EMBEDDED SYSTEMS PROGRAMMING 2017-18

Multitasking

PARALLELISM

- **Bit-level parallelism**: increasing the word size
- **Instruction-level parallelism**: adding more functional units that can operate in parallel
- **Task parallelism**: adding multiple processors/cores to execute multiple programs concurrently
- "Law of diminishing returns" applies

PROCESSES AND THREADS

- **Process** (aka task): an instance of computer program currently being run by the operating system.
  Different processes do not share resources

- **Thread**: a unit of processing handled by the operating system scheduler.
  A process may contain multiple threads sharing the resources of the process (memory address space, file handlers, etc.)

MULTITASKING (1/3)

- (More appropriately: multithreading)
- It is the ability of **handling multiple streams of execution** (SEs) **concurrently.**
  The number of SEs may be higher than the number of execution units (EUs)
- Multiple EUs available: **parallelism**
- Only one EU available: **illusion of parallelism**

CONCURRENT PROGRAMMING

- Programming with multiple streams of execution (processes, threads)

- SEs can communicate with one another: this fact may cause interference

- Machine-level instructions in different SEs are interleaved in an unpredictable, non-deterministic way. If an order is required, it must be imposed explicitly

CONCURRENCY ISSUES (1/2)

- **Thread interference: race condition**
  Multiple SEs manipulate common data (e.g., increment a common variable) assuming a particular interleaving of operations which is not always true

- **Memory consistency errors**
  A SE reads data being concurrently manipulated by another SE <u>before</u> the manipulation is over: retrieved data are inconsistent

- Solution: locking

LOCKING

Several forms, depending on circumstances

- Access to the shared resource is *serialized* (**mutual exclusion**) via a *binary semaphore* (aka *mutex*) for both read and write operations
- Read access is always allowed, only write access is serialized
- (Read and/or write) access for $k \geq 2$ streams is allowed via a *counting semaphore*

CONCURRENCY ISSUES (2/2)

Mutual exclusion, locking, and synchronization in general, introduce **contention**

- **Starvation**
  A SE is unable to gain regular access to the resources it needs, therefore it cannot make progress.
  Livelock: starvation caused by SEs being too busy synchronizing with each other to perform actual work

- **Deadlock**
  Two SE are waiting for the other to finish, hence neither ever does

THREAD SAFENESS

- A piece of code is said to be **thread safe** if it can be safely invoked by multiple, simultaneous threads

- Thread-safe code is guaranteed to be free from race conditions and memory consistency issues
- Thread-safe code is usually — but not necessarily — designed to limit contention

CONCURRENCY: JAVA

- Concept of thread,
  associated with an instance of the class `Thread`.
  Every program has at least one thread
  and can create more

- Support for synchronization via the `wait()`, `notify()` (`Object` class) and `join()` methods (`Thread` class)

- Support for mutually exclusive access to resources with the `synchronized` keyword

THREAD CLASS

- Implements the interface `Runnable` with the single method `run()`, which contains the code to be run. In the standard implementation, `run()` does nothing

- Two strategies for creating a new thread

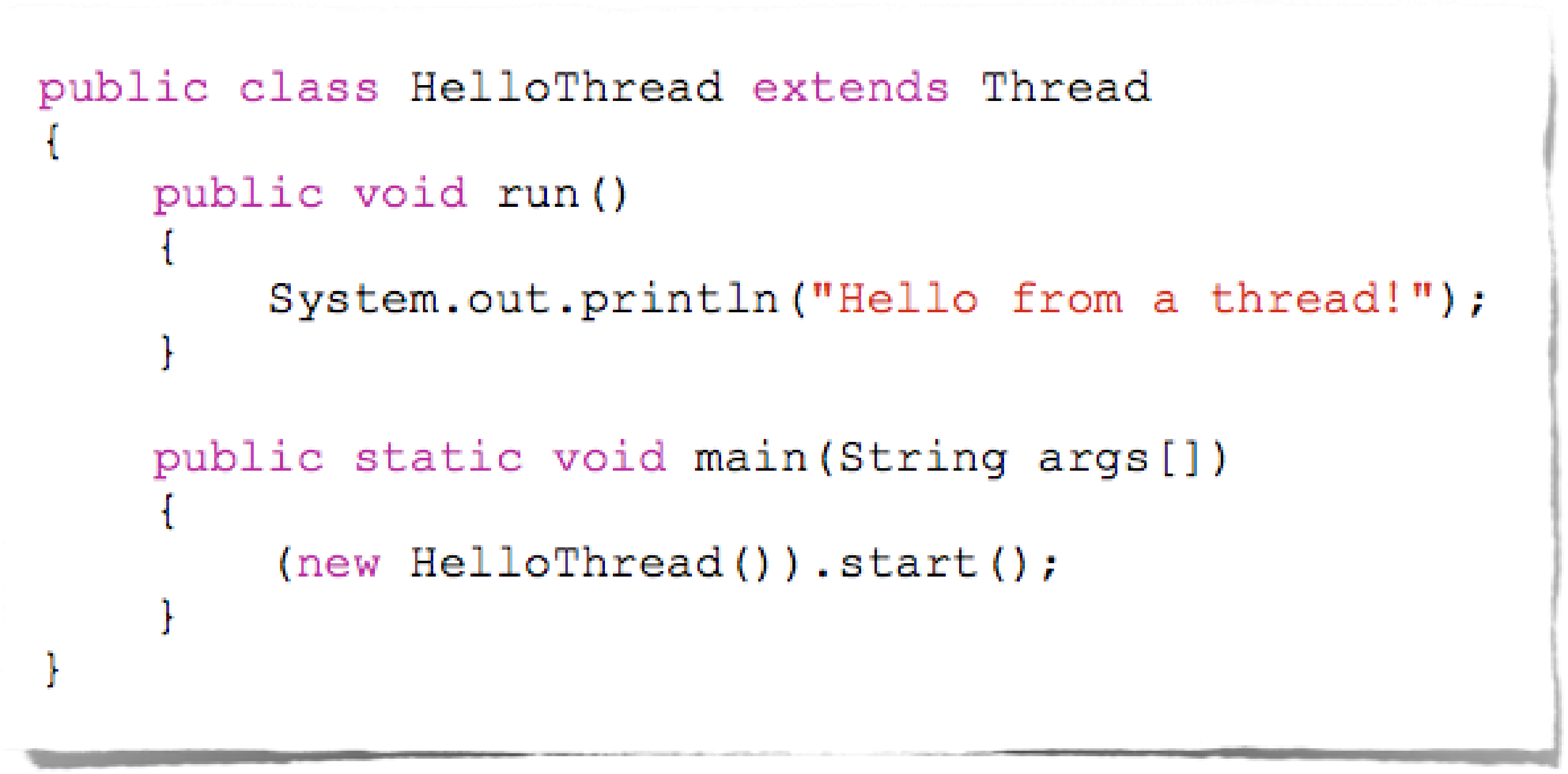
  1. Instantiate a class derived from `Thread`

