parallel concurrent

Amdahl's Law thread of control

processors versus processes

non-deterministic fork-join parallelism.

Parallel and concurrent programming
3. Multithreading in Java

protection data race synchronization

divide-and-conquer algorithms

THREAD

SAFETY correctness

Michelle Kuttel

MUTUAL EXCLUSION

locks

readers-writers problem

liveness

DEADLOCK starvation

HIGH PERFORMANCE COMPUTING

producer-consumer problem

EXECUTORS thread pools timing

Dining philosphoers problem

Overview: How to write parallel programs in Java with the shared memory model

To write a parallel program, you (the programmer) need new primitives from a programming language or library, that enable you to:

- run multiple operations at once
  - Java has concurrent **threads**
- share data between operations
  - Java uses **shared memory** which all threads can access
- coordinate *(a.k.a. synchronize)* operations
  - Java has a range **of synchronization primitives**, as well as **thread-safe** and **concurrent classes**.
  - We will start with **a simple primitive** for threads to **wait for each other.**

    (Will cover more when we do concurrency)

Java concurrency

Java provides both

- **basic concurrency** support in the Java programming language and the Java class libraries

- **high-level concurrency** APIs

  `java.util.concurrent` package

Java is *always multithreaded*

The Java Virtual Machine

- executes **as a process** under the operating system
- supports **multiple threads**.

- In Java, every program has more than one thread
  - start with just one thread, called the *main thread*. This thread can create additional threads.
  - System threads perform **garbage collection** and **signal handling** (e.g. input from mouse and keyboard and play audio)
  - Threads compile Java byecode into machine-level instructions
  - standard libraries use threads

Basic Threads in Java

All threads are associated with an instance of the class **`java.lang.Thread`**.

Two **basic strategies** for using Thread objects :

- **instantiate Thread** each time the application needs to initiate an asynchronous task
  - common approach for **concurrent applications**

or

- pass the application's tasks to an **executor** (`java.util.concurrent`) that will launch and manage threads
  - Approach **for parallelisation** – high-level API that manages a thread pool for large-scale applications that run on multiprocessor and multi-core systems.

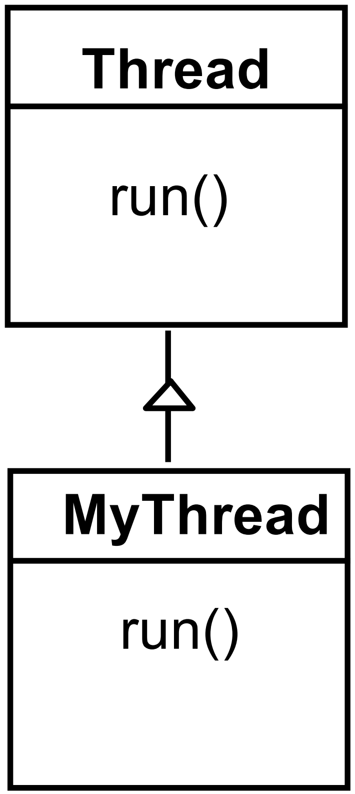
We will discuss **Thread objects first**

and then the **fork/join framework,** an implementation of the **ExecutorService interface** that helps you take advantage of multiple processors.

Thread Class in Java

A Thread class manages a **single** sequential **thread** of control. Threads may be **created** and **deleted dynamically**.

Thread class executes instructions from its method **run()**. The actual code executed depends on the implementation provided for run() in a derived class.




```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

Creating a thread object:

