CS 478: Software Development for Mobile Platforms

Set 5: Multi-threading

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

March 14, 2019

Useful definitions

- *Process*: A self-contained program running with its own data
  - Processes execute in parallel (concurrently) with other processes (sharing such resources as memory, CPU, external storage, etc.)
  - In general, data not shared among processes
    - ➢ On multicore chips found in smart phones, processes can run *simultaneously,* each on a different core
  - *Simultaneous* vs. *concurrent* execution
    - ➢ Simultaneous = physically concurrent (at the very same time)
    - ➢ Logical concurrency sometimes through time-sharing one CPU
  - **Android**: In general each app runs on its own process
    - ➢ By default, all components in an app run in the same process

1

Useful definitions

- *Thread:* A different kind of concurrent unit than a process
  - Like a process, a thread has its own control flow, runs asynchronously with respect to other threads
    - ➢ Each thread has its own run-time stack, registers
  - Part of a process
    - ➢ A process can have multiple threads
  - However, a thread shares statically-allocated and dynamically-allocated objects with other threads in same process
  - Thread = A light-weight process?
  - **Android**: Each app given a thread called *main* or the *UI Thread*

See http://developer.android.com/guide/components/processes-and-threads.html

2

Process hierarchy

Used to determine what process to kill when system is low on resources

1. *Foreground process*: Required for what the user does now, may host either:
   - Resumed activity
   - Service bound to resumed activity
   - Service running lifecycle callbacks (e.g., *onCreate()*)
   - Broadcast receiver running *onReceive()*

- Android will not destroy a foreground processes except under catastrophic lack of resources (CPU time, RAM, etc.)

See https://developer.android.com/guide/topics/processes/process-lifecycle.html

3

Process hierarchy

2. *Visible process*: Does work the user can see or hear, may host either:
   – Activity in paused state
   – Service bound to paused activity
   – Service running in foreground (e.g., playing music)

• Fairly safe from automatic killing by OS even when resources are very scarce

See https://developer.android.com/guide/topics/processes/process-lifecycle.html

Process hierarchy

3. *Service process*: Process running a service in background (e.g., synchronizing with a server or downloading data)
   – Doing things user cares about but not affecting UX
   – Could be killed by OS
4. *Cached process*: Process running an activity that's not visible (stopped)
   – Least recently used processes are good targets for killing
   – Empty process: Process doing nothing (cached for future reuse?)

See https://developer.android.com/guide/topics/processes/process-lifecycle.html

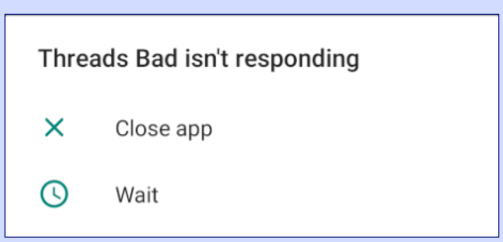
The *Main Thread*

- Android app has its own process but can contain multiple threads
- Apps we saw thus far had just one thread
- *Main thread*: App's initial thread when app starts execution
- Also known as the *UI thread*
  - **Main thread is the only app thread allowed to modify the app's display**
  - Main thread has an event queue where requests are posted (e.g., to update UI widgets or execute callbacks)
  - When new components are created (e.g., starting a service, executing *onReceive*()) they still run in the main thread

6

The *Main Thread* (cont'd)

- Be careful not to put too many computations in the UI thread
  - ➢ If the UI thread busy computing, it won't respond to new events
- Android checks periodically whether an app's UI is responsive or not
- If no response 5 seconds after user event, Android will display infamous *ANR* (*Application not Responding*) dialog to kill running application, possibly leading frustrated user to uninstall your app
  - ➢ You probably don't want that fate for your app…



7

The *Main Thread* (cont'd)

- Conclusion: If you must run long computations, spawn a worker thread and run computations in worker thread...
  - **... But then change the UI in the UI thread**
  - E.g., database operations, page download or other network access should never be done in UI thread...

8

The bottom line

- Two simple rules to live happily with Android threads:
  1. **Don't ever block the UI Thread (i.e., avoid ANR)**
  2. **Don't ever change the UI from another thread**
     - Only UI thread allowed to modify the UI
- Consequences:
  1. Use worker thread for long-running computations
  2. Let UI thread post computation results on device's display
- Two approaches to multi-threading Android apps
  – Java threading capabilities, e.g., *Thread* class and *Runnable* interface
  – Added-on Android threading functionality (e.g., *AsyncTasks* and *Handlers*)

9

Managing the threads

- Somewhat tricky
  - Must do background work in worker thread but update UI in main thread
  - The UI thread is not thread-safe!
    - Different threads will be reading and writing shared data structures
    - Access to shared structures by different threads must be synchronized to avoid simultaneous access and possibly inconsistent changes

10

Concurrency in Java

- Java language supported concurrency from the outset
- Two main units for defining threads
  - Class *Thread*—Instances of this class can be run in parallel with each other and with the main thread
  - Interface *Runnable*—Define objects that can be passed to thread instances for execution in parallel
- Both *Thread* and *Runnable* declare method *run()* which specifies code to be executed in a separate thread
- When program starts, user code is executed in a single (main) thread
- Programmer can spawn additional threads as program runs by creating and *start*ing additional *Thread* instances