  2. Create an instance of `Thread`, and pass to the constructor an object implementing `Runnable`

CREATING A THREAD (1/2)

First strategy

- Subclass `Thread` and override the `run()` method, then create an instance of the subclass



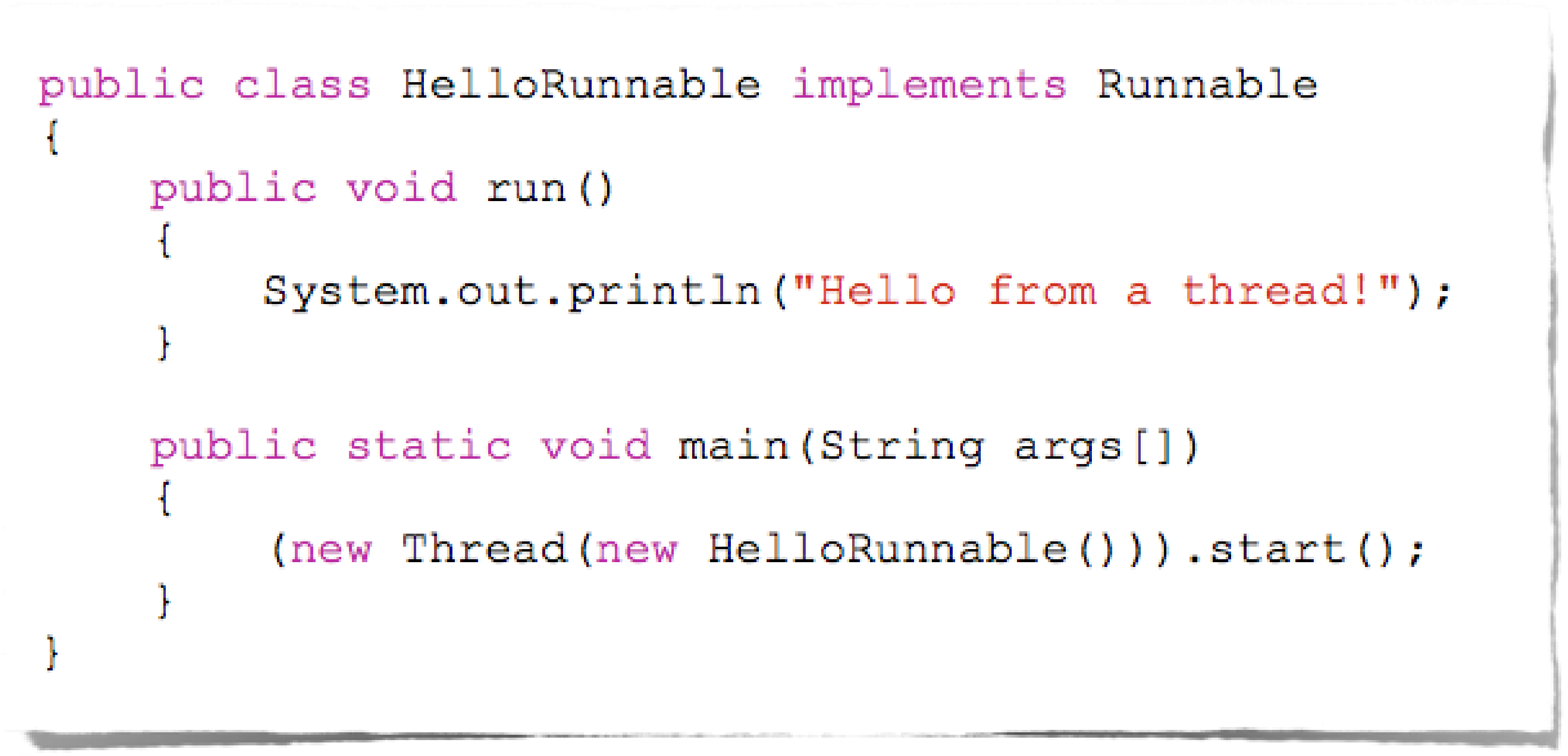
```
public class HelloThread extends Thread
{
    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new HelloThread()).start();
    }
}
```

Code: docs.oracle.com

CREATING A THREAD (2/2)