```
Thread a = new MyThread();
```

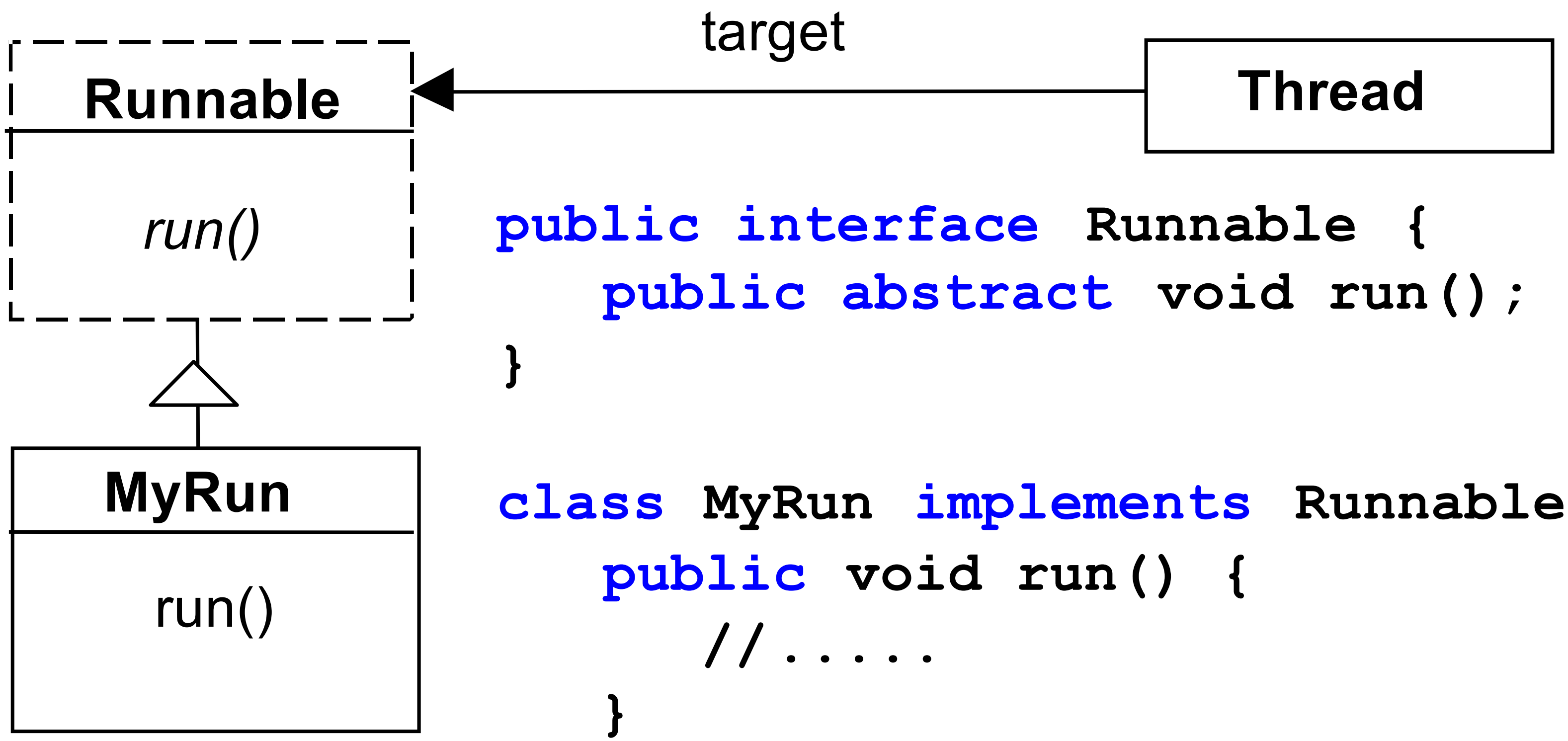
Runnable interface in Java

Since Java does not permit **multiple inheritance**,

it is sometimes more convenient to implement the **run()** method in a class not derived from Thread, but from the interface Runnable.

Runnable interface in Java

*The Runnable interface defines a **single method**, **run**, containing the code executed in the thread.*




```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

Creating a thread object:

```
Thread b = new Thread(new MyRun());
```

The Runnable object is passed to the Thread constructor.

Basic Threads in Java

So, there are two ways to create **a basic thread** in Java:

- Extend/subclass the Thread class (`java.lang.Thread`)

  Easy for simple programs

- Implement the Runnable interface in an object and pass that to a thread's constructor (`java.lang.Runnable`)

  More general/flexible approach

Hello Cruel Thread World

```
public class HelloThread extends java.lang.Thread {
    private int i;
    HelloThread(int i) { this.i = i; }

    public void run() {
        System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");
    }

    public static void main(String[] args) {
        for(int i=1; i <= 10; ++i) {
            HelloThread c = new HelloThread(i);
            c.start();
        }


}
```

Constructor

This is what each thread does.