11

Java's concurrency model

- Multiple threads can be run concurrently (either through logical or physical concurrency, if enough cores are not available)
  - ➢ Logical concurrency: Multiple threads share time slices on the same core or CPU
  - ➢ Physical concurrency: Multiple threads run at the same time on different cores

12

Java's concurrency model

Support for data structure sharing (i.e., objects and fields)

- Each Java object has a mutual exclusion lock associated with it
  - *Mutex* lock—Lock that can be held by a single thread at a time; other threads trying to grab the same lock will have to wait
  - Also known as **intrinsic lock** or **monitor lock**
- Objects accessed by multiple threads should be locked if one of the threads attempts to modify the object
- We'll see how…

13

Creating and running Java threads

Simple steps for creating a Java thread:

1. Create class that either:
   – **extends** *Thread,* or
   – **implements** *Runnable*
2. Create a new thread instance either by
   – Creating **new** instance of your *Thread* subclass, or
   – Creating **new** instance of predefined *Thread* class while passing a *Runnable* instance to it
3. Start the *Thread* instance that you just created, e.g., by
   – Calling *aThread.start()* will cause *aThread* start executing its *run()* method

14

Two recipes for creating threads

First recipe:

1. Define a subclass *MyThread* of *Thread*
   – *MyThread* will override method *run()* to do specific actions for that thread
   – Method *run()* in *Thread* does not nothing
   – E.g., **class** *MyThread* **extends** *Thread*
2. Create a **new** *MyThread()* instance
   – E.g., aThread := **new** *MyThread*() ;
3. Send message *start()* to the new instance of *MyThread*
   – E.g., aThread.*start()* ;
   – Now aThread will execute its *run()* method

15

Two recipes for creating threads (cont'd)

Second recipe:

1. Define a class that implements *Runnable*
   - Since *Runnable* declares abstract method *run()*, your class must implement (give code for) *run()*
   - E.g., **class** *MyRunnable* **implements** *Runnable* …
2. Create a **new** *Thread*, initialize the thread by passing as argument an instance of your *Runnable* realization (e.g., *MyRunnable*)
   - E.g., Thread aThread = **new** Thread(**new** *MyRunnable()*) ;
3. Send message *start()* to the new thread instance
   - E.g., aThread.*start()* ;
   - Now aThread will execute its *run()* method

16

Synchronized methods and blocks

- Basic mechanism for synchronizing access to shared structures
- **Synchronized block**: Locks object specified in *synchronized statement*
  - Two threads might be executing this code, but only one will be allowed to enter the *synchronized* block and access locked object
  - Syntax of synchronized statement:

```
synchronized(anObject){
    // statements to be executed by one thread at a time
}
```

  - Example:

```
synchronized(aPerson) {
  // read aPerson – aPerson accessed by multiple threads simultaneously
}
```

17

Synchronized methods and blocks (cont'd)

- **Synchronized method**: Locks the method's receiver
  - Other threads trying to execute synchronized methods on the same receiver must wait
  - If method is static, the class of the method is locked (i.e., calls to other static methods in same class must wait)
  - Syntax:

{<*access-level*>} <*return-type*> **synchronized** <*method-name*>(<*parameter-list*>)

  - Example:

```
public void synchronized add(int i) {
  …
}
```


A simple threading example

- Design button in UI that downloads and displays web page
  - This is a long running operation, defer to another thread

```
…
theButtonThatAteChicago.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
     new Thread(new Runnable() {
         public void run() {
             Bitmap b = loadImageFromNetwork(
                          "http://example.com/image.png");
             mImageView.setImageBitmap(b);
         }
     }).start();
  }
```

Set a button listener.

Create new thread on user click.

Thread downloads image and sets content of image view.

Start executing thread's *run()* method.

This seems quite right!

But it's wrong!

Why??!?

Fixing the problem: *runOnUiThread()*

- Various ways to fix above issue
- Use method *runOnUiThread(Runnable)*
- Must provide *Runnable* object that will modify the UI
- Call to *runOnUiThread()* does not block caller
- Resulting code can be complicated
  – Ugo's note: Isn't this always the case with Android?

20

Another solution: *AsyncTask*

- Special Android construct to make multithreading easier
- Goal: Delegate work away from UI Thread to a background thread
- *AsyncTask*: Background thread responsible for
  – Carrying out work
  – Reporting ongoing progress to UI thread
  – Returning final result to UI thread
- UI thread responsible for handling AsyncTask's results and updating display
  – Recall that only UI thread can update display
- Convenient alternative to *runOnUiThread(),* handlers and raw Java threads for short tasks (requiring 1-2 seconds at most)