Second strategy

- Create an instance of `Thread`,
  pass a `Runnable` object to the constructor



```
public class HelloRunnable implements Runnable
{
    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Code: docs.oracle.com

THREAD CLASS: SOME METHODS

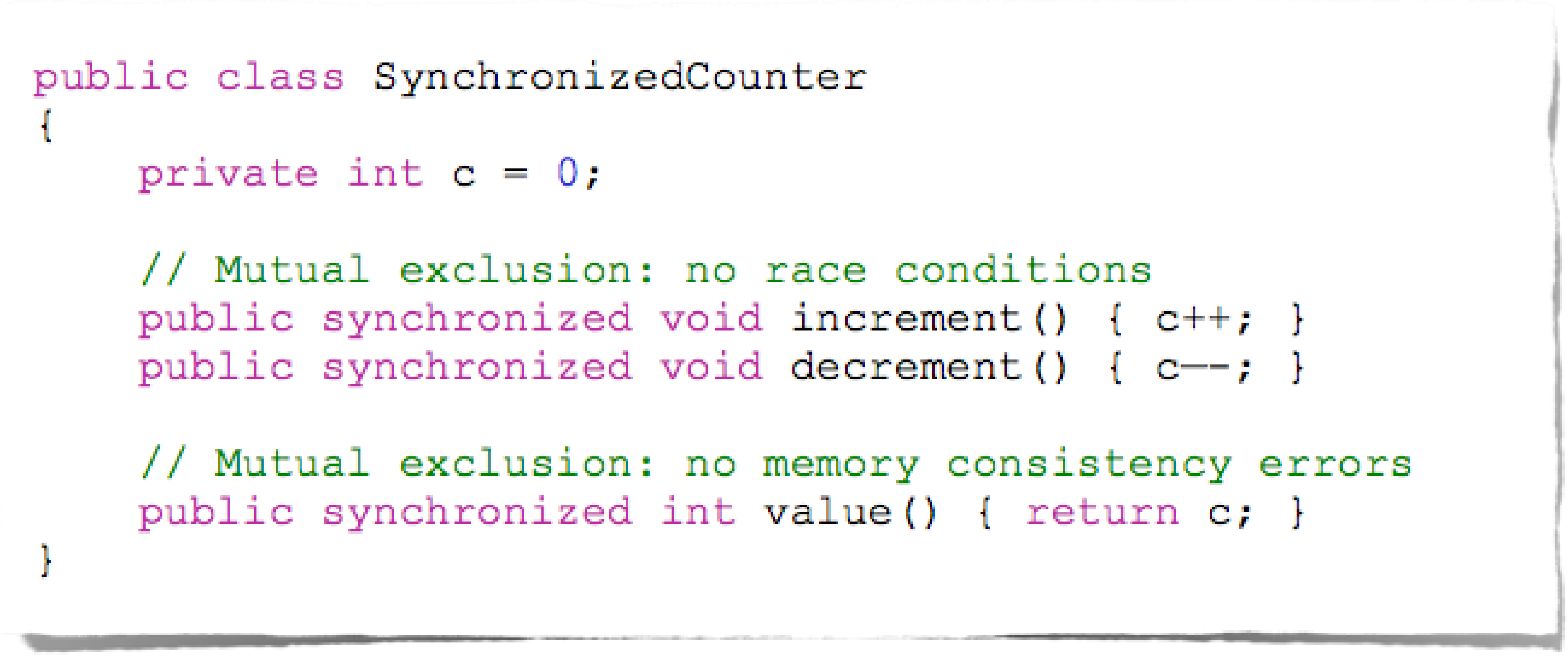
- **`void start()`**
  Causes the thread to begin execution
- **`void setPriority(int newPriority)`**
  Changes the priority of the thread
- **`static void sleep(long millis, int nanos)`**
  Causes the thread to pause execution for the specified number of milliseconds plus the specified number of nanoseconds
- **`public final void wait(long timeout)`** (inherited from `Object`)
  Causes the thread to wait until either another thread invokes the notify() method or a specified amount of time has elapsed
- **`public final void notify()`** (inherited from `Object`)
  Wakes up the thread
- **`void join()`**
  Causes the current thread to pause execution until the thread upon which `join()` has been invoked terminates. Overloads of `join()` allow to specify a waiting period

SYNCHRONIZED METHODS

- No two concurrent executions of `synchronized` methods on the same object are possible
- Mutual exclusion: invocations are serialized.
  The object behaves like it has a global lock which all its synchronized methods must acquire
  (indeed, it is exactly so)
- Constructors cannot be synchronized
  (does not make sense anyway)

EXAMPLE

- If an object is visible to more than one thread, all reads or writes to that object's variables can be done through synchronized methods to avoid some concurrency issues



```
public class SynchronizedCounter
{
    private int c = 0;

    // Mutual exclusion: no race conditions
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }

    // Mutual exclusion: no memory consistency errors
    public synchronized int value() { return c; }
}
```

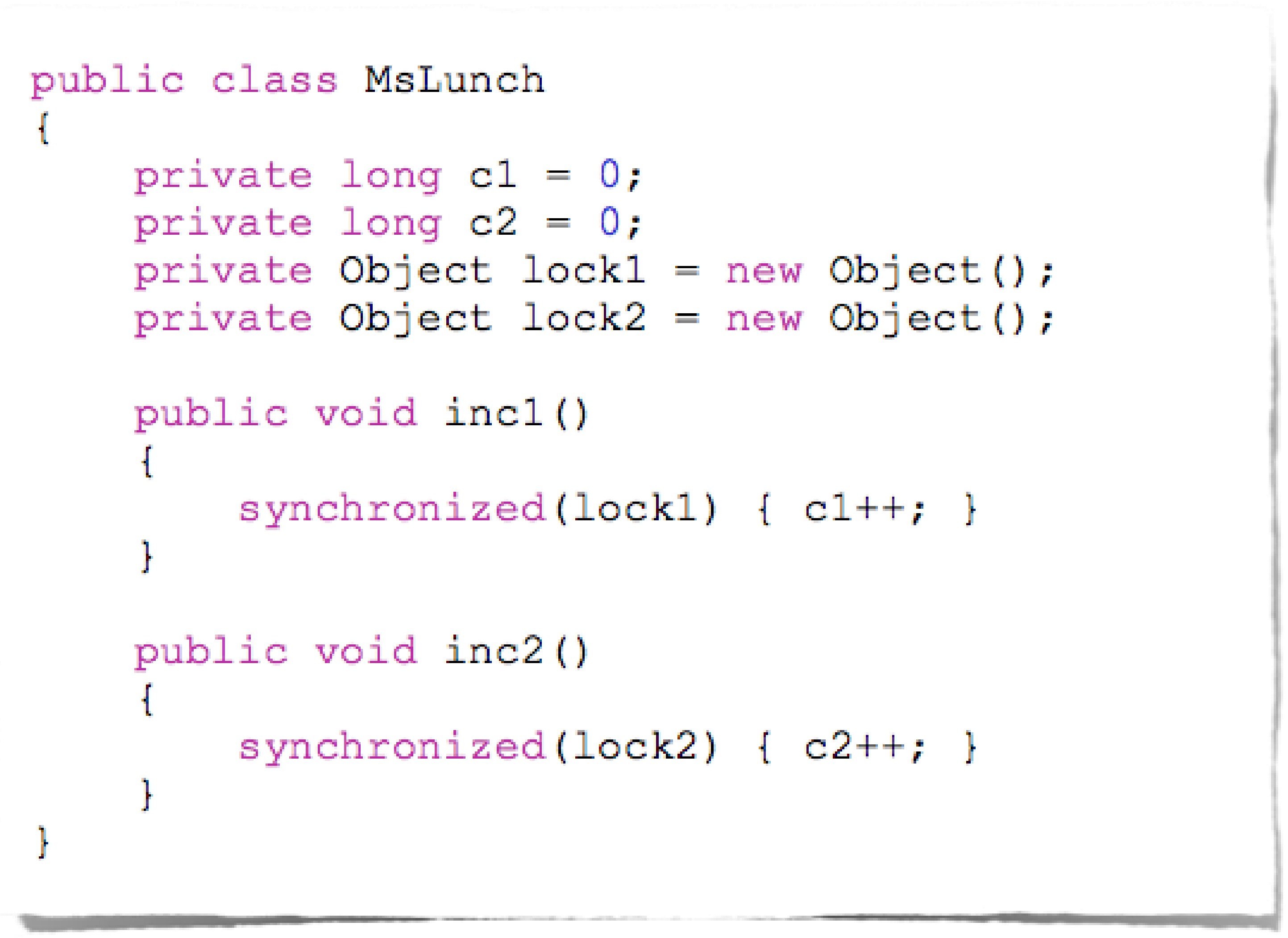
Code: docs.oracle.com

SYNCHRONIZED STATEMENTS

- Any statement, or group of statements, can be declared as `synchronized` by specifying the object that provides the lock
- All accesses to the statement(s) are serialized
- Improves concurrency: only a portion of a method is serialized

EXAMPLE

- In the following code, there is no reason to prevent interleaved updates of `c1` and `c2`



```java
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1()
    {
        synchronized(lock1) { c1++; }
    }

    public void inc2()
    {
        synchronized(lock2) { c2++; }
    }
}
```

Code: docs.oracle.com

JAVA: MORE ON CONCURRENCY

Look at the packages

- `java.util.concurrent`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`

CONCURRENCY: C++

- Concept of thread,
  associated with an instance of the class `Thread`.
  Every program has at least one thread
  and can create more

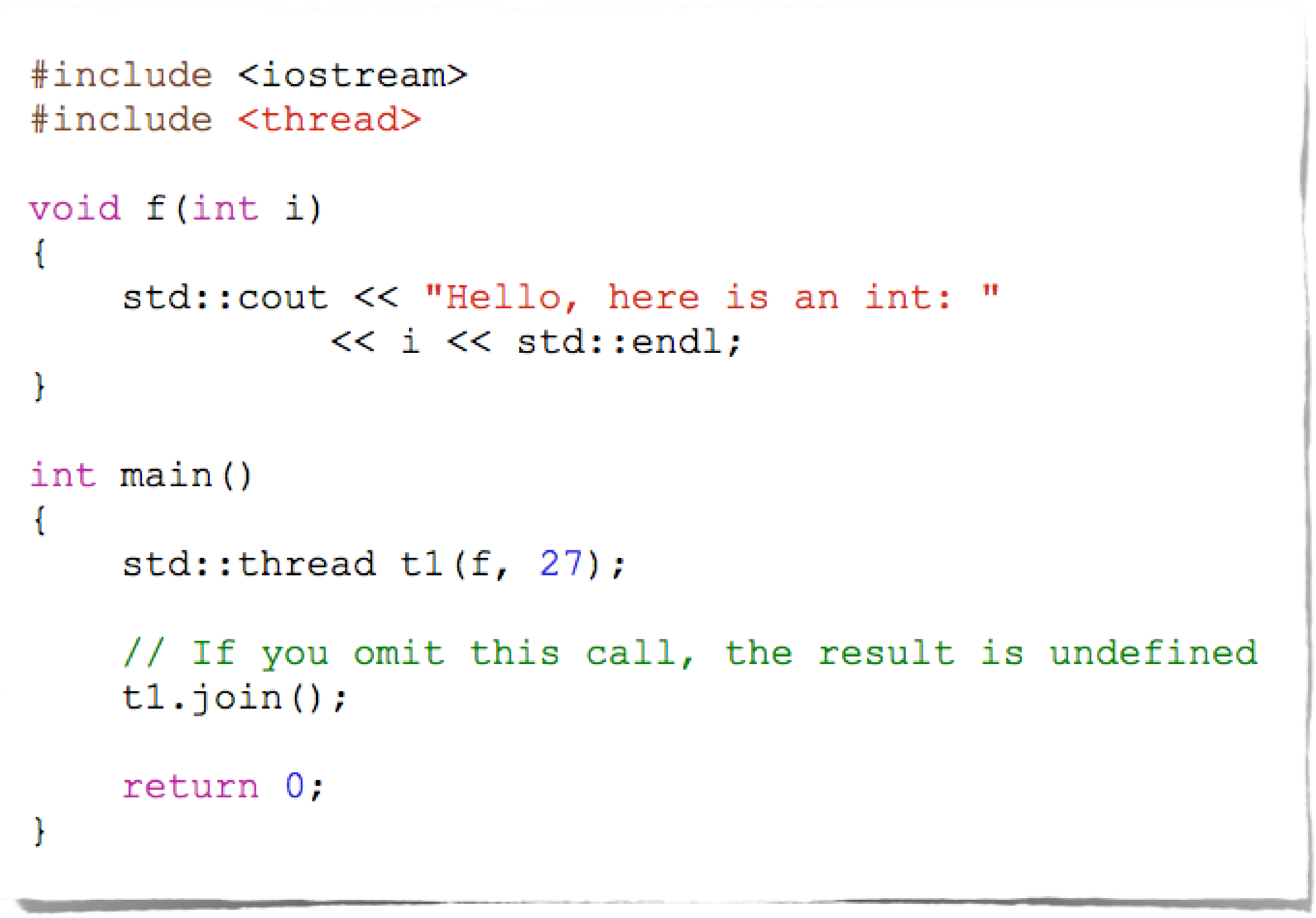
- Support for synchronization via the `join()` and `detach()` methods of the `Thread` class

- Support for mutually exclusive access to resources with **atomic types** and **mutex classes**

THREAD CLASS

- A thread starts immediately when an object is instantiated from the class `Thread`
- The code to be run is passed inside a function as a parameter to the constructor of `Thread`
- Further arguments for the constructor are passed as parameters to the function

EXAMPLE



