Thread and runnable interfaces

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

Threads

- Threads are collections of independent sub-processes within an application that run concurrently.
- Threads have the ability to share data and variables among themselves.
  - This enables communication and collaboration between threads.
- Regarding threads in Java applications:
  - Threads play a crucial role in the concurrent execution of tasks.
  - Instances of GUI classes are specifically designed to run on threads.
  - Instances of any class in Java can be executed on threads.

Sample program download

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

Runnable interface

- The `Runnable` interface is an interface that defines a single method, `run()`.
- A class that implements the `Runnable` interface can be executed on a thread.
- The `run()` method is invoked only once at the start of the thread.
- 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- `stop()` method is obsolete and should not be used.
  - Stop the `run()` instead.

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

- `ExampleWithThread`
  - Starting the instance inside an anonymous implementation of the `Runnable` interface
- `ExampleRunnable`
  - Implementing the `Runnable` interface

See `Thread.example0`

Example class

```
1  public class Example {
2
3      protected volatile boolean running = true;
4      //flag to stop the thread
5      protected int c = 0;//counter
6      private final int id;//
7      private final int maxCount = 10;
8
9      public Example(int id) {this.id = id;}
10
11     public void update() {
12         Date date = new Date();
13         System.out.println(id + ":" + c + " " + date.toString());
14         c++;
15         if (c > maxCount) {//Stop after maxCount updates
16             running = false;//change the flag to stop the thread
17         }
18     }
19
20     public boolean isRunning() {return running;}
21 }
```

ExampleWithThread class

```
1   public static void main(String[] args) {
2       Thread thread0 = new Thread(new Runnable() {
3           //create an instance of Example
4           Example s = new Example(1);
5
6           @Override
7           public void run() {
8               while (s.isRunning()) {
9                   s.update();//update the state
10                  try {
11                      Thread.sleep(1000);//wait for 1 second
12                  } catch (InterruptedException e) {}
13              }
14          }
15      });
16      thread0.start();
17  }
```

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

ExampleRunnable class

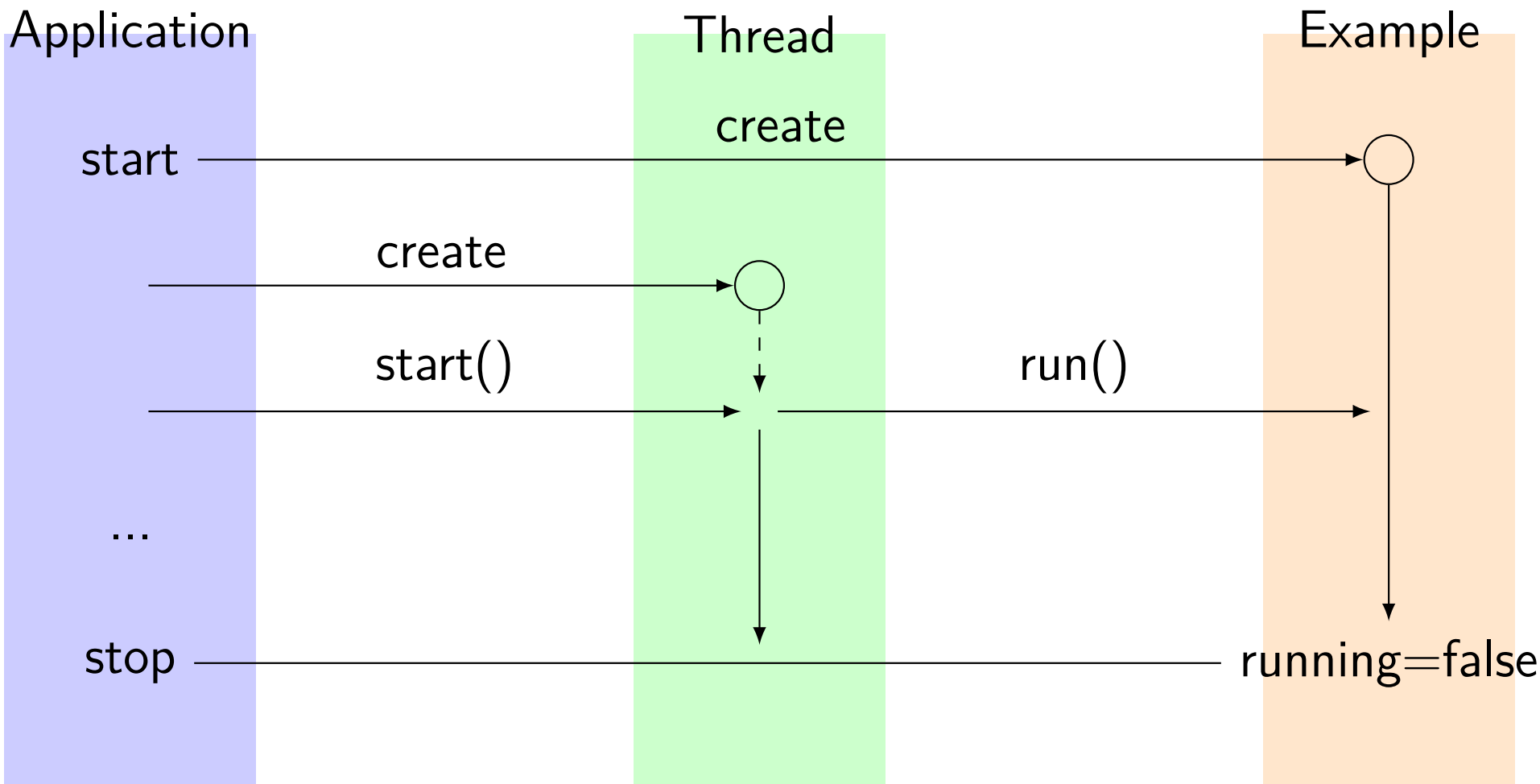
```
1  public class ExampleRunnable extends Example implements Runnable {
2
3      public ExampleRunnable(int id) {
4          super(id);
5      }
6
7      /**
8       * update() at random timing
9       */
10     @Override
11     public void run() {
12         while (running) {
13             update();//update the state
14             int t = (int) (1000 * Math.random());
15             try {
16                 Thread.sleep(t);//wait for t milliseconds
17             } catch (InterruptedException e) {
18             }
19         }
20     }
21     ...
22 }
```

