Android
Java Threads

What is a thread?

- A thread is an independent path of execution through program code
- Threads can be managed independently by a scheduler, which is typically a part of the operating system
- Multiple threads can exist within the same process and share resources such as memory
- On a multiprocessor or multi-core system, threads can be executed in a true concurrent manner, with every processor or core executing a separate thread simultaneously

Multithreading Advantages

- Responsiveness
- Faster execution
- Lower resource consumption
- Better system utilization
- Parallelization

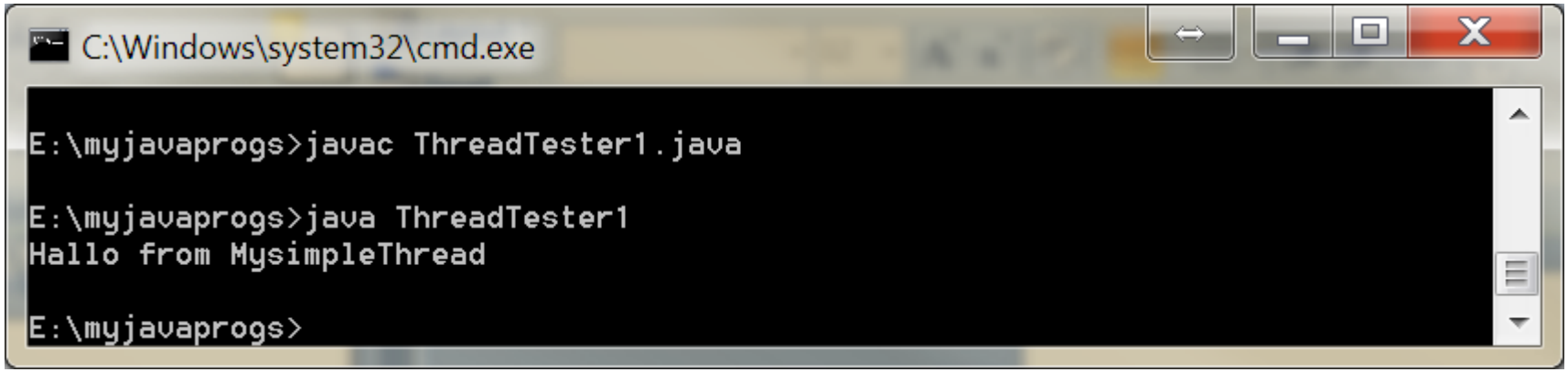
CREATING AND USING THREADS

Extending the Thread class

```
class MysimpleThread extends Thread
{
    public void run ()
    {
        System.out.println ("Hallo from MysimpleThread");
    }
}
```

Using our thread

```
public class ThreadTester1
{
    public static void main (String [] args)
    {
        MysimpleThread mt = new MysimpleThread();
        mt.start ();
    }
}
```