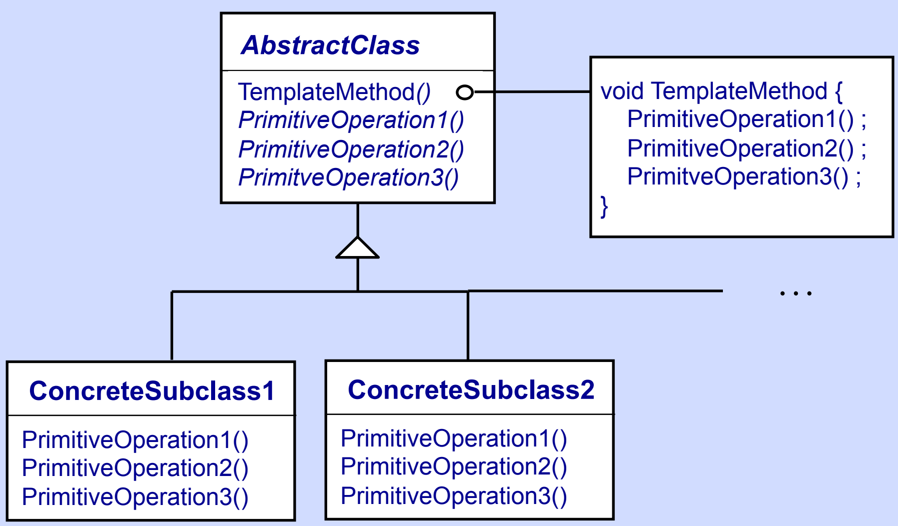
21

Division of labor

- UI thread
  - Creates instance of *AsyncTask* subclass, prepares for task execution, creates subclass instance, and calls *execute()* on instance
  - Displays progress reports while *AsyncTask* instance running
  - Updates display upon *AsyncTask* completion
- *AsyncTask*
  - Performs background work
  - Publishes progress status to UI Thread while running in background
  - Returns results of background work to UI thread
  - A realization of pattern *Template Method*

22

The *Template Method* design pattern



23

Template Method: Motivation

- Context of application: Common sequence of operations, where each operation is specialized by subclasses
- Example: Drawing a widget in Android UI always involves 3 operations
  1. onMeasure()
  2. onLayout()
  3. onDraw()
- Different widgets (*View* subclasses) do each operation differently
  – But each widget repeats exactly that sequence of operations

24

Template Method: Design

- Abstract superclass defines template method calling operation sequence
- Abstract superclass **declares abstract methods** for each operation
- Subclasses **define concrete methods** for each operation
- Execution sequence
  1. Subclass instance receives *TemplateMethod* call
  2. Superclass executes inherited *TemplateMethod* defined there
  3. Superclass's *TemplateMethod* calls each operation
  4. Operations in subclass of receiver are executed in the order specified by the template method
- Take advantage of message polymorphism (aka dynamic binding of messages and methods)

25

Template Method: Consequences

- Useful when a bunch of subclasses must follow a common sequence of operations
- Each subclass gets to specialize how each operation is conducted
- Operations need not execute in strict sequence (conditional and iterative operations are possible)
- Again, possible because of dynamic binding of messages and methods
  - Calls to *PrimitiveOperations* from superclass's *TemplateMethod()* get kicked back down to subclasses

26

AsyncTask's Java definition

- Generic class taking 3 type parameters:
  - *Params*: Type of parameters passed to async task upon execution
  - *Progress:* Type of progress units published by background thread
  - *Result*: Type of result
- *AsyncTask* subclass typically nested inside *Activity* class
- Example:
  - private class MyTask extends AsyncTask<String, Integer, Bitmap> { … }

  Source: http://developer.android.com/reference/android/os/AsyncTask.html
- Doc on Java variable parameter number:

  http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

27

Components of *AsyncTask* instance

- Typical *AsyncTask* subclass defines 4 key (protected) methods
  1. *onPreExecute()*
  2. *doInBackground()*
  3. *onProgressUpdate()*
  4. *onPostExecute()*
- Methods automatically called by OS after *AsyncTask* instance receives *execute()* call
- Methods 1, 3 and 4 executed in UI Thread
- Method 2 (*doInBackground()*) executed in worker thread

28

Components of *AsyncTask* instance

Four callbacks in lifetime of an *AsyncTask* instance

1. *onPreExecute()*: Callback automatically invoked before *doInBackground()*
   – Prepare for execution of background task
   – Run in UI thread, not *AsyncTask* thread (even though defined within *AsyncTask* subclass)
   – **void** return type, no input parameters
   – Typical action: set up a progress bar

29

Components of *AsyncTask* instance

Four callbacks in lifetime of an async task (cont'd)

2. doInBackground(Params …): Result

Does background work

- Runs in *AsyncTask* thread
- Called right after *onPreExecute()* returns
- Input: Array of type declared by generic argument *Params* when *AsyncTask* subclass created
- Output: Type declared by generic argument *Result*
- Can call *publishProgress(Progress …)* to update UI thread on current progress
- Example: `protected Bitmap doInBackground(String… strings) {`

30

Components of *AsyncTask* instance

Four callbacks in lifetime of an async task (cont'd)

3. onProgressUpdate(Progress…): **void**

Update progress display

- Runs in UI thread whenever *doInBackground()* calls *publishProgress()* (in background thread)
- Typical action: Update progress bar
- Input parameters of type specified by generic argument *Progress* when *AsyncTask* subclass was defined
- Example: `protected void onProgressUpdate(Integer… values)`

31

Components of *AsyncTask* instance

Four callbacks in lifetime of an async task (cont'd)

4. onPostExecute(Result): **void**

- Runs in UI thread after *doInBackground()* returns
- Result value returned by *doInBackground()* is automatically passed as input parameter to *onPostExecute()*
- Example: `protected void onPostExecute(Bitmap result)`