Running three threads

```
1  public static void main(String[] args) {
2      new Thread(new ExampleRunnable(1)).start();
3      new Thread(new ExampleRunnable(2)).start();
4      Thread t = new Thread(new ExampleRunnable(3));
5      t.start();
6  }
```

- `Thread` class is initialized with an instance of `ExampleRunnable` class.
- The `start()` method is called to run the thread.

Flow of running thread



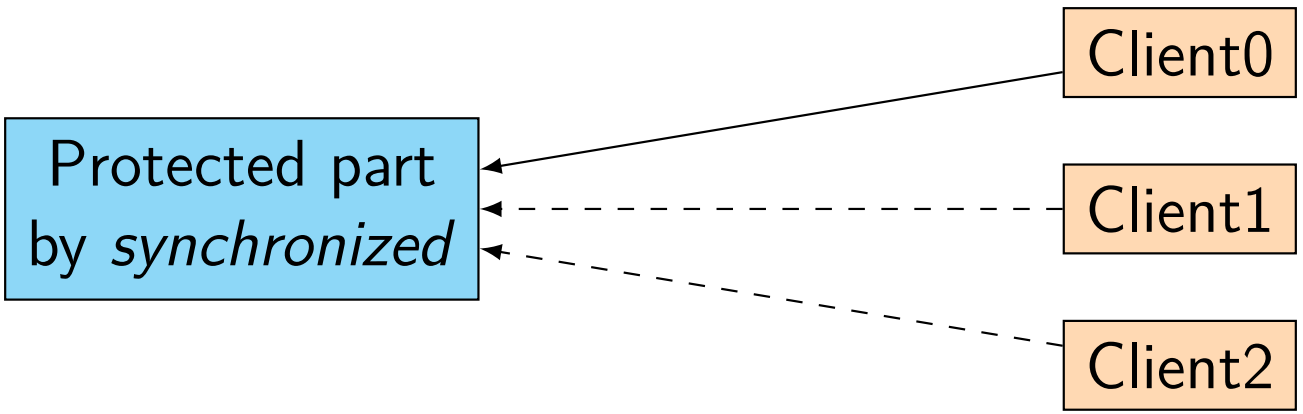
Asynchronously updates: 非同期更新

- 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 need to implement synchronization mechanisms for shared data updates.