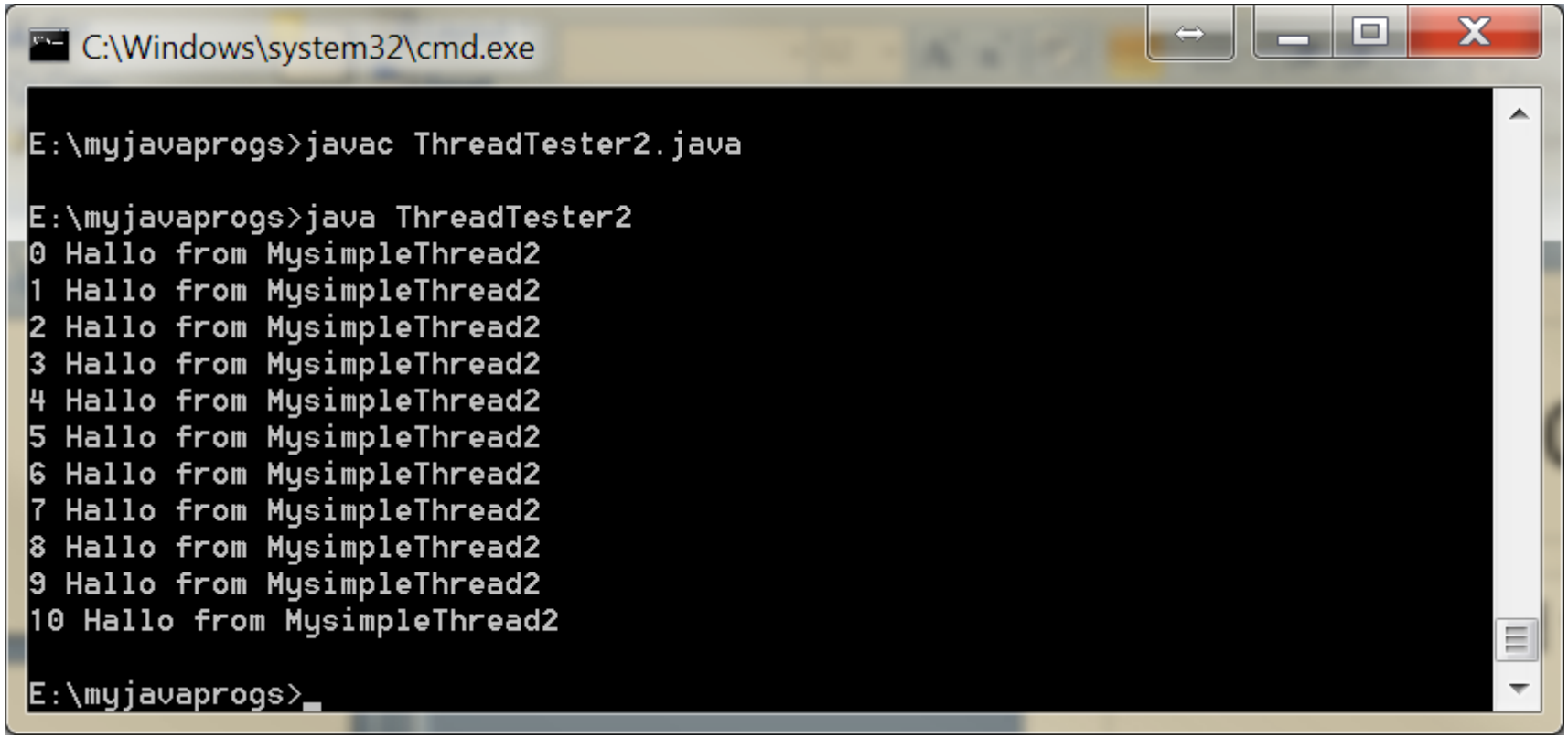

C:\Windows\system32\cmd.exe
E:\myjavaprogs>javac ThreadTester1.java
E:\myjavaprogs>java ThreadTester1
Hallo from MysimpleThread
E:\myjavaprogs>

Example 1/2

```
class MysimpleThread2 extends Thread
{
    public void run ()
    {
        for (int i=0;i<=10;i++)
        {
                    try
                     {
                         Thread.sleep (1000); // Sleep for 1 second
                     }
                     catch (InterruptedException e)
                     {
                     }
                System.out.println (i+" Hallo from MysimpleThread2");
        }
    }
}
```

Example 2/2

```
public class ThreadTester2
{
    public static void main (String [] args)
    {
        MysimpleThread2 mt2 = new MysimpleThread2();
        mt2.start ();
    }
}
```


C:\Windows\system32\cmd.exe
E:\myjavaprogs>javac ThreadTester2.java
E:\myjavaprogs>java ThreadTester2
0 Hallo from MysimpleThread2
1 Hallo from MysimpleThread2
2 Hallo from MysimpleThread2
3 Hallo from MysimpleThread2
4 Hallo from MysimpleThread2
5 Hallo from MysimpleThread2
6 Hallo from MysimpleThread2
7 Hallo from MysimpleThread2
8 Hallo from MysimpleThread2
9 Hallo from MysimpleThread2
10 Hallo from MysimpleThread2
E:\myjavaprogs>

Implementing the Runnable Interface

```
public class MyRunnable implements Runnable {

    public void run(){
        System.out.println("Hallo from runnable");
    }
}
```

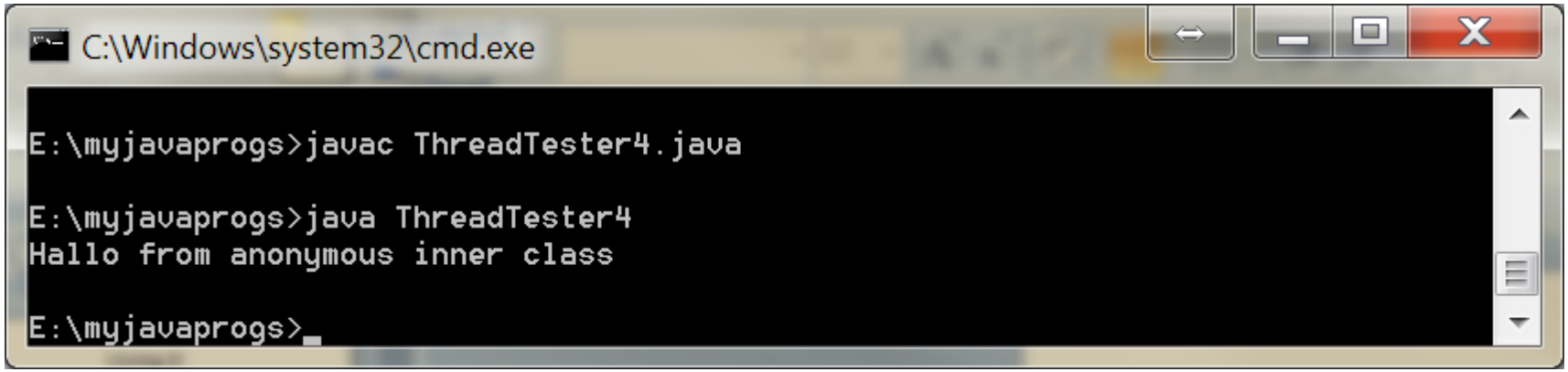
Using it

```
public class ThreadTester3
{
    public static void main (String [] args)
    {
        Thread mt2 = new Thread(new MyRunnable());
        mt2.start ();
    }
}
```


