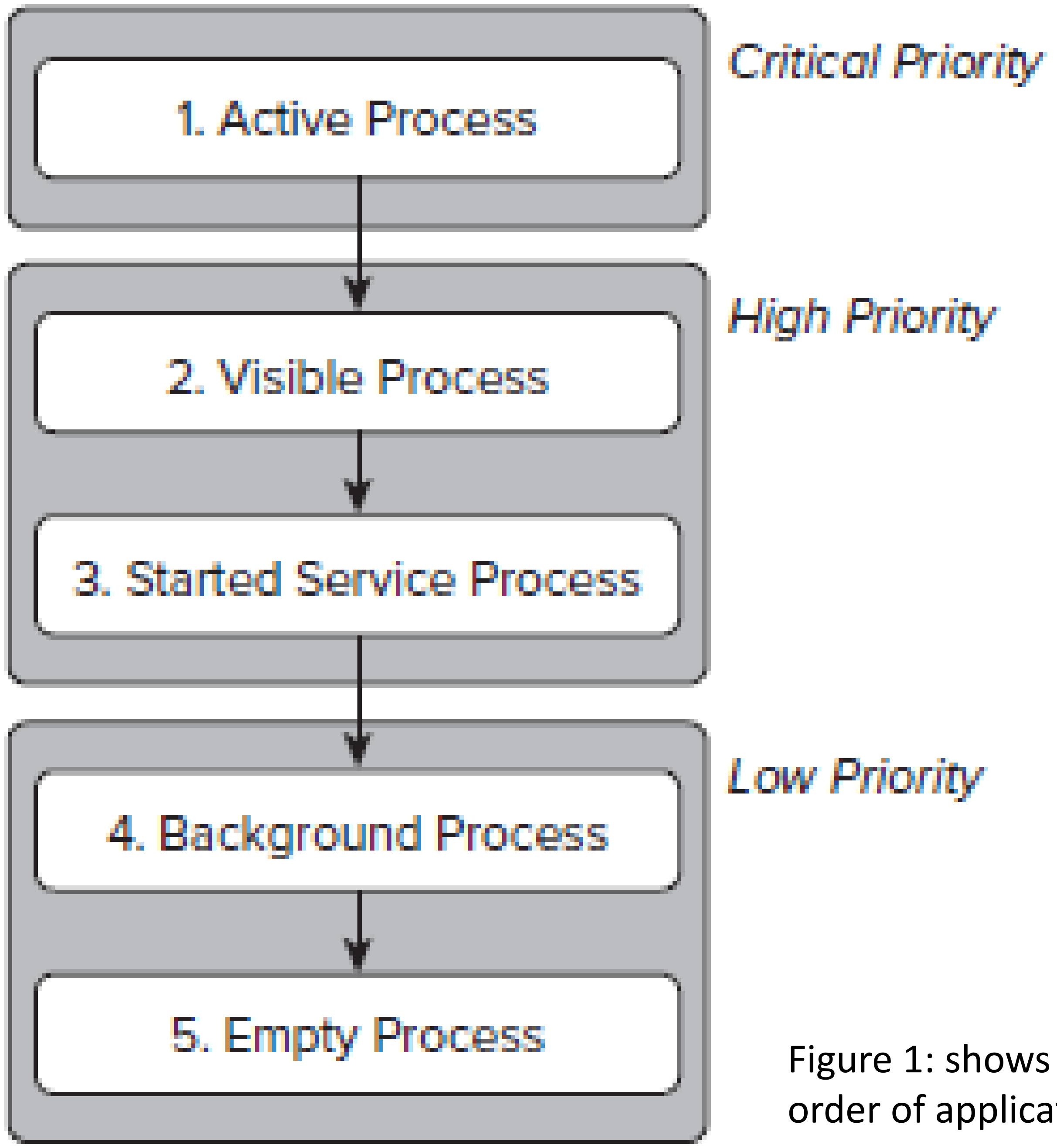
UNIVERSITY OF PLYMOUTH

COMP2000: Software engineering 2

Processes priorities and Threading in Java

Outline

- Introduction to processes priorities
- Threading in Java



It's important to structure your application to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important, or it could remain running when it is no longer needed.

Figure 1: shows the priority tree used to determine the order of application termination.