This is what makes the thread run

When this program runs, it will print 10 lines of output, one of which is:

```
Thread 10 says hi
```

Using Java Threads

Allocation and construction of a Thread object **do not** cause the thread to **run.**

To get a new thread running:

1. Define a subclass `C` of **`java.lang.Thread`**, overriding **`run`**
2. Create an object of class `C`
3. Call that object's **`start`** method
   - Not **`run`**, which would just be a normal method call
   - **`start`** sets off a new thread, using **`run`** as its "main"

Using Java Threads

Allocation and construction of a Thread object **do not** cause the thread to **run.**

To get a new thread running:

1. Define a subclass `C` of **`java.lang.Thread`**, overriding **`run`**
2. Create an object of class `C`
3. Call that object's **`start`** method
    - Not **`run`**, which would just be a normal method call
    - **`start`** sets off a new thread, using **`run`** as its "main"

*What if we instead called the **`run`** method of `C`?*

- This would just be a normal method call, in the current thread

A question you may have…

We might also wonder if two lines of output would ever be mixed, something like:

```
Thread 13 Thread says 14 says hi hi
```

This is really a question of how the `System.out.println()` method handles concurrency

`System.out.println()` is **atomic**

[a **synchronized method** **-** more about this later]

so will *always keep a line of output together*

• (no other thread can interrupt).

Hello World, this time using *Runnable interface*

```
public class HelloInterface implements Runnable {
	private int i;
	HelloInterface(int i) { this.i = i; }

	public void run() {
		System.out.println("Thread " + i + " says hi");
		System.out.println("Thread " + i + " says bye");
	}

	public static void main(String[] args) {
		for(int i=1; i <= 10; ++i) {
			Thread c = new Thread(new HelloInterface(i));
			c.start();
		}
	}
}
```

Example of HelloWord with shared object (and lots of race conditions)

```java
public class HelloThread extends java.lang.Thread {
	private int i;
	private static String sharedString;
	HelloThread(int i) { this.i = i; }
	public void run() {
		System.out.println("Thread " + i + " says hi");
		sharedString="Thread " + i + " was here!";
		System.out.println("Thread " + i + " says bye");
	}
	public static void main(String[] args) {
		sharedString = "main thread string\n";
		for(int i=1; i <= 10; ++i) {
			HelloThread c = new HelloThread(i);
			c.start();
		}
		System.out.println(sharedString);
}}
```

Checkpoint: start versus run

```java
public class HelloThread extends java.lang.Thread {
    private int i;
    HelloThread(int i) { this.i = i; }
    public void run() {
    System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");
    }

    public static void main(String[] args) {
        for(int i=1; i <= 5; ++i) {
            HelloThread c = new HelloThread(i);
            c.start();
        }
    }
}
```

```java
public class HelloThread extends java.lang.Thread {
    private int i;
    HelloThread(int i) { this.i = i; }
    public void run() {
    System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");
    }

    public static void main(String[] args) {
        for(int i=1; i <= 5; ++i) {
            HelloThread c = new HelloThread(i);
            c.run();
        }
    }
}
```

What will be the effect on the **runtime** and **output** of changing c.start() to c.run()?

