

# Thread and runnable interfaces

Object Oriented Programming  
2024 First Semester  
Shin-chi Tadaki (Saga University)

1 Threads

2 Synchronization

# Today's theam

- Thread and runnable interfaces
- Synchronization between threads
- Protection by "synchronized" keyword

# Sample program download

`https://github.com/oop-mc-saga/Thread`

# Threads

- Threads are a collection of independent, concurrently running sub-processes within an application.
- Threads can share data and variables among themselves.
  - communication and collaboration between threads
- Threads in java applications
  - Threads play a pivotal role in the concurrent execution of tasks.
  - GUI class instances are specifically designed to run on threads
  - Any class instances in Java can be executed on threads

# Runnable interface

- Classes with the Runnable interface can be executed on threads
- Runnable interface has **only one method** `run()`, called only once from other threads
- Controlling variables for `run()` should be *volatile*
  - *Volatile* variables can be updated immediately

# Major Methods of Thread class

- `start()`
  - Executes `run()` method of a specified instance
- `sleep()`
  - Sleeps the thread during the specified time (millisecond)
- `stop()` method is obsolete and should not be used.

# Two ways for defining a class runnable on thread

- By implementing the Runnable interface
- By defining an anonymous class extending Runnable.
- Both ways need to implement run() method



# Example of Runnable implementation

- `SampleWithThread`
  - Starting the instance inside an anonymous implementation of the `Runnable` interface
- `SampleRunnable`
  - Implementing the `Runnable` interface

See `Thread.example0`

# Sample class

```
1 public class Sample {
2
3     protected volatile boolean running = true;
4     protected int c = 0;
5     private final int id;
6
7     public Sample(int id) {this.id = id;}
8
9     public void update() {
10         Date date = new Date();
11         System.out.println(id + ":" + c + " "
12             + date.toString());
13         c++;
14         if (c > 10) {//Stop after 10 updates
15             running = false;
16         }
17     }
18
19     public boolean isRunning() {return running;}
20 }
```

# SampleWithThread class

```
1 public static void main(String[] args) {  
2     Thread thread0 = new Thread(new Runnable() {  
3         Sample s = new Sample(1);  
4  
5         @Override  
6         public void run() {  
7             while (s.isRunning()) {  
8                 s.update();  
9                 try {  
10                    Thread.sleep(1000);  
11                } catch (InterruptedException e) {}  
12            }  
13        }  
14    });  
15    thread0.start();  
16 }
```

This example defines an anonymous instance of Runnable class. Inside the definition, an instance of Sample class is created and run() method is implemented.

# SampleRunnable class

```
1 public class SampleRunnable extends Sample implements Runnable {
2
3     public SampleRunnable(int id) {
4         super(id);
5     }
6
7     /**
8      * update() at random timing
9      */
10    @Override
11    public void run() {
12        while (running) {
13            update();
14            int t = (int) (1000 * Math.random());
15            try {
16                Thread.sleep(t);
17            } catch (InterruptedException e) {}
18        }
19    }
```

# Running three threads

```
1  /**
2   * @param args the command line arguments
3   */
4  public static void main(String[] args) {
5      new Thread(new SampleRunnable(1)).start();
6      new Thread(new SampleRunnable(2)).start();
7      Thread t = new Thread(new SampleRunnable(3));
8      t.start();
9  }
10
11 }
```

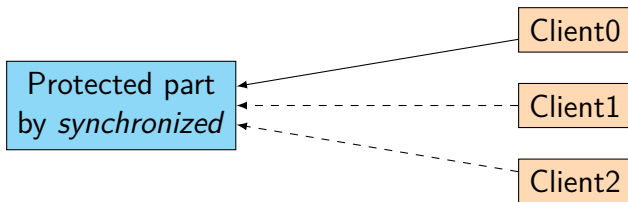
## Asynchronously updates: 非同期更新

- In concurrent programming, threads are granted the ability to perform asynchronous updates on shared data within an application.
- These asynchronous updates can induce data inconsistencies in shared data structures like containers.
- To maintain data integrity, applications must implement synchronization mechanisms for shared data updates.

# Synchronization: 同期

- To safeguard methods and objects from concurrent accesses, synchronization techniques are available.
- The `synchronized` modifier indicates to allow only a single thread to access the protected methods or objects at a time.
- This ensures that concurrent operations do not interfere with each other,
  - preserving data consistency
  - preventing potential issues stemming from concurrent accesses.

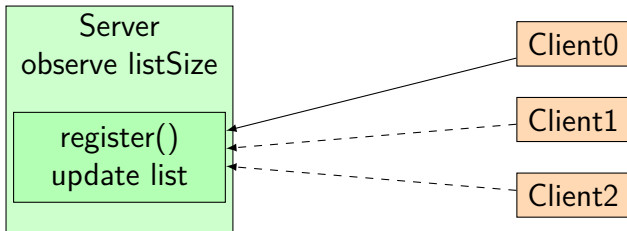
# Protection with *synchronized*



Only one of clients is allowed to access the resource.



# Thread.example1



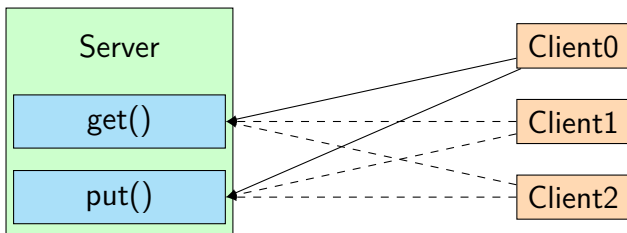
- Clients try to connect to the `register()` method by random durations.
- Only one of the clients is allowed to connect.

See `Thread.example1`

```
1 public void run() {  
2     while (running) {  
3         //waiting the list unlocked  
4         synchronized (messageList) {  
5             if (messageList.size() == max) {  
6                 running = false;  
7             }  
8         }  
9         try {  
10            Thread.sleep(10);  
11        } catch (InterruptedException e) {  
12        }  
13    }  
14 }
```

```
1 synchronized public void register(Client client,
2   int c, String dateStr) {
3   Date date = new Date();
4   //The time the client tries to connect and succeeds to connect
5   String ss = client + ":" + c + " "
6     + dateStr + "->" + date.toString();
7   messageList.add(ss);
8   System.out.println(ss);
9   try {
10    Thread.sleep(1000);
11  } catch (InterruptedException e) {
12  }
13 }
```

# Thread.example2



- The number of tokens equals to the number of clients.
- Clients try to get a token through `get()` method by random duration.
- After returning the token through `put()` method, the client is allowed to get another token.

See `Thread.example2`

# Client side

```
1 private void update(){
2     if(!tokens.isEmpty()){//put token if this has
3         running=server.put(this, tokens.poll());
4     }
5     Token t = server.get(this);//get token from the server
6     if(t!=null){
7         if(t==Server.falseToken)running=false;
8         else{
9             tokens.add(t);
10        }
11    }
12 }
```

# Server side

```
1 synchronized public Token get(Client client) {  
2     Token b = getSub(client);  
3     try {  
4         Thread.sleep(1000);  
5     } catch (InterruptedException e) {  
6     }  
7     return b;  
8 }
```

```
1 synchronized boolean put(Client client, Token t) {  
2     if (running) {  
3         putSub(client, t);  
4         try {  
5             Thread.sleep(1000);  
6         } catch (InterruptedException e) {  
7         }  
8     }  
9     return running;  
10 }
```

# Exercise

In `example0/SampleRunnable.java`, understand

- How to start the thread,
- When the thread stops,
- Which variable and method triggers the stop event.