```cpp
#include <iostream>
#include <thread>

void f(int i)
{
    std::cout << "Hello, here is an int: "
              << i << std::endl;
}

int main()
{
    std::thread t1(f, 27);

    // If you omit this call, the result is undefined
    t1.join();

    return 0;
}
```

THREAD CLASS: SOME METHODS

- **`bool joinable()`**
  Returns `true` if the thread object is *joinable*, i.e., it actually represents a thread of execution, and `false` otherwise

- **`id get_id()`**
  If the thread object is joinable, returns a value that uniquely identifies the thread

- **`void join()`**
  Causes the current thread to pause execution until the thread upon which `join()` has been invoked terminates

- **`void detach()`**
  Causes the current thread to be detached from the thread upon which `detach()` has been invoked

ATOMIC TYPES

- **Atomic types** are types that are guaranteed to be accessible without causing race conditions
- Some examples:

Atomic type	Contains
`atomic_bool`	`bool`
`atomic_char`	`char`
`atomic_int`	`int`
`atomic_uint`	`unsigned int`

MUTEXES

Allow mutually-exclusive access to critical sections of the source code

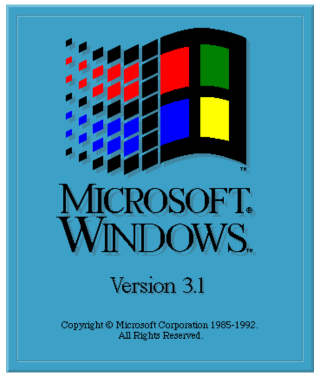
- **`mutex`** class
  Implements a binary semaphore. Does not support recursion (i.e., a thread shall not invoke the `lock()` or `try_lock()` methods on a mutex it already owns): use the `recursive_mutex` class for that

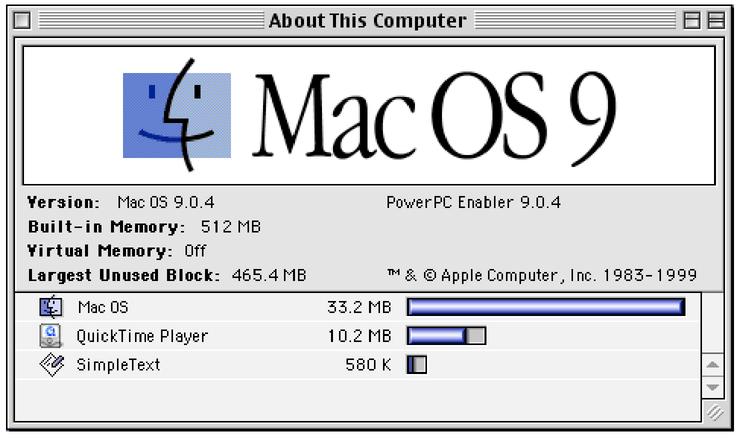
- **`timed_mutex`** class
  A mutex that additionally supports timed "try to acquire lock" requests (`try_lock_for(…)` method)

- **`void lock(…)`** function
  Locks all the objects passed as arguments, blocking the calling thread until all locks have been acquired

- **`bool try_lock(…)`** function
  Nonblocking variant of `lock(…)`. Returns `true` if the locks have been successfully acquired, `false` otherwise

MULTITASKING (2/3)





About This Computer

Mac OS 9

Version: Mac OS 9.0.4 PowerPC Enabler 9.0.4
Built-in Memory: 512 MB
Virtual Memory: Off
Largest Unused Block: 465.4 MB ™ & © Apple Computer, Inc. 1983-1999

Mac OS	33.2 MB
QuickTime Player	10.2 MB
SimpleText	580 K

- **Cooperative multitasking**
  SEs <u>voluntarily</u> release EUs.
  If SEs do not cooperate, the system malfunctions.
  Old versions of Mac OS and Windows worked this way

- **Preemptive multitasking**
  SEs are <u>forcibly</u> removed from EUs when the OS decides it is time to do so.
  SEs reliably receives a slice of execution time proportional to their importance

MULTITASKING (3/3)

- **Android, iOS, Windows Phone:** modern OSs with **full support for preemptive multitasking**
- Multiple SEs can execute concurrently...
  ...but, in general, multiple apps **can not**

LIMITATIONS

Android, iOS, Windows Phone:

- the one app in the **foreground** executes without need to ask permissions,