Synchronization: 同期

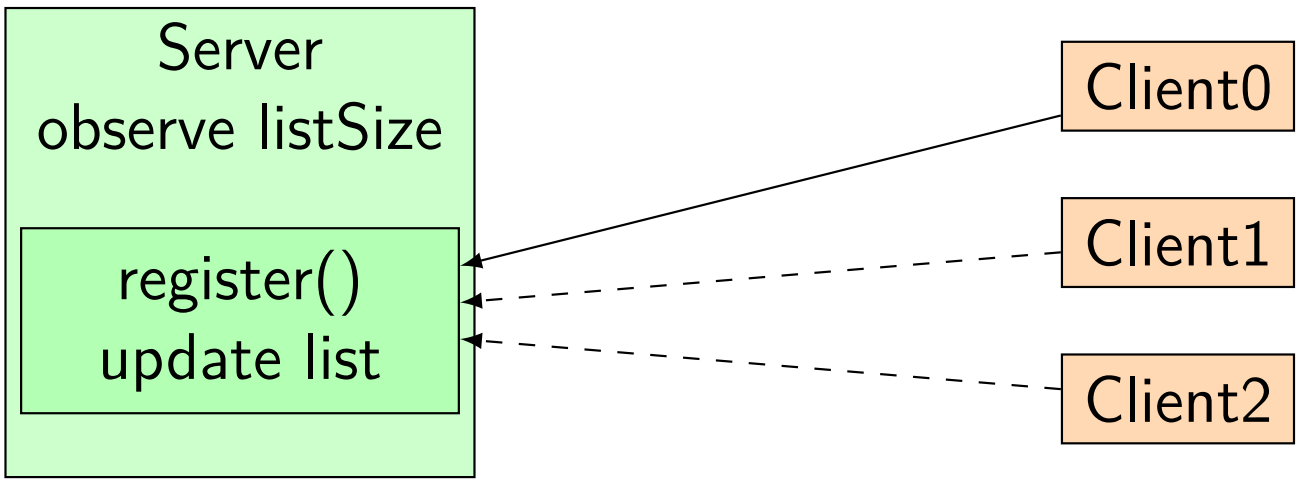
- To safeguard methods and objects from concurrent accesses, synchronization techniques are available.
- The `synchronized` modifier indicates to allow only a single thread to access the protected methods or objects at a time.
- This ensures that concurrent operations do not interfere with each other,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

Protection with *synchronized*



Only one of clients is allowed to access the resource.

Example 3.1: 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`

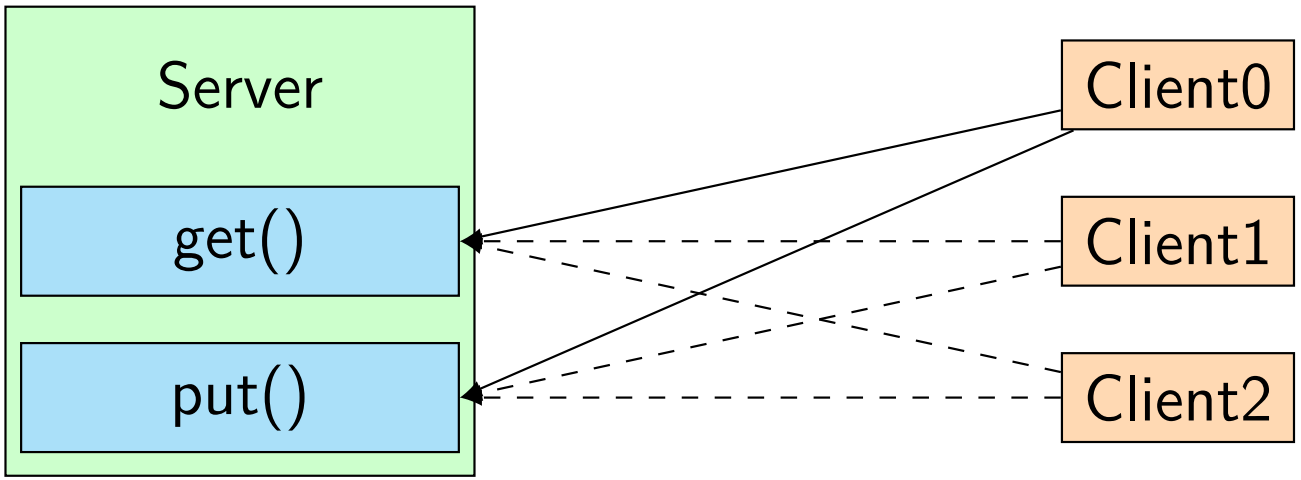
run() in Server class

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

register() method in Server class

```
1   synchronized public void register(Client client,
2           int c, String dateStr) {
3       Date date = new Date();
4       //The time stamp at the client 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  }
```

Example 3.2: 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()) {// if this has tokens
3          // put token to the server
4          // if the server is terminated, the return value is false
5          running = server.put(this, tokens.poll());
6      }
7      if (running) {
8          Token t = server.get(this);// get token from the server
9          if (t != null) {
10             if (t == Server.falseToken) {
11                 running = false;
12             } else {
13                 tokens.add(t);
14             }
15         }
16     }
17 }
```

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 }
```

Record class

- `Record` class is a simple data carrier class.
- `public record Token(int t){}` is equivalent to

```
1  public final class Token{
2      private final int t;
3
4      public Token(int t){this.t=t;}
5      public int t(){return t;}
6  }
```

- `equals()`, `hashCode()` and `toString()` methods are automatically generated.