Concurrency:

It is the process in which an application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.

Concurrent Software:

Software's which are written in such a way that they enable performing of multiple things at the sometimes are known as concurrent software.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes.

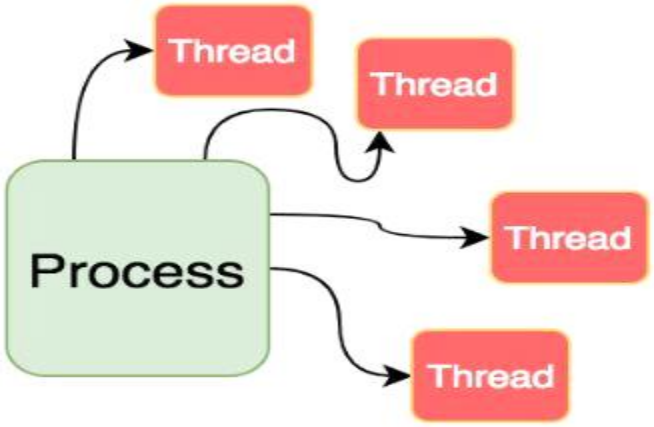
To facilitate communication between processes, most operating systems use Inter Process Communication (IPC) resources, such as pipes and sockets.

Threads

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

A Representation between Processes and Threads can be given by the following diagram:



Thread Objects

Each Thread is identified by a class Thread in Java.

There are two basic strategies for using Thread objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
- To abstract thread management from the rest of your application, pass the application's tasks to an executor.

Creating a Simple Thread in JAVA:

The SimpleThreads Example

Class SimpleThreads consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the Runnable object, MessageLoop, and waits for it to finish. If the MessageLoop thread takes too long to finish, the main thread interrupts it.

The MessageLoop thread prints out a series of messages. If interrupted before it has printed all its messages, the MessageLoop thread prints a message and exits.

```
public class SimpleThreads {


    // Display a message, preceded by

    // the name of the current thread

    static void threadMessage(String message) {

        String threadName =

            Thread.currentThread().getName();

        System.out.format("%s: %s%n",

                                threadName,

                                message);

    }


    private static class MessageLoop

        implements Runnable {
```

```
        public void run() {
                String importantInfo[] = {
                        "Mares eat oats",
                        "Does eat oats",
                        "Little lambs eat ivy",
                        "A kid will eat ivy too"
                };
                try {
                        for (int i = 0;
                                i < importantInfo.length;
                                i++) {
                            // Pause for 4 seconds
                            Thread.sleep(4000);
                            // Print a message
                            threadMessage(importantInfo[i]);
                        }
                } catch (InterruptedException e) {
                        threadMessage("I wasn't done!");
                }
        }
}

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

        // Delay, in milliseconds before
        // we interrupt MessageLoop
        // thread (default one hour).
```

```
long patience = 1000 * 60 * 60;

// If command line argument
// present, gives patience
// in seconds.
if (args.length > 0) {
        try {
                patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
        }
}

threadMessage("Starting MessageLoop thread");
long startTime = System.currentTimeMillis();
Thread t = new Thread(new MessageLoop());
t.start();

threadMessage("Waiting for MessageLoop thread to finish");
// loop until MessageLoop
// thread exits
while (t.isAlive()) {
        threadMessage("Still waiting...");
        // Wait maximum of 1 second
        // for MessageLoop thread
        // to finish.
        t.join(1000);
```

```
                if (((System.currentTimeMillis() - startTime) > patience)
                        && t.isAlive()) {
                    threadMessage("Tired of waiting!");
                    t.interrupt();
                    // Shouldn't be long now
                    // -- wait indefinitely
                    t.join();
                }
            }
            threadMessage("Finally!");
        }
}
```

Output:

```
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
```

main: Still waiting...

main: Still waiting...

main: Still waiting...

main: Still waiting...

Thread-0: Little lambs eat ivy

main: Still waiting...

main: Still waiting...

main: Still waiting...

main: Still waiting...

Thread-0: A kid will eat ivy too

main: Finally!

Synchronization

Synchronization is a tool in parallel programming which enables error free communication between threads. It eliminates two conditions during inter thread communication:

Thread interference and memory consistency errors.

There are different types of errors and inconsistencies which could arise during inter thread communication process:

- **Thread Interference** describes how errors are introduced when multiple threads access shared data.
- **Memory Consistency Errors** describes errors that result from inconsistent views of shared memory.
- **Synchronized Methods** describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
- **Implicit Locks and Synchronization** describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.

- **Atomic Access** talks about the general idea of operations that can't be interfered with by other threads.

Liveness

A concurrent application's ability to execute in a timely manner is known as its *liveness*. This section describes the most common kind of liveness problem, deadlock, and goes on to briefly describe two other liveness problems, starvation and livelock.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other.

Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed.

Example of Guarded Block (Between producer and Consumer)

In this example, the data is a series of text messages, which are shared through an object of type Drop:

```
public class Drop {
```

```
    // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

The producer thread, defined in Producer, sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

```
import java.util.Random;

public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }
```

```
    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

The consumer thread, defined in Consumer, simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

```
import java.util.Random;

public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

Finally, here is the main thread, defined in `ProducerConsumerExample`, that launches the producer and consumer threads.

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Example:

```
final public class ImmutableRGB {

    // Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red,
                        int green,
                        int blue,
                        String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }


    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                       255 - green,
                       255 - blue,
```

```
                        "Inverse of " + name);
    }
}
```

High Level Concurrency Objects

higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

There are new concurrent data structures in the Java Collections Framework.

- **Lock objects** support locking idioms that simplify many concurrent applications.
- **Executors** define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
- **Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- **Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.
- **ThreadLocalRandom** provides efficient generation of pseudorandom numbers from multiple threads.