- apps in the **background** must take explicit action

LIMITATIONS: WHY? (1/2)

- Every **application uses up resources** (memory, energy if the app is running) **that are limited in embedded systems**
- If resources are needed
  - the foreground app cannot be affected because the user would immediately notice,
  - all other apps are expendable

LIMITATIONS: WHY? (2/2)

- Windows Mobile introduced the idea of politely ask background apps to close so as to reclaim resources
- Some apps needed more resources just to close themselves, causing a complete system lock-up
- Solution: “close” has been replaced by “**kill**”

HARD TIMES FOR EMBEDDED DEVELOPERS

© Nokia

- **The user doesn't care if apps in the background have limitations**
- Recall the UI model
    - The user knows nothing about paused/stopped/killed...
    - He sees all apps as "running" (available)
- **It is the developer's task to manually maintain the illusion of multitasking** if the application needs it (e.g., to load a web page in the background, to play music...)
- **The platforms help the developer** via suitable APIs

ANDROID: PROCESSES, THREADS

- When the first component of an app starts, the OS creates a new process (a Linux process) with **a single thread of execution**: Java's **main thread**
- By default, all components of the app will run in this very thread; execution requests are put in a queue
- The main thread is also the user interface thread ("**UI thread**") in Android

MULTIPLE PROCESSES

- Different components of an app can run in separate processes
- Components of different apps can run in the same process, provided such apps are signed with the same certificate
- In the manifest, set the **`android:process`** and **`android:multiprocess`** attributes of the app components appropriately

MULTIPLE THREADS

- Long operations in the main thread (e.g., retrieving data from a database or a network server) will block the whole UI
- **Do not block the main thread:**
  spawn additional threads (aka "worker threads")
- Use Java facilities (`Thread` class, etc.)
- Use additional classes provided by Android

THREADS AND UI TOOLKIT

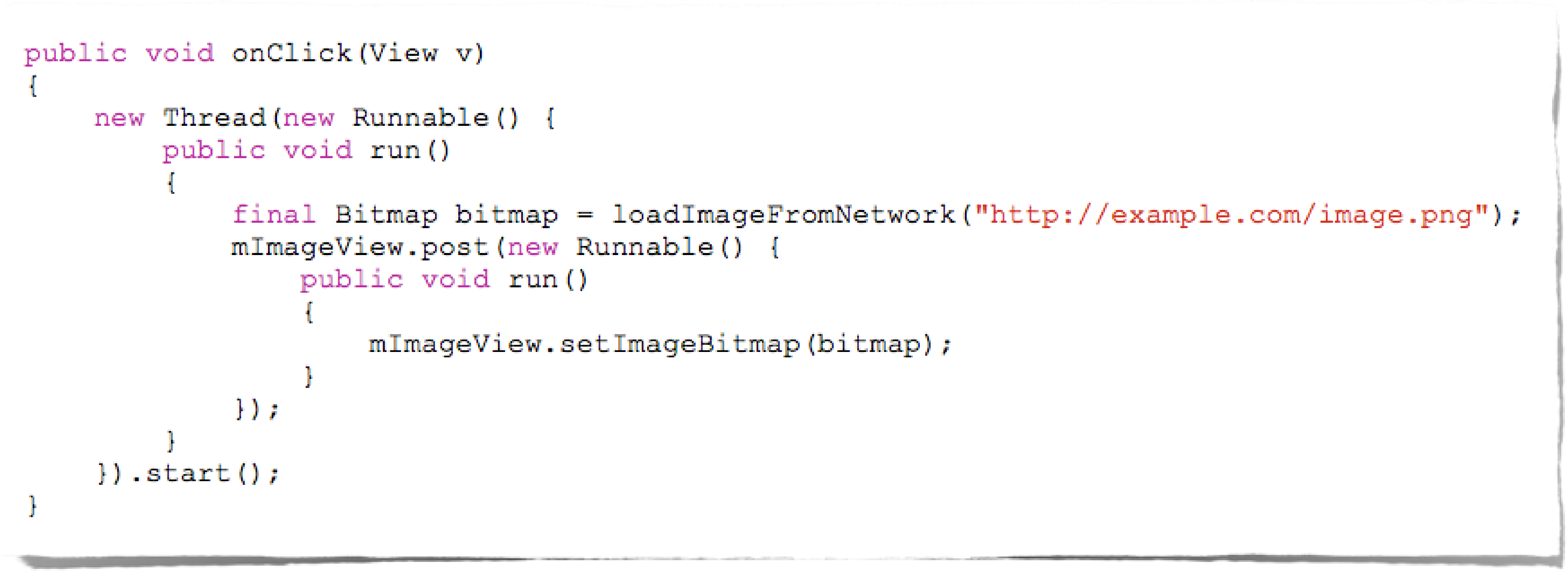
- **Do not access the Android UI toolkit from outside the main thread** (i.e., from worker threads): the Android UI toolkit is not thread safe

- **`Activity.runOnUiThread(Runnable)`**
  Posts the `Runnable` to the event queue of the main thread

- **`View.post(Runnable)`**
  `View.postDelayed(Runnable, long)`
  The `Runnable` is added to the message queue of the object, and executed later (after at least the specified delay, in the case of `postDelayed(...)`) in the main thread

UI TOOLKIT: EXAMPLE

- When a button is tapped, the following code downloads an image from a separate thread and displays it in an `ImageView`