- The following list details each of the application states shown in Figure 1, explaining how the state is determined by the application components of which it comprises:
- **Active processes** — Active (foreground) processes have application components the user is interacting with.
- These are the processes Android tries to keep responsive by reclaiming resources from other applications.
- There are generally very few of these processes, and they will be killed only as a last resort.
- Active processes include the following:
  - Activities in an active state — that is, those in the foreground responding to user events.
  - Broadcast Receivers executing *onReceive* event handlers.
  - Services executing *onStart*, *onCreate*, or *onDestroy* event handlers.
  - Running Services that have been flagged to run in the foreground.

- **Visible processes** — Visible but inactive processes are those hosting “visible” Activities. As the name suggests, visible Activities are visible, but they aren’t in the foreground or responding to user events.
- This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity).
- There are generally very few visible processes, and they’ll be killed only under extreme circumstances to allow active processes to continue.
- **Started Service processes** — Processes hosting Services that have been started. Because these Services don’t interact directly with the user, they receive a slightly lower priority than visible Activities or foreground Services.
- Applications with running Services are still considered foreground processes and won’t be killed unless resources are needed for active or visible processes.
- When the system terminates a running Service it will attempt to restart them (unless you specify that it shouldn’t) when resources become available.

- **Background processes** — Processes hosting Activities that aren’t visible and that don’t have any running Services.
- There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern in order to obtain resources for foreground processes.
- **Empty processes** — To improve overall system performance, Android will often retain an application in memory after it has reached the end of its lifetime.
- Android maintains this cache to improve the start-up time of applications when they’re relaunched. These processes are routinely killed, as required

Threads

- Threads are the cornerstone of any multitasking operating system and can be thought of as mini-processes running within a main process, the purpose of which is to enable at least the appearance of parallel execution paths within applications.

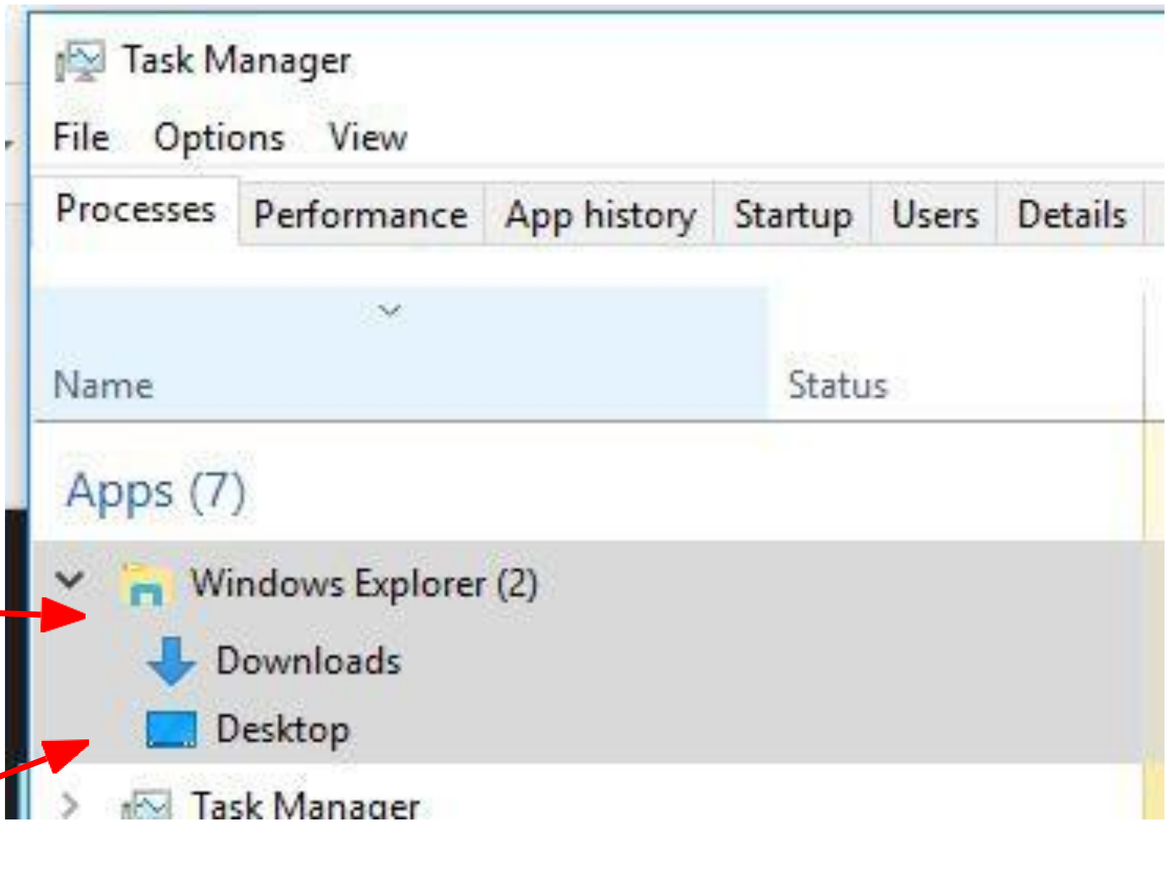
- When an Android application is first started, the runtime system creates a single thread in which all application components will run by default. This thread is generally referred to as the main thread ("main.").

