2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Concurrency to decouple GUI from (long running) computations
- Threads, `SwingWorker`
- Interface Segregation Principle (ISP)

Need for Concurrency in GUI

- Main Event Loop (Java: *Event Dispatch Thread*) drives the GUI.
- Event handlers must return quickly to guarantee responsiveness:

  "Slow" event handlers cause the GUI to "hang".
- In Java, it is even impossible to force screen updates while event handlers runs.
- Solution: Run slow non-GUI code in a separate thread of control.

See: `download.oracle.com/javase/tutorial/uiswing/concurrency`

Need for Concurrency in GUI: Example

- See `SwingWorkerDemo` in `Threads.zip`
- Calculate n-th Fibonacci number (not recommended: why?):

```
/**
 * @param n  the index, starting at 0
 * @return the {@code}-th Fibonacci number
 */
private long fibRecursive(final int n)
        throws InterruptedException
{
    if (n <= 1) {
        return n;
    } else {
        return fibRecursive(n - 1) + fibRecursive(n - 2);
    }
}
```

Need for Concurrency to Improve Performance

- Processors (in hardware) offer limited performance.

  Even performance improvement in next generations is now limited.

- To get more work done, use more processors concurrently.

  Need to distribute the computation, and coordinate the threads.

- Beyond the scope of this course.

Concurrent Execution

- Expression evaluation and method execution are not atomic.

  For example, `++ x` is executed as sequence of smaller operations:

```
reg_i = x;               // copy x to local register
reg_i = reg_i + 1;       // increment register
x = reg_i;               // copy register back to x
```

- Concurrent execution of *threads* interleaves *low-level* operations.

  Thread may be interrupted at any time for work of other threads.

Concurrent Execution: Example

Two threads each execute `++ x` on shared variable `x`, initially `0`.

The final value of `x` depends on the order of interleaving.

The final can be `2` (as expected), but also possible is:

Thread 1	Thread 2
`x == 0`	
`reg_1 = x`	
`reg_1 = reg_1 + 1`	
	`reg_2 = x`
`x = reg_1`	
	`reg_2 = reg_2 + 1`
	`x = reg_2`
`x == 1`	

Concurrent Execution: Puzzle

100 threads each execute `++ x` 100 times.

Shared variable `x` is initially `0`.

The largest (and maybe expected) final value is `10000`.

What is the smallest possible final value?

Concurrent Execution: Puzzle (continued)

100 is possible:

1. All threads start by reading `x`.
2. They all read the same value `0`.
3. They all write the same value `1` into `x`.
4. This interleaving is repeated 100 times.
5. This establishes `x == 100`.

Is a smaller final value possible?

Concurrent Execution: Puzzle Solution

1. All threads read `x`, putting `0` into their register.

2. Thread T_0 completes 99 increments: now `x == 99`.

3. Threads T_1 through T_{98} complete all their 100 increments.
   (For instance, one after the other.
   This establishes `x == 100`, since each starts from `0`.)

4. T_{99} completes its first increment (having read `0`): now `x == 1`.

5. T_0 reads `x`, retrieving value `1`.

6. T_{99} does its remaining 99 increments: now `x == 100`.

7. T_0 does its final increment from `1` to `2`: now `x == 2`.

Concurrency: Dangers

- Operation interleaving is non-deterministic (not predetermined).

  Hence, not reproducible.

  This hinders reasoning, testing, and debugging.

- Shared data: How to guarantee invariants?
  - Result of two concurrent increments of variable `x`?
  - Concurrent update and printing of a time or date?