Checkpoint: write down all possible output of this code

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println(”B"); }
}
public class STest {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
        }
}
```

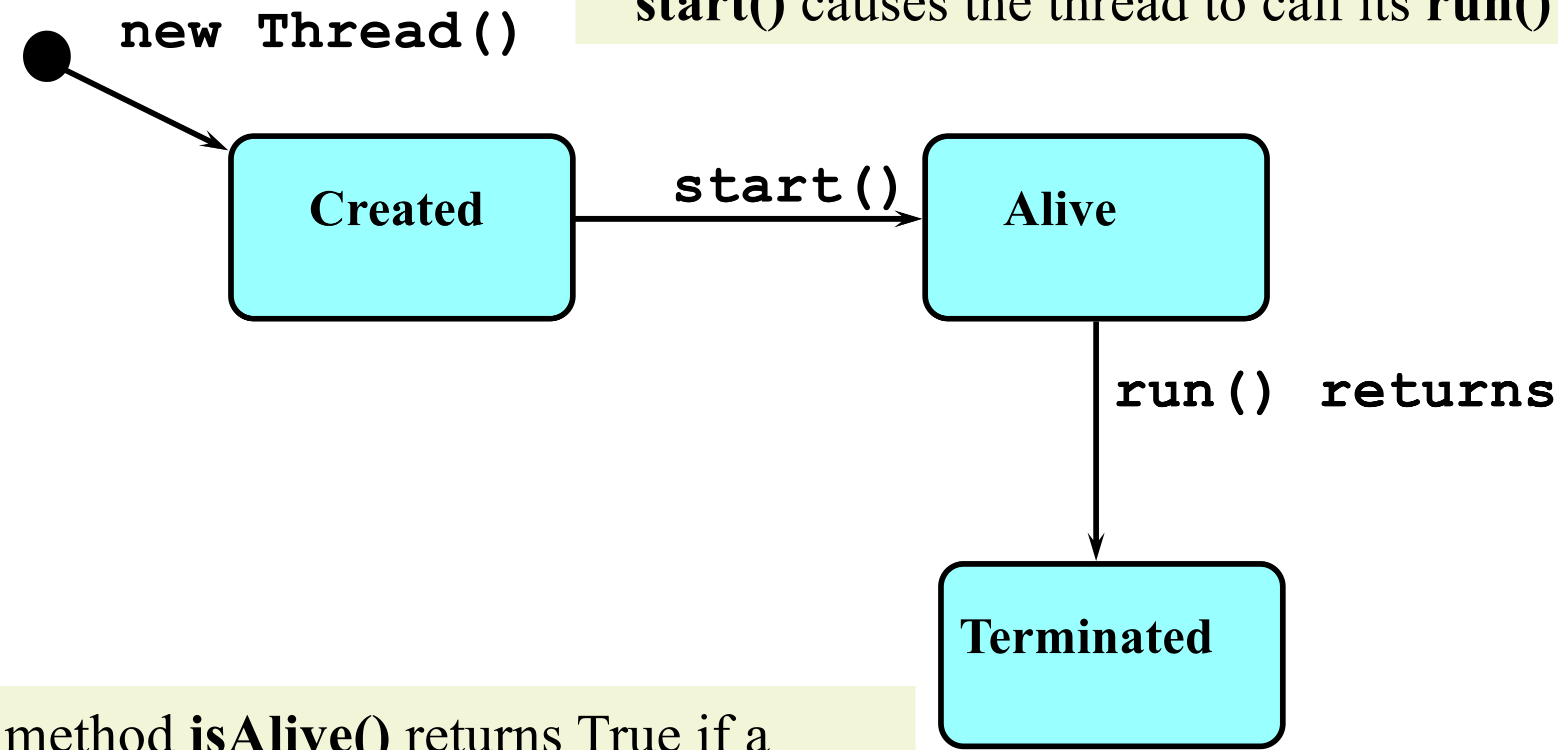
Checkpoint: write down all possible output of this code

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println(”B"); }
}
public class STest {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
                System.out.println("C");
        }
}
```

Thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

start() causes the thread to call its **run()** method.



The method **isAlive()** returns True if a thread has been started but not terminated.

Once terminated, **it cannot be restarted.**

Thread class: Commonly used methods

- `start()` – starts the thread.
- `sleep(long millis)` – pause the thread for the specified time (milliseconds).
- `yield()` – hint to the scheduler that the current thread is willing to yield its current use of a processor
- `join()` – wait the current thread until the thread called has terminated

Example code

public class HelloThreadMethods **extends** java.lang.Thread

Showing the use of these methods

- `start()`
- `sleep(long millis)`
- `yield()`
- `join()`

```java
public class HelloThreadMethods extends java.lang.Thread {
    private int i;
    private boolean sleepy;
    private boolean polite;

    HelloThreadMethods(int i) {  this(i, false, false); }

    HelloThreadMethods(int i, boolean slpy, boolean plt) {
        this.i = i;
        sleepy=slpy;
        polite=plt;
        }
    public void run() {
        System.out.println("Thread " + i + " says hi");
        if(polite) yield(); //thread more likely to finish last
        if(sleepy)
            try {
                System.out.println("Thread " + i + " snoozing");
                sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        System.out.println("Thread " + i + " says bye");
    }
    public static void main(String[] args) throws InterruptedException {
        int noThrds=10;
        HelloThreadMethods [] thrds = new HelloThreadMethods[noThrds];
        for(int i=0; i < noThrds; ++i) {

            if (i==0)  thrds[i] = new HelloThreadMethods(i,true,false); //first thread is slow
            else if(i==(noThrds-1)) thrds[i] = new HelloThreadMethods(i,false,true); //last thread is polite
            else  thrds[i] = new HelloThreadMethods(i);
            thrds[i].start();
        }
        for(int i=0; i < noThrds; ++i) {
            thrds[i].join(); //main thread waits for HelloThread i
        }
        System.out.println("we are all done");
    }}
```