- The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface.

Threads VS Processes

Processes

- Programs running on your computer
- Also have sub-processes



Threads VS Processes

Threads

- Each process/sub-process can have any number of threads

Threads in Java

- A thread is an object that can be assigned to a processor
- Threads allow for ‘flow control’ of a code base

We are already using Threads!

- The JVM (Java Virtual Machine) is a process.
- In the JVM, the ‘main’ method is a Thread!
- IntelliJ and Android studio also has threads for its editor

Benefits of using Threads

- Efficient computer processor usage
- Can take turns executing heavy processing loads
- Split larger processing work into smaller units
- Take advantage of multi-core processors
    - Most computers/devices have these today

Thread safety

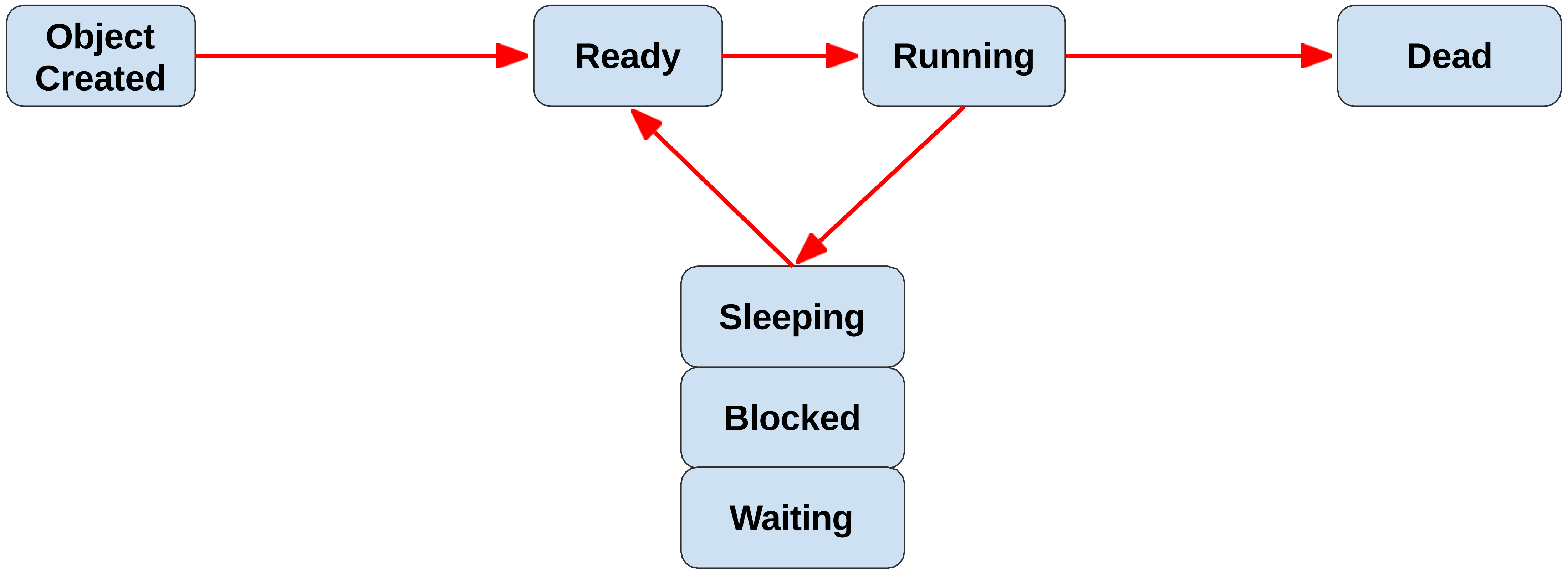
Running certain code on threads may cause problems (collisions/race conditions)

This code is always contained within a try/catch block:

- Sending a thread to sleep (pause) for a time

Note:
Use the InterruptedException class for the catch block parameter

Thread lifecycle



- Any component within an application that performs a time consuming task using the main thread will cause the entire application to appear to lock up until the task is completed.