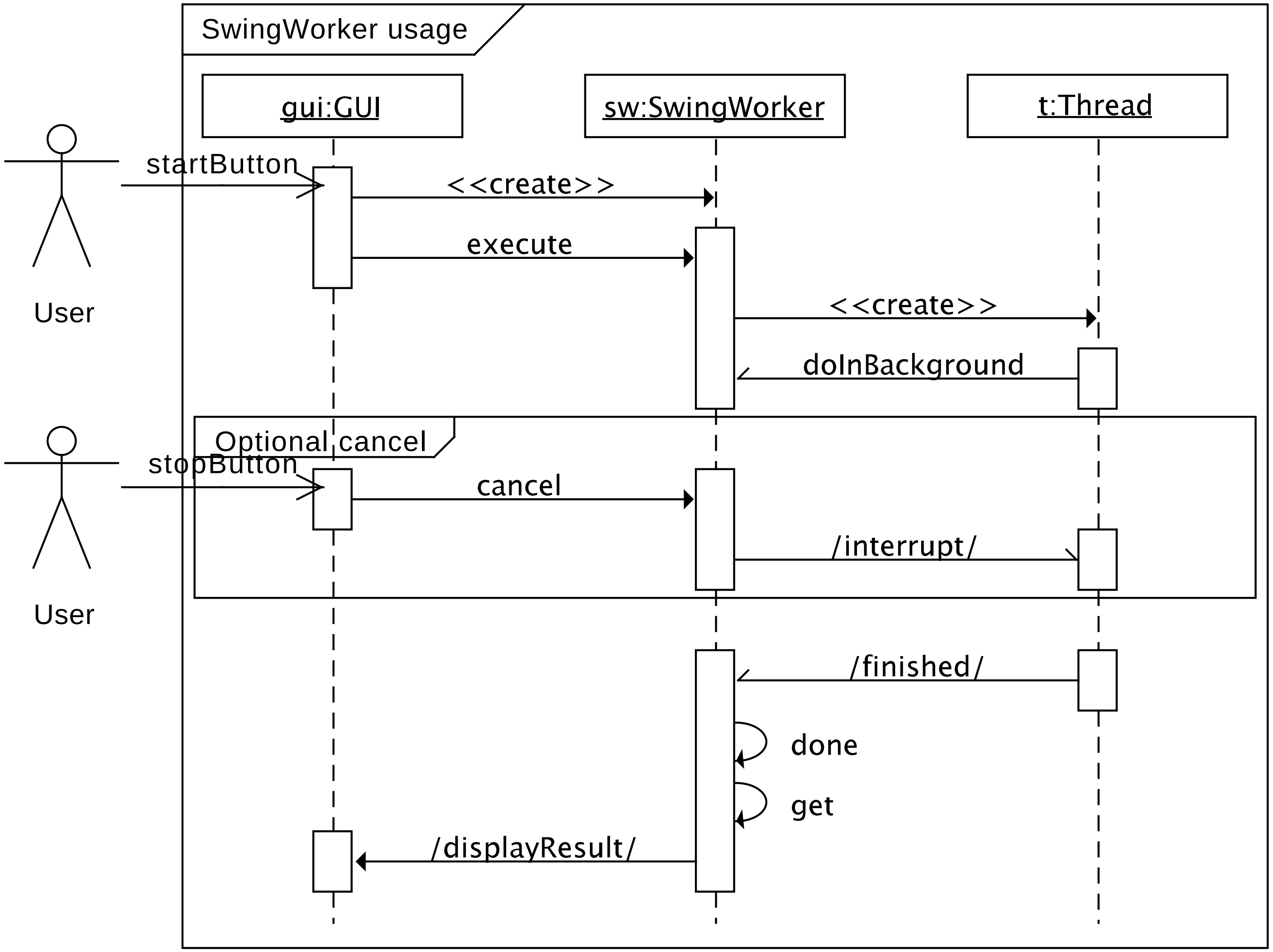
- *Thread interference, memory consistency errors, race conditions*

- First encounter: `SwingWorker`

`SwingWorker` Simplifies Threads for Swing GUI Apps

- `SwingWorker` acts as a *Façade* for the Java `Thread` facilities.
- It makes it easy to run some code in a background thread, and still interact with it in a controlled way.
- It uses the *Template Method* pattern to let clients define steps:
  - Which computation to do in background: `doInBackground()`
  - How to handle intermediate results: `process()`
  - What to do with the final result: `done()`

`SwingWorker` Sequence Diagram



`SwingWorker<T,V>`: Start and End

- `T`: type of result returned by the SwingWorker's `doInBackground` and `get` methods; to ignore result, use `Void`
- `@Override T doInBackground()`: implement background task here; do not call; is called automatically *in new thread*, after `execute()`
- `@Override` **`void`** `done()`: implement finalization here; do not call; is called automatically *in GUI thread* when background task ends
- Create new SwingWorker for each run of background task
- **`void`** `execute()`: call this in GUI thread to start background task
- `<T> get()`: call this in `done()` to get the result; catch exception

See `SwingWorkerDemo` in `Threads.zip`

`SwingWorker<T,V>`: Communication to GUI

- `V`: the type of intermediate results delivered by this SwingWorker's `publish` and `process` methods; to ignore, use `Void`
- `publish(<V>...)`: call *in background task* to deliver items to GUI
- `@Override` **`void`** `process(List<V> chunks)`: implement processing of delivered items here; do not call; is called automatically *in GUI thread*

  Separately `publish`ed items may be delivered in one `process` call.