32

Use case of an *AsyncTask*

1. UI thread creates instance of (subclass of) *AsyncTask*
2. UI thread sends message *execute()* to *AsyncTask* instance
   - *execute()* args automatically passed to *doInBackground()*
3. Appropriate callbacks automatically called by OS (do not call any of them)
   - *onPreExecute()* – Run in UI thread before *doInBackground()*
   - *doInBackground()* – Run in *asyncTask* thread
   - *onProgressUpdate()* (zero or many times) – Run in UI thread after each *publishResult()* call in *doInBackground()*; argument(s) automatically passed to *onProgressUpdate()*
   - *onPostExecute()* – Run in UI thread after *doInBackground()* returns

33

Caveats on *AsyncTasks*

- *AsyncTask* instance **must** be created in UI thread
- Method *execute()* must also be called from UI thread
- **Each *AsyncTask* instance can only be executed once**
  - If multiple executions needed, create new instances
- Convenience advice: Nest *AsyncTask* class inside activity class
  - That way, *onPreExecute(), doInBackground()* and *onPostExecute()* will have access to UI fields that async task is likely to access and modify (e.g., in *onPostExecute()*)
  - If separate class used, then use getter and setter methods
- Either way, **beware of data races and deadlocks between async task and UI thread**

34

The benefits of *AsyncTasks*

- Support for forking and joining a background thread
- Support for passing input objects (of type *Params…*) to background thread
- Support for obtaining result value (of type *Result*) from background thread
- Support for posting progress indications (of type *Progress…*) from background thread
- Note: *Params* and *Progress* are variable arg type, *Result* is not
- Note II: Remember to keep track of different threads executing code in same file
  - Multiple flows of control in same class makes programming more exciting

35

The drawbacks of *AsyncTasks*

- In Java inner class instance gets implicit reference to fields of outer class
  - Typically, async task gets reference to fields of activity instance that creates async task
- If a configuration change occurs, activity will be paused, stopped and destroyed while async task is running
  - A new activity will be created, started and resumed
- However, old activity instance won't be deleted because async task still running and referencing fields of that activity
  - This is a (big) memory leak
- Worse yet, results of async task's execution will be posted on destroyed activity, not the new activity

36

The drawbacks of *AsyncTasks* (cont'd)

- The moral of the story:
  - If activity destroyed while task running, async task causes memory leak
  - Updates and result of async task will go to the wrong activity
- Consequence: Configuration changes are not automatically supported
  - Use async tasks only for very short background chores (< 2 seconds)
  - Make sure to handle configuration changes yourself
  - A good solution: Start *AsyncTask* from fragment, call *setRetainInstance(true)* on fragment, refer changes back to fragment
  - See, e.g., http://www.androiddesignpatterns.com/2013/04/retaining-objects-across-config-changes.html

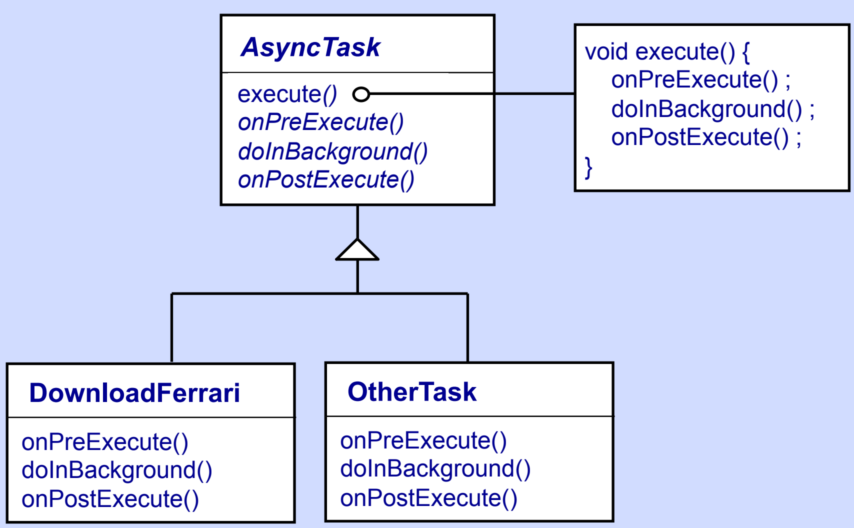
37

Sidestepping memory leaks in *AsyncTasks*

1. Declare *AsyncTask* subclass *static*
   - Subclass instance no longer holds reference to outer class
   - But now *AsyncTask* subclass can no longer reference non-static fields in enclosing activity
2. Pass needed fields to subclass constructor, assign to subclass fields
   - If fields are *Views*, they will still retain a reference to their context
   - The memory leak is still possible!
3. Turn subclass subclass fields into *weak references*
   - Weak references do not prevent garbage collection!
   - Also, check that references to fields have not become **null**
   - https://medium.com/freenet-engineering/memory-leaks-in-android-identify-treat-and-avoid-d0b1233acc8

38

Async tasks and *Template Method*



39

Classes *Handler, Looper,* and *MessageQueue*

- Support division of labor between any two arbitrary threads
  - *AsyncTask* only covered case of UI thread and a background thread
- Structure
  - A worker thread *T* can have a *Looper* and associated *MessageQueue*
  - Thread *T* creates one or more *Handler* instances associated with its message queue
  - Other threads use *T*'s handlers to add requests on *T*'s message queue
  - *T*'s looper takes requests from *T*'s message queue and executes them
- https://developer.android.com/reference/android/os/Handler.html
  https://developer.android.com/training/multiple-threads/communicate-ui#java