Checkpoint: write down all possible outputs of this code

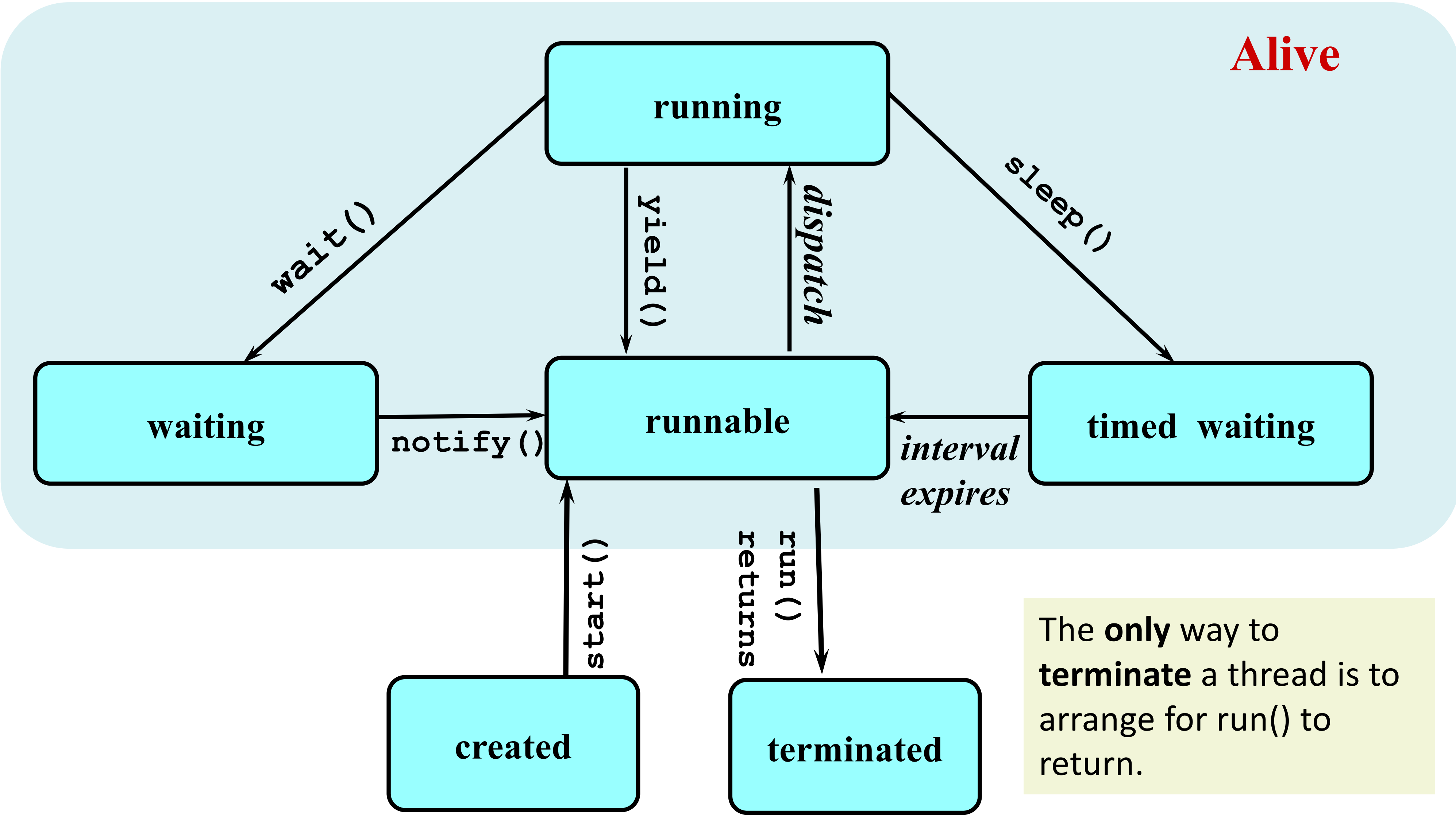
```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }}
public class BThread extends Thread {
        public void run() {System.out.println(”B"); }}
public class STest throws InterruptedException {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
                threadB.join();
                System.out.println("C");}}
```

Checkpoint: write down all possible output of this code

```
public class AThread extends Thread {
        public void run() {System.out.println("A"); }
}
public class BThread extends Thread {
        public void run() {System.out.println(”B"); }
}
public class STest throws InterruptedException {
        public static void main(String[] args) {
                AThread threadA = new AThread();
                BThread threadB = new BThread();
                threadA.start();
                threadB.start();
                threadA.join();
                System.out.println("C");
        }
```

Thread alive states in Java

Once started, an **alive** thread has a number of substates :



The **only** way to **terminate** a thread is to arrange for run() to return.

