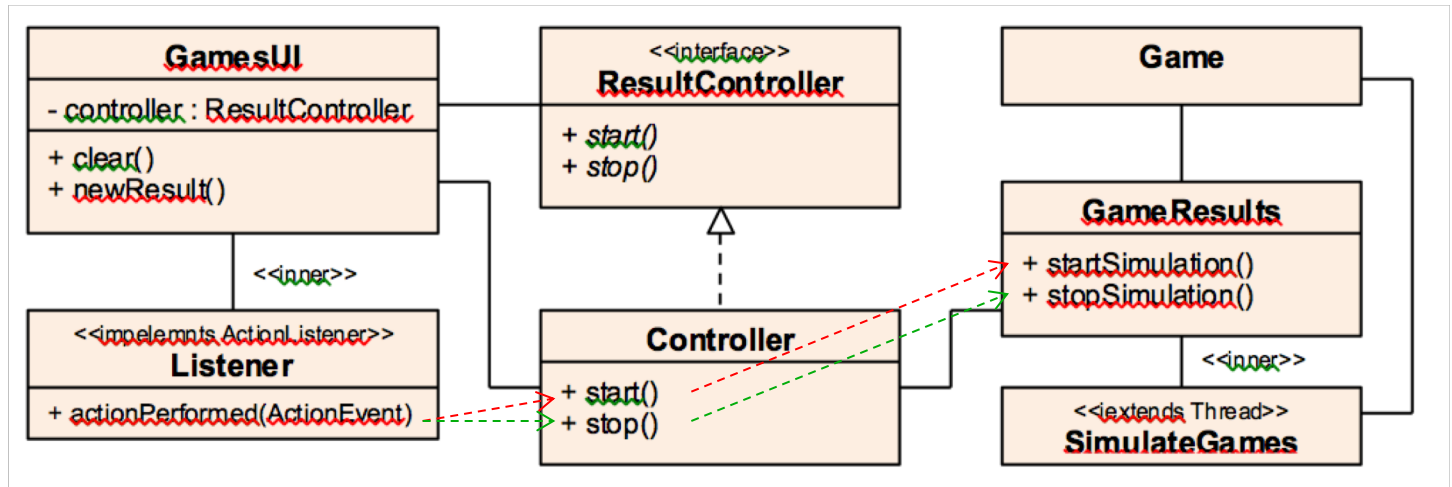


# Föreläsning 6

- Observer / Callback- avrundas
- Synkronisering
- Buffert
- Tråd med buffert

JNP: s 70-92

# Laboration 5 – avrundning, repetition



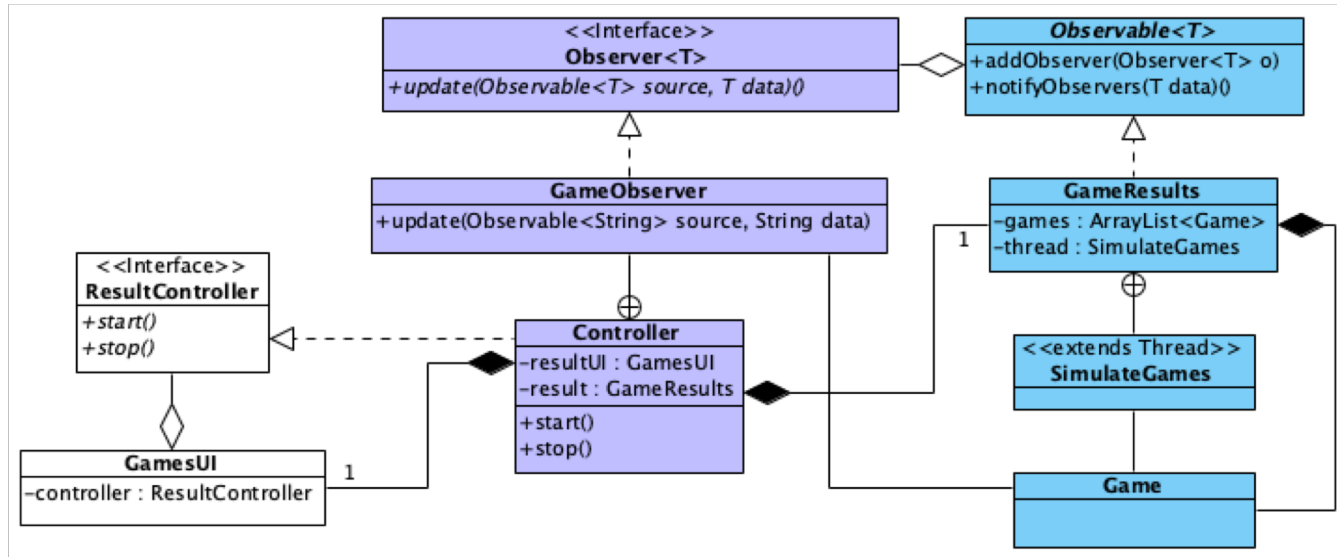
Då användaren klickar på "Start" så anropas `Controller.start()` som i sin tur anropar `GameResults.startSimulation()`.