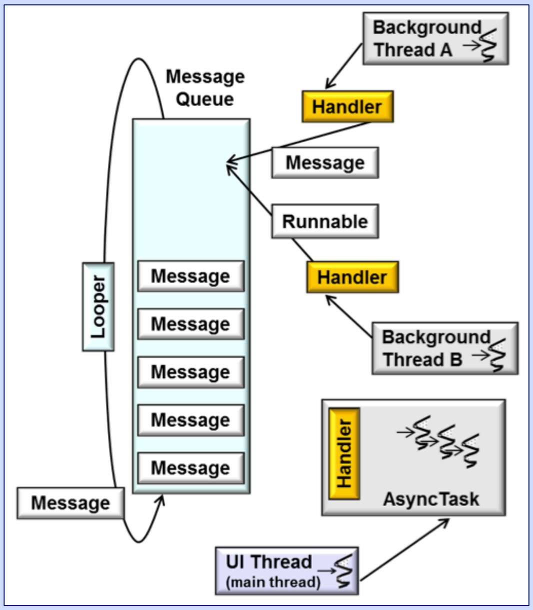
40

Relating loopers, message queues, and handlers

- A thread can have at most one looper and message queue
  - UI thread comes with looper and message queue
  - Looper and message queue must be created for worker threads
  - 1-to-1 correspondence among thread, its looper and message queue
- A thread with a looper and message queue can have zero, one or many handlers
- Each handler is associated with just one thread

41

The looper and message queue



Picture courtesy of Douglas C. Schmidt

Explanation:

Threads *A* and *B* send messages and post runnables on UI thread's message queue using UI thread's handlers.

42

The thread looper

- By default a thread does not have a *Looper* (except for UI thread)
  - Create and start one by calling *Looper.prepare()* and *Looper.loop()* in thread that wants looper
  - 1-to-1 correspondence among thread, its looper and message queue
- Looper executes a continuous loop in which it…
  1. Checks thread's message queue
  2. If not empty, dequeues first job
  3. Dispatches job to appropriate method
- Implementation: Check out *Looper* and *MessageQueue* classes

  https://developer.android.com/reference/android/os/Looper.html

43

Use case of *Handlers* and *Message Queues*

Use case of client and server threads

1. Server thread *S* creates and starts looper
2. Thread *S* creates handler on its looper
3. Client (e.g., an activity) running in thread *A* gets reference to *S*'s handler
4. Client uses handler to adds job to *S*'s message queue
5. Looper in *S* dequeues job and runs it
6. Client gets result from handler

Messages and Runnables

- Job in message queue can be either a **message** or a **runnable**
- Terminology:
  - **Send** a message
  - **Post** a runnable
- An implementation of the *Active Object* design pattern, which builds upon the *Command* pattern by the Go4
- Goal: Decouple method invocation from method execution in concurrent system
- Threads communicate by adding tasks on each other's job queues
- Caveat: Threads already exist
- Source: https://developer.android.com/reference/android/os/Handler.html

Handlers, loopers and design patterns

- *Active Object* design pattern
  - Invented by Greg Lavender and Doug Schmidt
- *Active Object* is an extension of *Command* pattern from Go4 system
- ~~Let's see *Command* first, then *Active Object*~~
- Highly recommended video: https://www.youtube.com/watch?v=U9Tf7h-etl0

46

Command pattern: The problem

Not Covered

- Suppose you are designing a system with a GUI (e.g., WYSIWYG editor)
- Build support for user-invoked commands (e.g., cut, paste, save, etc.)
- Requirements:
  1. Different commands have different interfaces
  2. Same command can be invoked in different ways by user
  3. Some commands should support Undo and Redo, others do not
- Obvious solution: Associate operations with user interface widgets!
- Of course, this is BAD, BAD, BAD
  - No support for (2) and (3) above (code duplication likely)
  - Also, proliferation and coupling of classes (each command likely to be implemented by several classes)

47

Command pattern: Solutions

Not Covered

- An improved solution
  - Action-performing views are subclasses of a special *View* subclass that takes command
  - Instances of *View* subclass can parameterize command execution by defining an *execute()* method that carries out the command
- Problems still:

1. Passing context to the command function (e.g., different commands will have different interfaces)
2. Undo and redo not adequately supported yet

48

Command pattern: Solutions (cont'd)

Not Covered

- Key idea: Associate an object, rather than a function, with each widget that can execute a command
  - Object is instance of a *Command* abstract class
  - *Command* instances embed executable code
  - *Command*'s API has *reversible()* method that returns boolean value
  - *Command*'s API has *execute()* method causing execution of code in command
  - Reversible *Command*s also remember previous state (e.g., before moving an object)
  - Store command history in a list