Thread class: method to avoid

- `setPriority()` – set priority of thread (1-10,bigher is higher). Only a **hint** to the scheduler.

“Avoid the temptation to use thread priorities, since they increase platform dependence and can cause liveness problems. Most concurrent applications can use the default priority for all threads.”

Java Concurrency in Practice

Joshua Bloch, Brian Goetz, Tim Peierls, Joseph Bowbeer, David Holmes, Doug Lea

Although thread priorities exist in Java and many references state that the JVM will always select one of the highest priority threads for scheduling, this is **not guaranteed** by the Java language or virtual machine specifications. **Priorities are only hints to the scheduler**.

Stopping a thread

Most of the time we allow threads to stop by running to completion

Sometimes we want to stop threads sooner, e.g. when

- user cancels operation
- application needs to shutdown quickly

- Not easy to get threads to stop safely, quickly and reliably
  - Thread.stop and Thread.suspend were an attempt at doing this
  - deprecated ages ago, as too dangerous
  - Java **does not now provide any mechanism for forcing a thread to stop**
    - instead, **ask** the thread to stop what it is doing with an `interrupt`

Java Threads: java.util.concurrent

`java.util.concurrent`

- an extensive library of **utility classes** useful in concurrent and parallel programming:
  - **Executors**, queues, timing, synchronizers, concurrent collections
- Makes threading simpler, easier and less-error prone.

Java Executors for parallel programming with lots of threads

In **large-scale parallel** applications,

it makes sense to **separate** thread **creation** and **management** from the rest of the application.

Executor objects in Java **distribute tasks to** worker threads in **a thread pool.**

Thread Pools reduce thread creation **overhead**:

- allocating and deallocating many thread objects requires significant memory management.

Java Executors for parallel programming

There are three Executor interfaces in Java. We focus on **Fork/Join.**

- Executor, basic support for **launching new tasks**.
  - **ExecutorService** (subinterface of Executor), adds features to **manage** tasks.
    - **Fork/Join** (implementation of ExecutorService)
      - designed for **divide-and-conquer algorithms.**
      - *work-stealing* algorithm: idle threads can steal tasks from busy threads.
      - `ForkJoinPool` is the main thread pool class
      - Two types of task sent to pool
        - `RecursiveAction` has no return value
        - `RecursiveTask` returns a value

    - **ScheduledExecutorService**, (subinterface of ExecutorService) enables future and/or periodic execution of tasks.

Hello World, with *Fork-Join* (and a race condition…)

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

// RecursiveAction has no return value, RecursiveTask returns a value
public class HelloMany extends RecursiveAction  {
     int greetings; // arguments
     int offset;