Då användaren klickar på "Stop" så anropas `Controller.stop()` som i sin tur anropar `GameResults.stopSimulation()`.

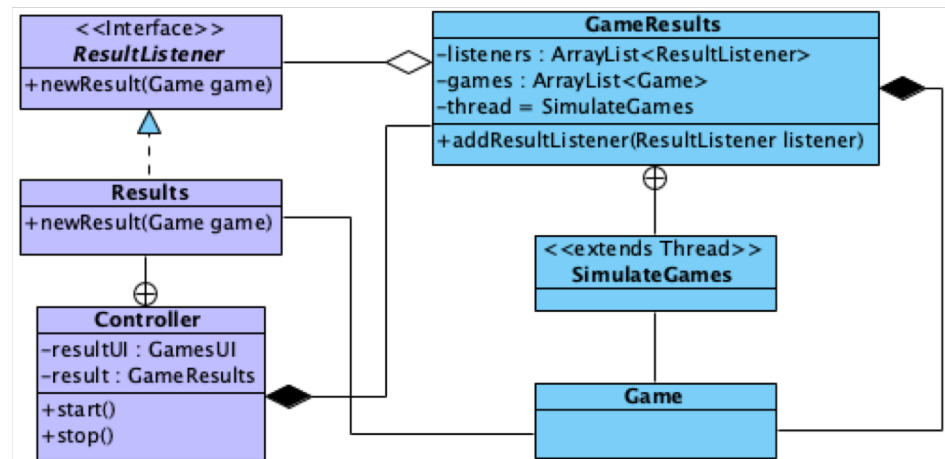
Uppgift: Se till att `GameResult`-objektet rapporterar resultat genom Observer-mönstret till registrerade Observers. Sedan ska resultaten visas i `GameUI`.

# Laboration 5 – avrundning (repetition)

## 1 Använda Observable + Observer



## 2 Använda Callback



# Synkronisera kodsekvens

Om flera trådar använder samma resurs, t.ex. en array eller ett LinkedList-objekt så kan det bli problem. "Samtidigt" som en tråd ändrar i strukturen så läser en annan. Och det kan ge oväntade fel eller t.o.m. en programkrash som resultat.

För att säkerställa att flera trådar kan använda samma kod utan felaktiga resultat kan man *synkronisera sekvenser av kod*. Man skapar *trådsäker kod*.

Man kan synkronisera all kod i en metoder:

```
public void synchronized setText( String txt)
    // skyddad kod
}
public void synchronized appendText( String txt ) {
    // skyddad kod
}
```

Endast en av ovanstående metoder kan exekveras åt gången, och endast av en tråd.

Counter1

Counter2

Counter1Thread

Counter2Thread

# Synkronisera kodsekvens

Det går också bra att synkronisera en kodsekvens (block) inuti en metod med *synchronized*-sats:

```
public void method() {  
    :  
    synchronized( objektref ) {  
        // skyddad kod  
    }  
    :  
}
```

*synchronized*-satsen har en referens, objektref, till ett objekts som argument. Om objektet, som objektref refererar till, innehåller *synchronized*-metoder så kan endast en av metoderna alternativt *synchronized*-satsen ovan exekveras åt gången.

Counter1

Counter2

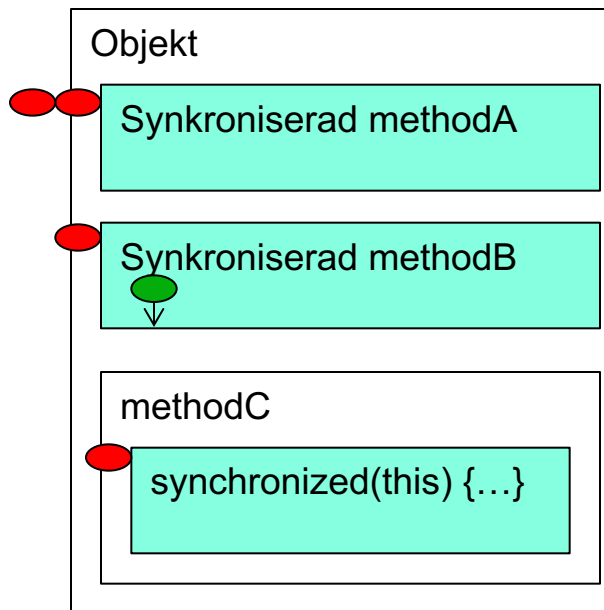
Counter1Thread

Counter2Thread



# Alla objekt har ett lås

Alla *objekt* i java har ett *lås*.