49

The *Command* pattern

Not Covered

Name

Command or *Action* or *Transaction*

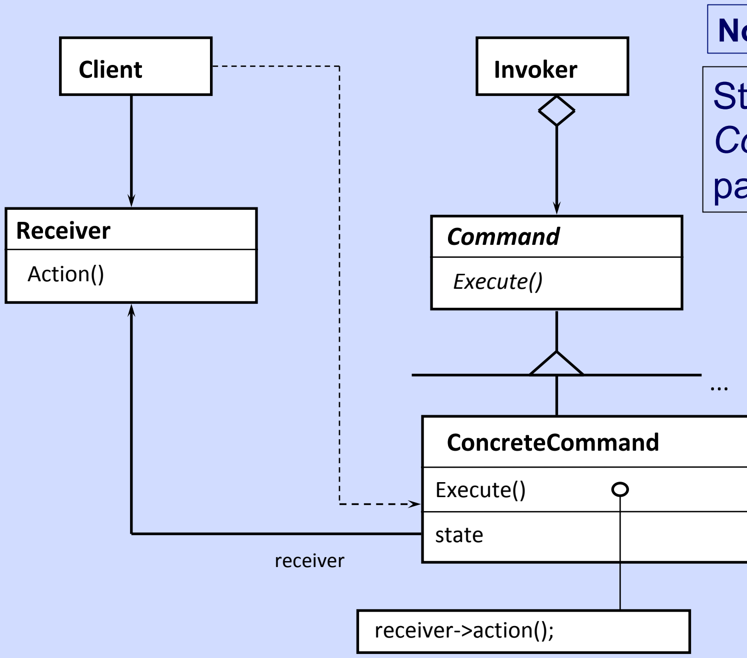
Applicability

- Parameterize objects by actions they perform
- Specify, queue, and execute requests at different times
- Support undo functionality by storing context information in command instances
- Support change log for recovery purposes
- Specify a transaction-based system (e.g., a database)

50

Not Covered

Structure of *Command* pattern



51

The *Command* pattern

Not Covered

Consequences:

1. Decouple object receiving a request from object carrying out request
   – Editor example: Different icons can be associated with the same command
2. Commands are first class objects
   – Easy to support *undo* and *redo* functionality (e.g., add state info to commands to avoid error hysteresis)
   – Copying of command objects may be required
3. Use simple commands to form complex ones (e.g., support for editor macros)
4. Easy to extend commands (not tied to interface)

52

The *Active Object* pattern

Name

Active Object or *Concurrent Object* or *Actor*

Applicability

- Clients accessing objects running in a different thread of execution

Goals

- Avoid blocking a server if a request is delayed
- Simplify synchronized access to a shared object
- Leverage concurrency available at hardware level

53

The *Active Object* pattern (cont'd)

Solution

- Decouple method invocation from method execution
- Call looks like normal call, forwarded to different thread under the hood

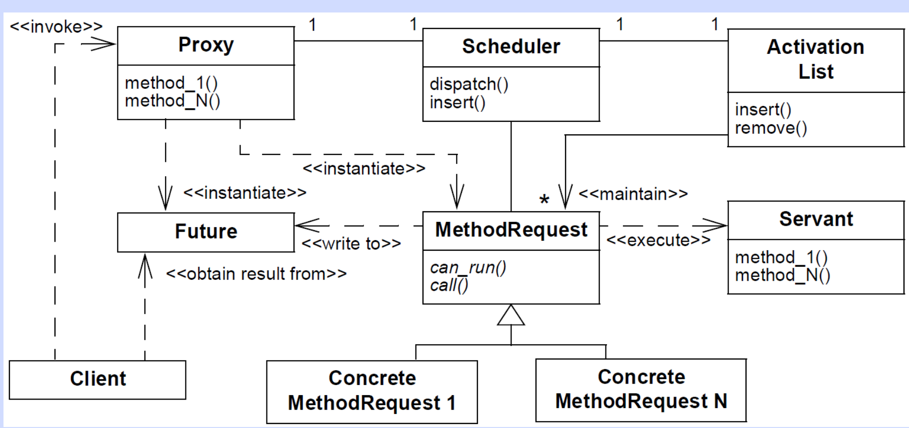
Participants

- Client—Performs method invocation
- Activation queue—Holds pending requests (invocations)
- Scheduler—Decides which request to service next
- Servant—Executes method requests
- Future—Returns results of servant execution to original requestor

54

Structure of *Active Object* pattern

Picture courtesy of Douglas C. Schmidt



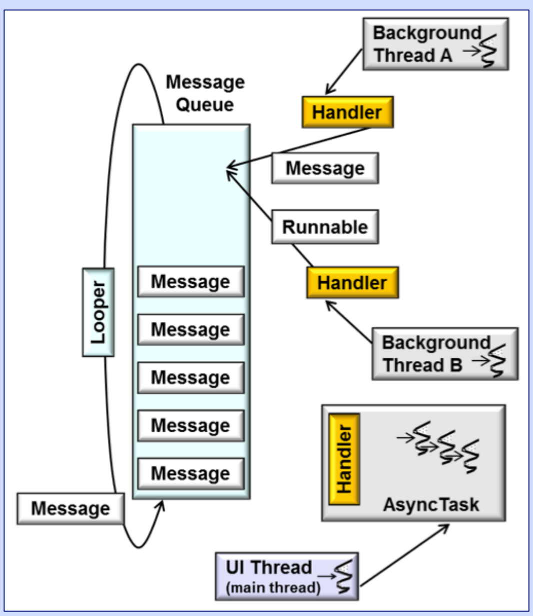
55

Advantages of *Active Object* pattern

- Simplify complexity of interthread synchronization and communication
  - Remote calls are transparent to caller (looking like local calls)
  - No worries about sockets or object locking, etc. as long as communicating threads operate in their own address space (no object sharing)
- Take advantage of multicore hardware
  - Create a thread pool and spawn threads for different computations
- Support scheduling of operations at future times
- But: Beware of run-time overheads
  - Context switches
  - Dynamic memory allocations (no buffer sharing between threads)
  - CPU cache updates

56

Recall looper diagram...