C:\Windows\system32\cmd.exe
E:\myjavaprogs>javac ThreadTester3.java
E:\myjavaprogs>java ThreadTester3
Hallo from runnable
E:\myjavaprogs>

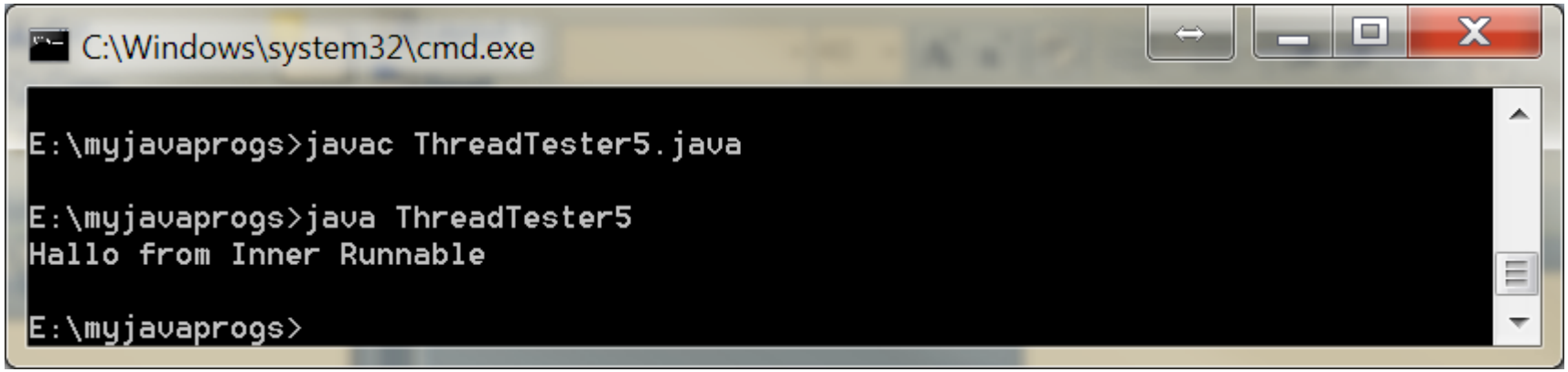
Through an anonymous class


```
public class ThreadTester4
{
    public static void main (String [] args)
    {
        Thread t = new Thread() {
            public void run() {
                System.out.println("Hallo from anonymous inner class");
            }
        };
        t.start();
    }
}
```


C:\Windows\system32\cmd.exe
E:\myjavaprogs>javac ThreadTester4.java
E:\myjavaprogs>java ThreadTester4
Hallo from anonymous inner class
E:\myjavaprogs>

Through anonymous inner class that implements runnable interface


```
public class ThreadTester5
{
    public static void main (String [] args)
    {
        Runnable myRunnable = new Runnable(){
            public void run(){
                System.out.println("Hallo from Inner Runnable");
            }
        };
        Thread t = new Thread(myRunnable);
        t.start();
    }
}
```


C:\Windows\system32\cmd.exe
E:\myjavaprogs>javac ThreadTester5.java
E:\myjavaprogs>java ThreadTester5
Hallo from Inner Runnable
E:\myjavaprogs>

Pausing Thread Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system
- Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond

Thread.sleep()

- Thread.sleep can throw an InterruptedException which is a checked exception
- All checked exceptions must either be caught and handled or else you must declare that your method can throw it
- Not declaring a checked exception that your method can throw is a compile error

Thread.sleep and InterruptedException

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
    // handle the exception...
    // For example consider calling Thread.currentThread().interrupt(); here.
}
```

Or declare that your method can throw an `InterruptedException`:

```
public static void main(String[]args) throws InterruptedException
```

Joining Threads

- Waiting for threads to finish their work is quite useful in many cases
- Because the while loop/isAlive() method/sleep() method technique proves useful, it is packaged into some methods:
  - join(), join(long millis), and join(long millis, int nanos).

join()

- The current thread calls join(), via another thread's thread object reference when it wants to wait for that other thread to terminate
- The current thread calls join(long millis) or join(long millis, int nanos) when it wants to either wait for that other thread to terminate or wait until a combination of millis millseconds and nanos nanoseconds passes

User Threads Vs Daemon Threads

- A *user thread* performs important work for the program's user, that must finish before the application terminates
- A *daemon thread* performs “housekeeping” and other background tasks that probably do not contribute to the application's main work but are necessary for the application to continue its main work
- Unlike user threads, daemon threads do not need to finish before the application terminates