- This will typically result in the operating system displaying an "Application is unresponsive" warning to the user.
- Clearly, this is far from the desired behaviour for any application.
- In such a situation, this can be avoided simply by launching the task to be performed in a separate thread, allowing the main thread to continue unhindered with other tasks.
- It is also almost always the thread in which your application interacts with components from the Android UI toolkit (components from the android.widget and android.view packages).

- As such, the main thread is also sometimes called the UI thread.

- The system does *not* create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread.

- Methods that respond to system callbacks (such as onKeyDown() to report user actions or a lifecycle callback method) always run in the UI thread of the process.

- For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue.

- When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly.

Examples of time-consuming processes?

- If everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI.

- When the thread is blocked, no events can be dispatched, including drawing events.
- From the user's perspective, the application appears to hang.

- Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "application not responding" (ANR) dialog.

- The user might then decide to quit your application and uninstall it if they are unhappy.

- The Android UI toolkit is *not* thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread.

- There are simply two rules to Android's single thread model:
  - Do not block the UI thread
  - Do not access the Android UI toolkit from outside the UI thread

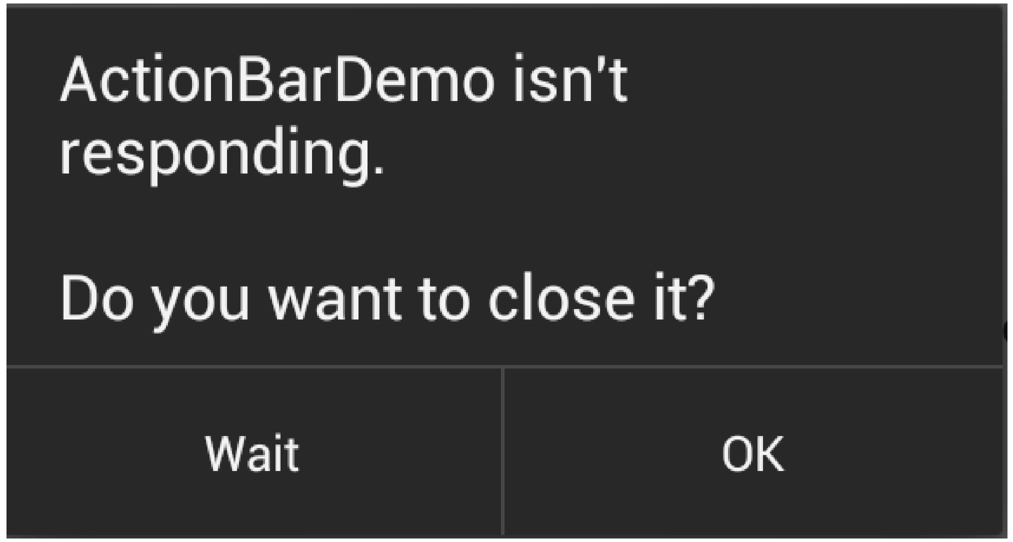
Worker threads

- To make sure that your application’s UI stay that you must do not block the UI thread

  - What do you mean “Do not block the UI Thread?”.
  - Anything that does any processing at all will block the UI Thread.
  - How long is acceptable? ¼ second? 1 second? 2 seconds?

- And if you block the main Thread in Android for more than 5 seconds, which seems a loooong time to me, you see this:

ANR or App Not Responding


ActionBarDemo isn't responding.

Do you want to close it?

- You should make sure to do the process that might take long time in separate threads ("background" or "worker" threads).

Important note:

you cannot update the UI from any thread other than the UI thread or the "main" thread.

- To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

- Activity.runOnUiThread(Runnable)

- View.post(Runnable)

- View.postDelayed(Runnable, long)

```java
public void onClick(View v) {

    new Thread(new Runnable() {
        public void run() {

            // a potentially time consuming task

            final Bitmap bitmap =
                    processBitMap("image.png");

            imageView.post(new Runnable() {

                public void run() {
                    imageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

This implementation is thread-safe: the background operation is done from a separate thread while the ImageView is always manipulated from the UI thread.

Using AsyncTask

- AsyncTask allows you to perform asynchronous work on your user interface.
-
- It performs the blocking operations in a worker thread and then publishes the results on the UI thread, without requiring you to handle threads and/or handlers yourself.

- To use it, you must subclass AsyncTask and implement the doInBackground() callback method, which runs in a pool of background threads.

- To update your UI, you should implement onPostExecute(), which delivers the result from doInBackground() and runs in the UI thread, so you can safely update your UI.

- You can then run the task by calling execute() from the UI thread.

Thank you