread(()->{while(true){
  try{cs.put(cake(ss.take(),ws.take()));}catch(..){return;}});
```

Such a nice library!

Reading!

docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/BlockingQueue.html

- In Particular, try to understand what is the best scenario to use the methods in the first table:

Insert:	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove:	remove()	poll()	take()	poll(time, unit)
Examine:	element()	peek()	n/a	n/a

Cakes, using CompletableFuture

```
public static Gift computeGift(int hunger){
  BlockingQueue<Wheat> ws=new LinkedBlockingQueue<>();
  BlockingQueue<Sugar> ss=new LinkedBlockingQueue<>();
  BlockingQueue<Cake> cs=new LinkedBlockingQueue<>();
  int[] timHunger=new int[]{hunger};
  CompletableFuture<Gift> res=new CompletableFuture<>();//wait for result

  Thread alice=new Thread(()->{while(true){    //add may work, but...
    try{ws.put(wheat());}catch(InterruptedException e){return;}});
    //thread dies: not "swallowed"

  Thread bob=new Thread(()->{while(true){
    try{ss.put(sugar());}catch(InterruptedException e){return;}});

  Thread charles=new Thread(()->{while(true){
    try{cs.put(cake(ss.take(),ws.take()));}catch(..){return;}});

  Thread tim=new Thread(()->{while(true){cs.take(); //setting result
    try{if(timHunger[0]>0){timHunger[0]-=1;return;}res.complete(gift());}
    catch(..){return;}   }});
  alice.start();bob.start();charles.start();tim.start();
  try{return res.join();}//waiting for result; join() is like get() but..
  finally{alice.interrupt();..bob...charles..;tim.interrupt();}
  }//remember to interrupt your threads
```

Producer/Consumer

- This is a very coarse grained parallelism, every problem would lent to a specific graph of producer/consumers, that have nothing to to with the specific resource of the underling machine.
- Not too bad, we still can use future/fork join inside some of the nodes.
- The example before assumes that would be impossible/wrong/hard to produce more then one computation for any kind at the same time.
- That is, if you can produce/eat more then one cake at the same time, probably just parallelise the process as a whole.

The proposed “correct code” is horrible

- Can we do something better?
- while/try-catch logic is repeated between nodes
- We create threads manually - slow if this is just a part of a bigger program
- Can we make an abstraction to run actions in parallel?

Ideal code:

```
public static Gift computeGift(int hunger) throws InterruptedException{
  BlockingQueue<Wheat> ws=new LinkedBlockingQueue<>(10);
  BlockingQueue<Sugar> ss=new LinkedBlockingQueue<>(10);
  BlockingQueue<Cake>  cs=new LinkedBlockingQueue<>(10);
  AtomicInteger timHunger=new AtomicInteger(hunger);

  ConcurrentRunner<Gift> runner=new ConcurrentRunner<>();

  runner.run(10,()->ws.put(wheat()));//alice, duplicated 10 times

  runner.run(10,()->ss.put(sugar()));//bob, duplicated 10 times

  runner.run(1,()->cs.put(cake(ss.take(),ws.take())));//charles, 1 time

  runner.run(5,()->{//tim, somehow he can eat 5 cakes at the same time
    cs.take();
    if(timHunger.decrementAndGet()>0){return;}
    runner.setResult(gift());
    });

  return runner.result();
  }
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks = new ArrayList<>();
  private CompletableFuture<T> end = new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r = ()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){ tasks.add(exe.submit(r));} }

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks = new ArrayList<>();
  private CompletableFuture<T> end = new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r = ()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){tasks.add(exe.submit(r));}}

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks = new ArrayList<>();
  private CompletableFuture<T> end = new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r = ()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){tasks.add(exe.submit(r));}}

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks=new ArrayList<>();
  private CompletableFuture<T> end=new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r=()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){tasks.add(exe.submit(r));}}

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks = new ArrayList<>();
  private CompletableFuture<T> end = new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r = ()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){tasks.add(exe.submit(r));}}

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

ConcurrentRunner<T>

```
public interface Action{ void run() throws InterruptedException; }

public class ConcurrentRunner<T>{//By Marco. Learn to make abstractions!

  private static ExecutorService exe=Executors.newCachedThreadPool(r->{
    Thread t = Executors.defaultThreadFactory().newThread(r);
    t.setDaemon(true); return t;});//deamons allows the JVM to terminate

  private List<Future<?>> tasks = new ArrayList<>();
  private CompletableFuture<T> end = new CompletableFuture<T>();

  public void run(int repeats,Action a) { //to add a task/action
    Runnable r = ()->{ while(runOnce(a)){} };
    for(int i=0;i<repeats;i+=1){tasks.add(exe.submit(r));}}

  private boolean runOnce(Action a) {
    try{ a.run(); }
    catch(InterruptedException|RuntimeException|Error e){
      end.completeExceptionally(e);
      for(Future<?> fi:tasks){fi.cancel(true);}}
    return !end.isDone() && !Thread.interrupted();}//if false we stop

  public void setResult(T res){//result is available!
    end.complete(res);for(Future<?> fi:tasks){fi.cancel(true);}}

  public T result(){ return end.join(); } } //wait for the result
```

What is the takeaway?

- Start with a **prototype** using available features
- Make it work:
  - in the good case
  - when it is called multiple times in a functional sense
  - when failure is involved
- Then, look at the result and try to abstract your process.
- Finally, make an abstraction supporting your code, and **rewrite** your code using your abstraction.
- The last step saves you some maintenance hell.