Picture courtesy of Douglas C. Schmidt

57

Setting up a looper in a worker thread

Use case of server thread:

1. Call *Looper.prepare()* to set up looper
2. Create one or more *Handlers* associated with this looper
3. Call *Looper.loop()* to start processing jobs in the looper's queue
4. Call *Looper.quitSafely()* to stop processing jobs
   – But currently posted jobs will be handled

Caveat:

- Attempting to set up multiple loopers in a thread causes a RT error
- UI thread comes with predefined looper
  – Just create handler(s), if needed

58

Setting up a looper thread...

- Code skeleton for setting up a thread with a looper

  Source: https://developer.android.com/reference/android/os/Looper.html

```
class LooperThread extends Thread {
   public Handler mHandler;

   public void run() {
      Looper.prepare();

      mHandler = new Handler() {
         public void handleMessage(Message msg) {
            // process incoming messages here
         }
      };
      Looper.loop();  }
}
```

Create new *Thread* subclass.

First, create a looper and message queue.

Second, create a handler.

Third, start the looper.

59

... Or using predefined class *HandlerThread*

- *HandlerThread* = *Thread* subclass that prepares looper automatically
- Use case:
  - Extend *HandlerThread* instead of *Thread*
  - Do not override method *run()* in your *HandlerThread* subclass
    - *HandlerThread.run()* defines and starts looper, must be executed
  - Specify thread actions in callback *onLooperPrepared()*
  - Subclass constructor must take a string argument (thread's name), pass string to superclass's constructor (i.e., *HandlerThread*'s constructor)
- https://developer.android.com/reference/android/os/HandlerThread
  https://developer.android.com/topic/performance/threads

Using a *HandlerThread*

- Skeleton code for *HandlerThread* subclass

```
class MyHandlerThread extends HandlerThread {
  Handler handler;
  public MyHandlerThread(String name) {
    super(name);
  }
  protected void onLooperPrepared() {
    handler = new Handler(getLooper()) {
      public void handleMessage(Message msg) {
        // process incoming messages here
        // this will run in non-ui/background thread
      }
    };
  }
}
```

Create *HandlerThread* subclass.

Class constructor.

After looper in place attach *Handler* to it.

Relevant *HandlerThread* source code

- *HandlerThread*'s *run()* method creates looper and starts it; don't override!
  - From jellybean/frameworks/base/core/java/android/os/HandlerThread.java:51—62.

```
class HandlerThread extends Thread {
  ...
  public void run() {
    mTid = Process.myTid() ;
    Looper.prepare() ;
    synchronized(this) {
        mLooper = Looper.myLooper() ;
        notifyAll() ; }
    Process.setThreadPriority(mPriority) ;
    onLooperPrepared() ;
    Looper.loop() ;
    mTid = -1 ;
  }
}
```

Definition of *HandlerThread* class.

The *run()* method prepares and starts the new thread's looper.

Also notify any clients that might be doing *wait()* on this thread.

62

Messages vs. runnables

- When job dequeued
  - If job is a runnable, call method *run()* on it
  - If job is a message, dispatch to method *HandleMessage()*
- Runnable: Same as before, implements *Runnable* Java interface by defining a *run()* method
- Message: Data structure carrying information about:
  1. Operation to be performed
  2. Operation's data
- Use a runnable when you know exactly what target thread must do; use message otherwise
- https://developer.android.com/reference/android/os/Message

63

The *Message* structure

- Messages are instances of class *Message*
- *Message* contains following **public** fields
  - *what* – Integer opcode, mutually agreed upon by message sender(s) and receiver thread
  - *obj* – Arbitrary data object
  - *arg1, arg2* – Additional integer values
- Additional key fields (package default access)
  - *data – A Bundle* specifying additional data to be sent
  - *target* – A *Handler* intended to receive this message
  - *callback* – The *Runnable* executed when message is handled
- Source code: https://android.googlesource.com/platform/frameworks/base/+/pie-release/core/java/android/os/Message.java

64

Message instance creation

- Two convenient ways to create instance of *Message* class
  1. Static message *Message.obtain()*
     - Returns message instance from global pool
     - Wildly overloaded with different initializers for return message
     - http://developer.android.com/reference/android/os/Message.html
  2. Use instance message *aHandler.obtainMessage()*
     - Similar behavior to *Message.obtain()*, but also sets target handler to *aHandler*
     - Again, multiple overloadings available

65

Message behavior

- Key *Message* methods
  - *getTarget(), setTarget(Handler)* – Getter + setter for message's target (a *Handler* instance)
  - *setData(Bundle), getData()* – Getter + setter for additional data (as a *Bundle*)
  - *sendToTarget()* – Sends message to target *Handler*
  - Public fields are typically accessed directly by clients