```
public void onClick(View v)
{
    new Thread(new Runnable() {
        public void run()
        {
            final Bitmap bitmap = loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run()
                {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

Code: Google

HANDLER CLASS

- Receives messages and runs code to handle the messages (`handleMessage()` method)
- An new instance of `Handler` can be connected to an existing thread or can run in a new thread
- Connect an instance of `Handler` to the main thread to safely manage messages that imply changes to the UI

ASYNCTASK CLASS

- Allows to perform background operations and publish results on the main thread while hiding `Thread`s and `Handler`s

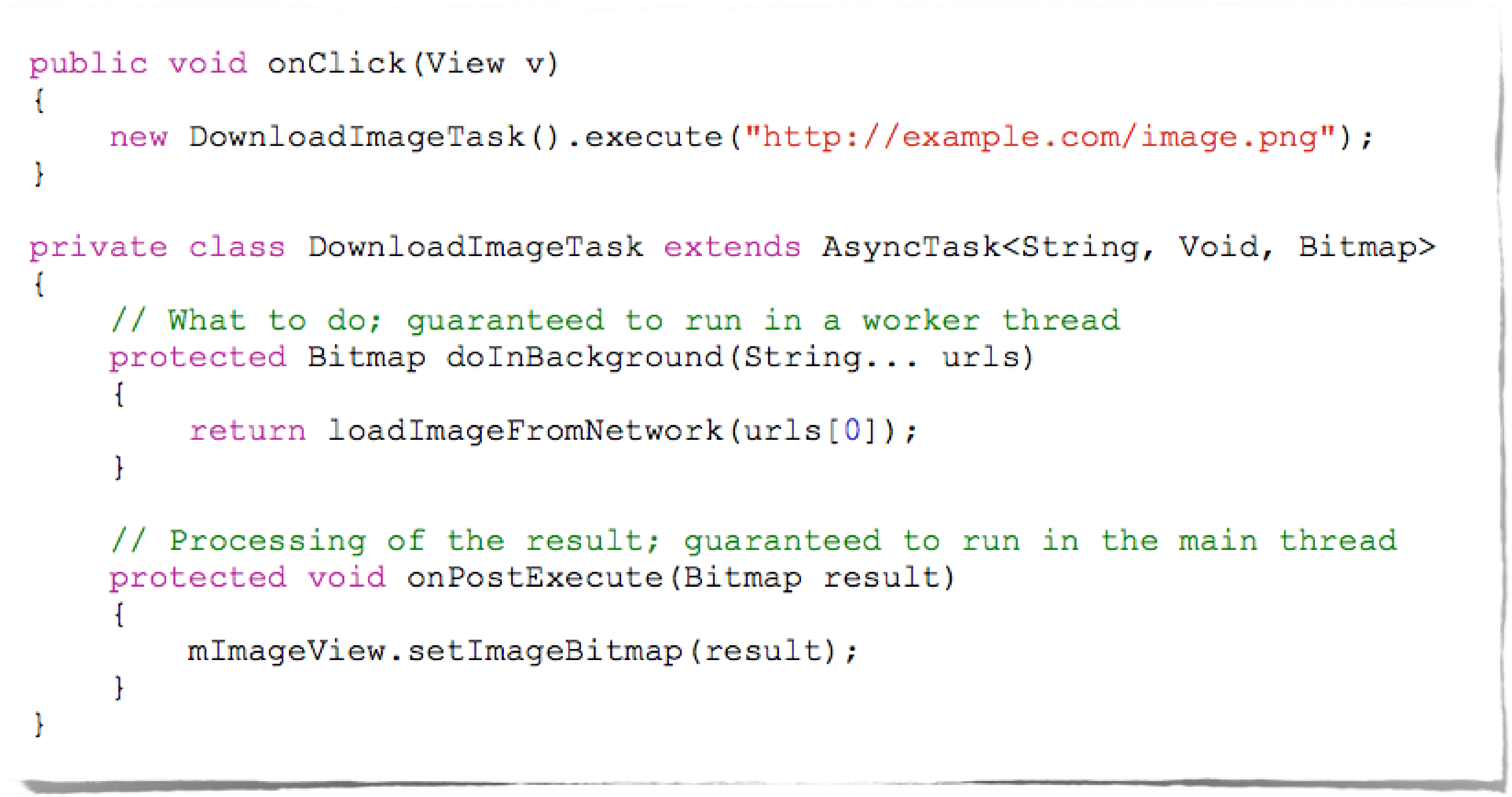
- Must be subclassed to be used

- Override the **`doInBackground(…)`** method to provide the code to be executed (compulsory)

- Override the **`onPostExecute(Result result)`** method, which receives the results from the background execution, to process them in the main thread

ASYNCTASK: EXAMPLE

- Downloading an image with the `AsyncTask` class



```
public void onClick(View v)
{
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap>
{
    // What to do; guaranteed to run in a worker thread
    protected Bitmap doInBackground(String... urls)
    {
        return loadImageFromNetwork(urls[0]);
    }

    // Processing of the result; guaranteed to run in the main thread
    protected void onPostExecute(Bitmap result)
    {
        mImageView.setImageBitmap(result);
    }
}
```

Code: Google

KILLING PROCESSES (1/2)

- The OS tries to maintain processes in memory for as long as possible, but might be forced to kill some of them when resources are low
- All app components running in a killed process are destroyed as well
- When deciding which processes to kill, the OS weighs their relative **importance to the user.** Therefore, **the choice depends on the state of the components running in the process**

KILLING PROCESSES (2/2)

- The OS maintains a **5-level "importance hierarchy"** based on the components running in processes
- Each process is assigned to the level of the most "important" component that runs in it
- A process' level might be increased because other processes are dependent on it
- When resources are needed,
  less important processes are killed first

PROCESS HIERARCHY (1/3)

- **Foreground process** (maximum importance)
  A process that is required for what the user is currently doing
  - It hosts an activity that the user is interacting with
  - It hosts a service that is bound to the activity that the user is interacting with
  - It hosts a service that has called `startForeground()`
  - It hosts a service that is executing one of its lifecycle callbacks
  - It hosts a broadcast receiver that is executing its `onReceive()` method

PROCESS HIERARCHY (2/3)

- **Visible process**
  A process that does not have any foreground components, but still can affect what the user is doing
  - It hosts an activity that is not in the foreground, but is still visible to the user (e.g., its `onPause()` method has been called; it started a modal dialog)
  - It hosts a service that is bound to a visible activity

PROCESS HIERARCHY (3/3)

- **Service process**
  A process that is running a service that does not fall into either of the two previous hierarchy levels
- **Background process**
  A process holding an activity that is not currently visible to the user.
  Processes in this hierarchy level are kept in an LRU list, so that the process with the activity that was most recently seen by the user is the last to be killed
- **Empty process** (minimum importance)
  A process that does not hold any active application components

KILLING THREADS

- Runtime configuration changes (e.g., screen rotation) kill worker threads even if they belong to a high-priority process
- Two possible solutions

1. properly shut down and restart threads
2. ask the OS not to kill the threads
   by **retaining** the activity and/or its fragments

SCHEDULING PROCESSES

- Android processes in the foreground (belonging to how many apps?) are scheduled regularly
- **Other processes** are scheduled too
  but **may be stopped or killed at any time**
  without further notice
- Consequence: <u>activities</u> in the background cannot be trusted to complete any job