     HelloMany(int g, int start) {
       greetings=g;
       offset=start;
     }

     protected void compute(){
        if((greetings) <=1) { //only one task left, do it. This cutoff would be bigger for proper programs
           System.out.println("hello"+offset );
        }
         else {
               int split=(int) (greetings/2.0);
               //split work into two
               HelloMany left = new HelloMany(split,offset);  //first half
               HelloMany right= new HelloMany(greetings-split,offset+split ); //second half
               left.fork(); //give first half to new thread
               right.compute(); //do second half in this thread
           }
     }
     public static void main(String[] args) {
         HelloMany sayhello = new HelloMany(50,0); //the task to be done, divide and conquer
         ForkJoinPool pool  = new ForkJoinPool(); //the pool of worker threads
         pool.invoke(sayhello); //start everything running - give the task to the pool
      }
}
```

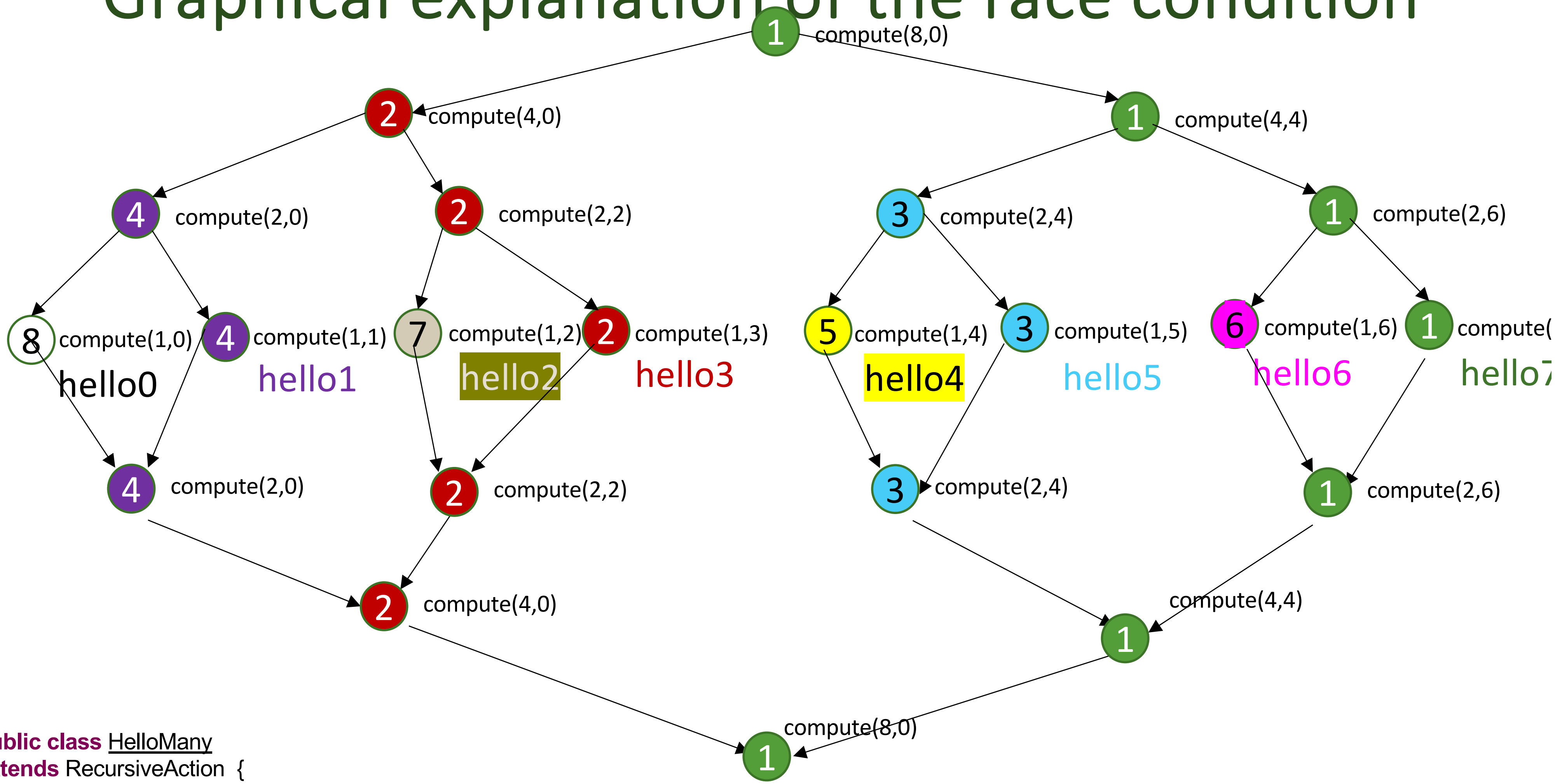
Hello World, with *Fork-Join* (race condition removed)

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

// RecursiveAction has no return value, RecursiveTask returns a value
public class HelloMany extends RecursiveAction  {
      int greetings; // arguments
      int offset;

      HelloMany(int g, int start) {
        greetings=g;
        offset=start;
      }
      protected void compute(){
         if((greetings) <=1) { //only one task left, do it. This cutoff would be bigger for proper programs
            System.out.println("hello"+offset );
         }
          else {
                int split=(int) (greetings/2.0);
                //split work into two
                HelloMany left = new HelloMany(split,offset);  //first half
                HelloMany right= new HelloMany(greetings-split,offset+split ); //second half
                left.fork(); //give first half to new threas
                //left.join(); // what will this do if included?
                right.compute(); //do second half in this thread
                left.join(); //corrected!!
          } }
      public static void main(String[] args) {
          HelloMany sayhello = new HelloMany(50,0); //the task to be done, divide and conquer
          ForkJoinPool pool  = new ForkJoinPool(); //the pool of worker threads
          pool.invoke(sayhello); //start everything running - give the task to the pool
       }
      }
```

Aside: So why didn't we see this before with our first version of Hello World?