66

Relevant *Handler* methods

- *post(Runnable)* – Adds runnable to receiver (*Handler* instance)
  - Use only for runnables
- *postDelayed(Runnable, long)* – Adds runnable to receiver (*Handler* instance) but delay execution by specified number of ms
  - Use only for runnables
- *postAtTime(Runnable, long)* – Adds runnable to receiver (*Handler* instance) at time specified by second argument in ms
  - Second arg == uptime (could be delayed by sleep time)
- *postAtFrontOfQueue(Runnable)* – Adds runnable to front of receiver's queue
- See: http://developer.android.com/reference/android/os/Handler.html

67

Relevant *Handler* methods (cont'd)

- *sendMessage(Message)* – Adds message to receiver (a *Handler* instance)
  - Use only for messages
- *sendMessageAtFrontOfQueue(Message)* – For urgent messages?
- *sendMessageDelayed(Message, long)* – At uptime (ms)
- *sendMessageAtTime(Message, long)* – After delay (ms)
- See: http://developer.android.com/reference/android/os/Handler.html

The looper's role

- Dispatch messages and runnables in handler queue to appropriate destination
- Runnable: Just dequeue *Runnable* from handler queue and call *run()* method in receiving thread
- Message: Call method *handleMessage()* of *Handler* in receiver's thread
  - Sender wants an operation to be run in thread of receiving handler
- *sendMessage(Message)* – Adds message to receiver (a *Handler* instance)
  - So, use either *handler.sendMessage(message)* or *message.sendToTarget()* to send message to handler

Use case: Posting runnable to handler

Suppose thread A wants some work done in worker thread B

1. Thread A creates a runnable
2. Thread A uses some kind of "post" method to place runnable on thread B handler queue
3. Thread B's looper eventually dequeues runnable
4. Thread B executes runnable, as per thread A's wish

- Doc: http://developer.android.com/reference/android/os/Handler.html
- Source: https://android.googlesource.com/platform/frameworks/base/+/pie-release/core/java/android/os/Handler.java

Message handling

- Typical structure of *handleMessage()* is a big **switch** statement, based on *message.what()*
  - From *MediaPlayerService.java* in *Music* app

```
private Handler mMediaplayerHandler = new Handler() { …

   public void handleMessage(Message msg) {
     switch (msg.what) {
     case SERVER_DIED:
          if (mIsSupposedToBePlaying) {
             gotoNext(true); } …
             break;
     case TRACK_WENT_TO_NEXT:
          …
          break;
     case TRACK_ENDED:
          …
        break;
```

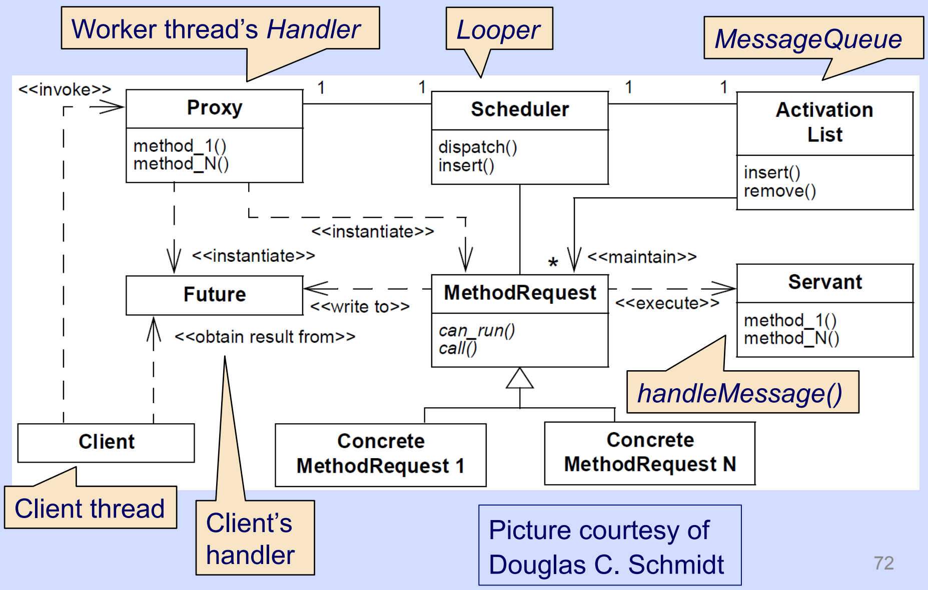
Create new *Handler*.

Define *handleMessage()* method.

Depending on *msg.what*, do different things.

Customary to use symbolic names for opcodes (*static final int*)

Handlers and *Active Object* pattern



Picture courtesy of Douglas C. Schmidt

Conclusions on *Handlers*

- A more powerful mechanism to use than *AsyncTasks*, but less convenient to use
  - *Messages* and *handleMessage()* seem to be used more often than *Runnables*
  - Sender thread can use runnable only when it knows exactly what work needs to be done by receiving thread

Summary of options for reporting to UI thread

Options for a worker thread to report computation results to UI thread:

- *View.post(Runnable)*
- *Context.runOnUiThread(Runnable)*
- Define, instantiate and execute an *AsyncTask*
- Use a *Handler* to communicate with UI thread
- Use ad-hoc Java synchronization

Other options available; these are just the most popular