ACTIVITY LIFECYCLE

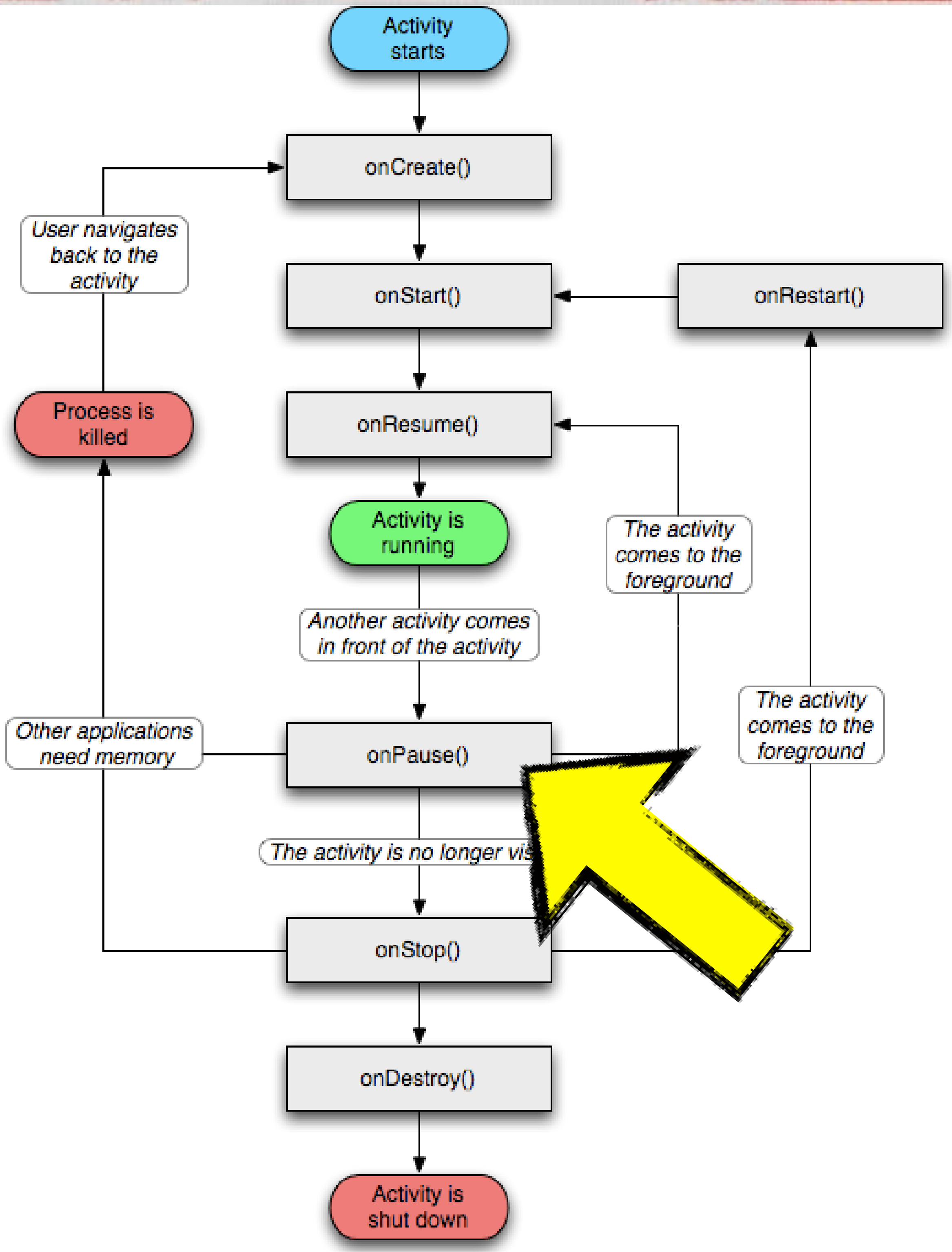

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer vis
onStop()
onDestroy()
Activity is shut down

Image from developer.android.com

A NOTE ON IOS

- Only <u>one</u> app in the foreground
- **Apps in the background**
  - iOS≤3: simply not allowed
    Apps are closed when they leave the foreground
  - iOS≥4: simply **not scheduled**
    Apps still in RAM but suspended.
    Can be killed at any time
- Apps in the background cannot be trusted to complete any job, but it is easier to determine when they are scheduled

BACKGROUND EXECUTION IN ANDROID (1/4)

- **Services**
  Allow an app to run in the background for a long period of time (several minutes)

A process running a service is ranked higher than a process with background activities, hence an activity that initiates a long-running operation might do well to start a service for it, rather than creating a thread—particularly if the operation will likely outlast the activity

BACKGROUND EXECUTION IN ANDROID (2/4)

- **Broadcast receivers**
  Allow an application to run in the background for a brief amount of time as a result of an external event

The time limit for broadcast receivers is currently 10 seconds, so background receivers should consider employing services as well

BACKGROUND EXECUTION IN ANDROID (3/4)

- **Content Providers**
  Encapsulate structured sets of data, and provide access to them

Content providers are the standard interface that connects data in one process with code running in another process.

Querying a content provider for data takes time.
If the query is run from an activity, the UI may slow down and/or the activity may get blocked.
Consider initiating the query on a separate thread

BACKGROUND EXECUTION: FURTHER LIMITS

- From Android 8.0 (API level 26), there are further limits on what apps can do in the background
- **Services**: after some minutes in the background, an app is considered to be **idle** and the the system stops the app's background services
- **Broadcast receivers**: with limited exceptions, apps cannot register for implicit broadcasts in the manifest

BACKGROUND EXECUTION IN ANDROID (4/4)

- **JobScheduler** class (API level ≥21)
  Allows to define a unit of work (“**job**”)
  to be scheduled asynchronously in the background

A job definition includes requirements for network connectivity and timing

The scheduler attempts to **batch and defer jobs as much as possible** to optimize the use of resources

TO LEARN MORE

- Keeping Your App Responsive
- Sending Operations to Multiple Threads
- Best Practices for Background Jobs
- Background Execution Limits
- SMP Primer for Android