- Every Java application starts up a **main** thread that runs a **main()** method.
- Java runtime environment distinguishes between **user thread** and **daemon threads**.
  - As long as a **user thread** is alive, the JVM does not terminate.
    - The main thread is a user thread, and **child** threads, spawned from the main thread, inherit its user thread status. The **main** method can then finish, but **the program will keep running until all the user threads have completed**.
  - However, **daemon threads** are at the mercy of the runtime system: they are stopped if there are no more user threads running, thus terminating the program.
    - ForkJoinPool uses daemon threads. These are **automatically terminated when all user threads have terminated.**

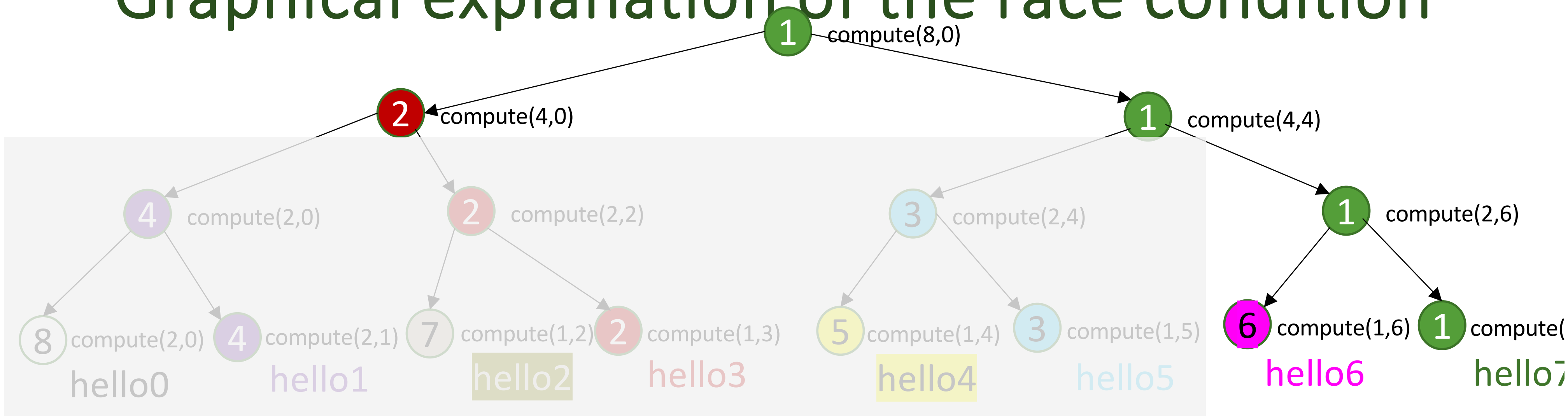
Graphical explanation of the race condition




```java
public class HelloMany
extends RecursiveAction {
      int greetings; // arguments
      int offset;

HelloMany(int g, int start) {
        greetings=g;
        offset=start;
      }

public static void main(String[] args) {
    HelloMany sayhello = new HelloMany(8,0);
    ForkJoinPool pool  = new ForkJoinPool();
    pool.invoke(sayhello);
    }}
```

```java
protected void compute(){
  if((greetings) <=1) { //only one task left, do it.
                System.out.println("hello"+offset );
  }
  else {
    int split=(int) (greetings/2.0);
    //split work into two
    HelloMany left = new HelloMany(split,offset);  //first half
    HelloMany right= new HelloMany(greetings-split,offset+split );
    left.fork(); //give first half to new thread
    right.compute(); //do second half in this thread
    left.join(); //wait unit done
} }
```

Graphical explanation of the race condition




```
public class HelloMany
extends RecursiveAction {
      int greetings; // arguments
      int offset;

HelloMany(int g, int start) {
        greetings=g;
        offset=start;
      }

public static void main(String[] args) {
    HelloMany sayhello = new HelloMany(8,0);
    ForkJoinPool pool  = new ForkJoinPool();
    pool.invoke(sayhello);
    }}
```

```
protected void compute(){
  if((greetings) <=1) { //only one task left, do it.
              System.out.println("hello"+offset );
   }
  else {
    int split=(int) (greetings/2.0);
    //split work into two
    HelloMany left = new HelloMany(split,offset);  //first half
    HelloMany right= new HelloMany(greetings-split,offset+split );
    left.fork(); //give first half to new thread
    right.compute(); //do second half in this thread
    left.join(); //wait unit done
} }
```