När en tråd börjar exekvera en synchronized-metod (methodA+B) så blir alla synkroniserade miljöer i objektet låsta för andra trådar. De får vänta på tillträde. När tråden lämnar den synkroniserade metoden/blocket så får en ny tråd tillträde.



När en tråd börjar exekvera en synchronized-sats (methodC) så kommer på samma sätt alla synkroniserade miljöer, i objektet som används som argument, att bli låsta för andra trådar.

-  Tråd som väntar på tillträde till den låsta kodsekvensen.
-  Tråd som exekverar synkroniserad kod. All synkroniserad kod är låst för andra trådar.

När den gröna tråden lämnar methodB kommer en av de väntande röda trådarna få tillträde till methodA / methodB / synkroniserade blocket. Vilken av trådarna som får tillträde är ej specificerat.

# Collection och synkronisering

Klasser som implementerar Collection / Map är i allmänhet ej synkroniserade. För att synkronisera en sådan klass kan man:

1. låta en instans av klassen vara attribut i en ny klass
2. skriva synkroniserade metoder vilka använder instansen.

```
public class SynchronizedHashMap<K,V> {  
    private HashMap<K,V> map = new HashMap<K,V>();  
  
    public synchronized V put(K key, V value) {  
        return map.put(key, value);  
    }  
  
    public synchronized V remove(K key) {  
        return map.remove(key);  
    }  
  
    public synchronized void clear() {  
        map.clear();  
    }  
  
    public synchronized V get(K key) {  
        return map.get(key);  
    }  
}
```

# wait, notify, notifyAll

I klassen Object deklareras nedanstående metoder. Metoderna kan anropas i synkroniserade metoder och i synkroniserade block.

- **public void wait(), public void wait(long ms)**  
Tråden, vilken anropar *wait*, avbryts (*wait()*) / avbryts en viss tid (*wait(long)*).  
Låset lämnas tillbaka och kan användas av annan tråd.  
Den avbrutna tråden kan göras körklar genom att en annan tråd anropar *notify()* eller *notifyAll()*.
- **public void notify()**  
En tråd som avbrutit sin exekvering i objektet görs körklar och sätts i kö för att erhålla objektets lås.
- **public void notifyAll()**  
Alla trådar som avbrutit sin exekvering i objektet körs körklara och de sätts i kö för att erhålla objektets lås.



# En synkroniserad kö – en buffert

En kö är en datastruktur som är som en lista men man kan bara lägga in objekt i slutet och ta ut objekten i början. En kö kan också kallas en buffert.



- Vi skapar en buffert där **Producer-objekt** kan lägga in objekt och **Consumer-objekt** hämta objekt.
- För att inga fel ska uppstå är metoderna **put(Object o)** och **get()** synkroniserade.
- Om bufferten är tom när en **Consumer** ska hämta ett objekt anropas **wait()** och tråden vilar. Metoden **wait** implementeras i klassen **Object** och finns därmed i alla klasser.
- När en **Producer** har lagt in ett objekt så meddelas detta en väntande **Consumer** med **notify()**. Metoden **notify** implementeras i klassen **Object**.
- Om alla väntande **Consumers** ska meddelas så används **notifyAll()**. Metoden **notifyAll** implementeras i klassen **Object**.

# Buffer<T>

```
public class Buffer<T> {  
    private LinkedList<T> buffer = new LinkedList<T>();  
  
    public synchronized void put(T o) {  
        buffer.addLast(o);  
        notifyAll();  
    }  
  
    public synchronized T get() throws InterruptedException {  
        while(buffer.isEmpty()) {  
            wait();  
        }  
        return buffer.removeFirst();  
    }  
  
    public int size() {  
        return buffer.size();  
    }  
}
```

Buffer.java

# Producer

```
public class Producer extends Thread {  
    private String message;  
    private long delay;  
    private Buffer buffer;  
  
    public Producer(String message, long seconds, Buffer buffer) {  
        this.message = message;  
        this.delay = seconds*1000;  
        this.buffer = buffer;  
    }  
  
    public void run() {  
        while(!Thread.interrupted()) {  
            try {  
                Thread.sleep(delay);  
                buffer.put(message);  
            }  
            catch(InterruptedException e) {  
                break;  
            }  
        }  
    }  
}
```