See `SwingWorkerWithPublish` in `Threads.zip`

`SwingWorker<T,V>`: Abort

- Call `worker.cancel(boolean)` to abort background task

- Background task must *cooperate*; otherwise, it does not work

  1. In GUI thread, call `worker.cancel(true)`
     In background task, regularly inspect `Thread.interrupted()` and terminate if **`true`**

     `Thread.sleep/join/wait` then throw `InterruptedException`

  2. In GUI thread, call `worker.cancel(false)`
     In background task, regularly inspect `worker.isCancelled()` and terminate if **`true`**

- A `cancel` cannot be revoked

- After cancellation, `get()` will throw `CancellationException`

`SwingWorker<T,V>`: Abort Examples

See `Threads.zip`:

- `SwingWorkerDemo` uses `cancel(`**`true`**`)`

  and `Thread.interrupted()`

- `SwingWorkerWithPublish` uses `cancel(`**`true`**`)`

  and relies on `InterruptedException` thrown by `Thread.sleep()`

- `SwingWorkerWithProgressBar` uses `cancel(`**`false`**`)`

  and `isCanceled()`

`SwingWorker<T,V>`: Progress Reporting

- In GUI, register a `PropertyChangeListener` on bound property `"progress"`, to get updated value, and update progress bar

- In background task, regularly update progress,
  via `worker.setProgress(int)`

  N.B. `setProgress(int)` is **protected** in `SwingWorker`,
  so you need a workaround to access it from outside the worker

See `SwingWorkerWithProgressBar` in `Threads.zip`

Swing GUI Thread Rules

Quoting from the Concurrency in Swing tutorial:

- Some Swing component methods are labelled "thread safe" in the API specification;
- these can be safely invoked from any thread.
- *All other Swing component methods must be invoked from the event dispatch thread.*
- Programs that ignore this rule may function correctly most of the time, but are subject to unpredictable errors that are difficult to reproduce.

See: `docs.oracle.com/javase/tutorial/uiswing/concurrency`

SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, see Lecture 10)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, treated in this lecture)
- **D**ependency Inversion Principle (DIP, see Lecture 8)

Which Principle?

Which Principle?

Interface Segregation Principle: The Issue for Clients

```
class Service {
    void methodA1() { . . . }
    void methodA2() { . . . }
    void methodB1() { . . . }
    void methodB2() { . . . }
}

class ClientA { /* only uses Service.methodAx */ }

class ClientB { /* only uses Service.methodBx */ }
```

- When interface for `ClientB` changes, also `ClientA` ‘suffers’

Interface Segregation Principle

(More advanced principle)

- When designing a class for several clients with different needs,
- rather than loading the class with all methods that clients need
- and making each client depend on the complete interface,
- create specific interfaces for each kind of client,
- make each client depend only on 'its' interface, and
- implement all interfaces in the class.

Interface Segregation Principle: Solution for Clients

```
interface InterfaceA {
    void methodA1();
    void methodA2();
}
interface InterfaceB {
    void methodB1();
    void methodB2();
}

class Service implements InterfaceA, InterfaceB {
    . . . /* implement all methods */
}

class ClientA { /* uses InterfaceA.methodAx */ }
class ClientB { /* uses InterfaceB.methodBx */ }
```

Interface Segregation Principle: The Issue for Providers

```
interface Service {
    void methodA1();
    void methodA2();
    void methodB1();
    void methodB2();
}

class ProviderA implements Service { /* provide only methodAx */

class ProviderB implements Service { /* provide only methodBx */
```

- `ProviderA` also forced to implement `methodBx`
- `ProviderB` also forced to implement `methodAx`

Interface Segregation Principle: Solution for Providers

```
interface InterfaceA {
    void methodA1();
    void methodA2();
}
interface InterfaceB {
    void methodB1();
    void methodB2();
}

interface Service extends InterfaceA, InterfaceB {
}

class ProviderA implements InterfaceA { ... }
class ProviderB implements InterfaceB { ... }
```

ISP: Practical Example, Mouse Event Handling

```java
public interface MouseListener extends EventListener {
    void mouseClicked(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
}

public interface MouseMotionListener extends EventListener {
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}

public interface MouseInputListener
        extends MouseListener, MouseMotionListener { }
```

Motion events are segregated from regular (non-motion) events.

ISP: Practical Example (cont'd)

- Solution still not ideal; that is why they introduced

  `... class MouseAdapter implements MouseInputListener`

  It provides empty implementations of all mouse event handlers

- To define a mouse event handler that responds to only one type of mouse event:

  - `MyHandler extends MouseAdapter`
    (instead of `MyHandler implements MouseInputListener`)

  - `@Override` only that one method

  - Other handlers inherit empty implementation from `MouseAdapter`

- In Java 8: default implementations can be provided in interfaces

Homework

- (W7, C12) Adapt your `SimpleKakuroHelper` with
  - a text field for the maximum number,
  - a `SwingWorker` to do the calculation in the background,
  - Make it interruptable.
- (Optional in W6, G2) Make your backtrack solver run in a `SwingWorker` in the *Kakuro Puzzle Assistant*.

Final Exam

- Q & A Session / by e-mail ?
- Written exam on Fri 21 Apr 2017, 09:00–12:00
- Open theoretical and small design questions, about

  slides, handouts, Design Patterns book by Eddie Burris
- Final grade:
  - final exam (40%)
  - interim tests (20%)
  - programming homework (40%)

Summary

- Concurrency via Java threads

  `SwingWorker`

- Synchronization concerns

- Interface Segregation Principle