Producer.java

# Consumer

```
public class Consumer implements Runnable {
    private Thread thread;
    private Buffer buffer;
    private long delay;

    public Consumer(long seconds, Buffer buffer) {
        this.delay = seconds*1000;
        this.buffer = buffer;
    }

    public void start() {
        if(thread==null) {
            thread = new Thread(this);
            thread.start();
        }
    }

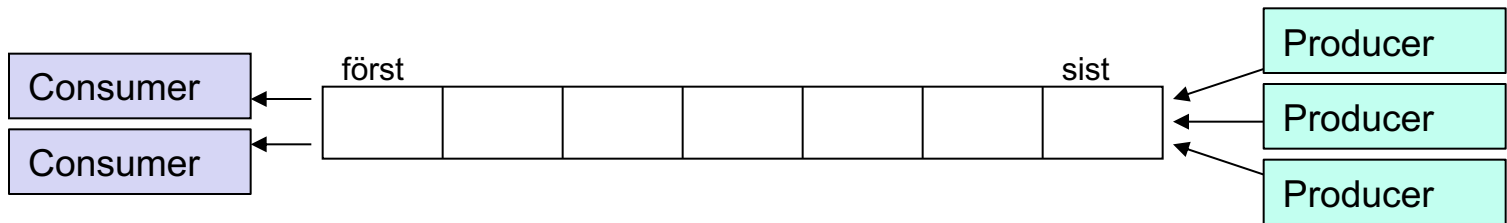
    public void stop() {
        if(thread!=null) {
            thread.interrupt();
        }
    }

    public void run() {
        String txt;
        while(!Thread.interrupted()) {
            try {
                Thread.sleep(delay);
                txt = (String)buffer.get();
                System.out.println(txt);
            } catch (InterruptedException e) {
                break;
            }
        }
        thread=null;
    }
}
```

Consumer.java

# Test av Buffer

- Vi kopplar tre **Producer**-objekt som skapar ord och lägger orden i bufferten. **Producer**-objekten lägger in orden Ole, Dole respektive Doff med olika tidsmellanrum. (Tre trådar)
- Vi kopplar två **Consumer**-objekt till kön som plockar ut orden och skriver ut dem. (Två trådar)



## Körresultat:

Ole

Dole

Doff

Ole

Dole

Ole

...

# TestBuffer

```
public class TestBuffer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer[] prod = {new Producer("Ole", 5, buffer), new Producer("Dole", 7, buffer),
                           new Producer("Doff", 9, buffer)};
        Consumer[] cons = {new Consumer(9, buffer), new Consumer(4, buffer)};

        for(int i=0; i<prod.length; i++) {
            prod[i].start();
        }
        for(int i=0; i<cons.length; i++) {
            cons[i].start();
        }

        try {
            Thread.sleep(60000);
        }
        catch(InterruptedException e) {}

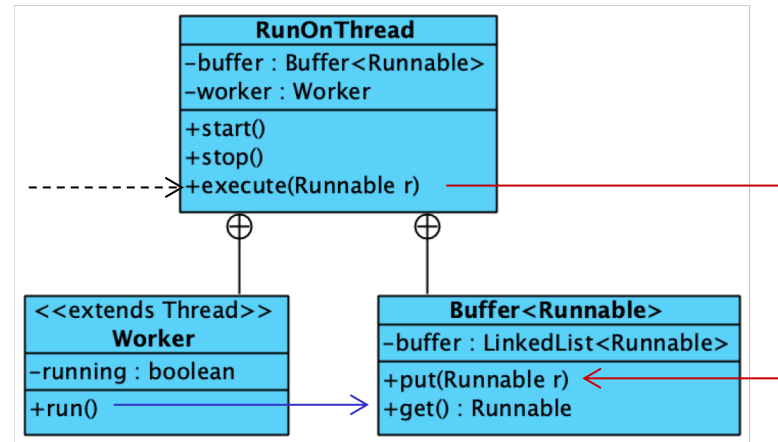
        for(int i=0; i<prod.length; i++) {
            prod[i].interrupt();
        }
        for(int i=0; i<cons.length; i++) {
            cons[i].stop();
        }
        System.out.println("Antal kvar i kön: " + buffer.size());
    }
}
```

TestBuffer.java

# RunOnThread

Det går utmärkt att koppla en buffert till en (eller flera) trådar. Klassen *RunOnThread* använder en tråd för att exekvera Runnable-objekt placerade i bufferten.

Strukturen är följande:



*Worker*-tråden startas och anropar *get*-metoden i bufferten. Det finns inga objekt i bufferten och tråden avbryts.

En extern tråd anropar *execute*-metoden. *Runnable*-objektet läggs i bufferten (*put*).

Om *Worker*-tråden ligger avbruten så görs den körklar och ett *Runnable*-objekt returneras av *get*-metoden.

*run*-metoden anropas därefter av *Worker*-tråden. När *run*-metoden avslutas så anropas på nytt *get*-metoden i bufferten.

RunOnThread

DemoRunOnThread