Threading

# Multi-threading

 Def. concurrent running of multiple tasks within a program

* A program may consist of many tasks that can run concurrently
* A thread is a flow of execution from beginning to end of a task
* Multiple threads run multiple CPUS
* Multiple threads can share a single CPU--- known as time sharing
	+ The operating system is responsible for scheduling and allocating resources for threads
* Threading is an example of asynchronous programming
* Multithreading can make your program more responsive and interactive, as well as enhance performance
* In java, each task is an instance of the Runnable interface, also called a runnable object
* a thread is essentially an object that facilitates the execution of a task

# creating tasks and threads

* Tasks are objects
* To create a task you must first define a class for tasks
* A task class must implement the runnable interface
* The runnable interface contains a run method
* You need to implement the method run to tell the system how you thread is going to run

# a template for developing a class Task

public class TaskClass implements Runnable

{

 public TaskClass(…) // constructors

 {…}

 //implement the method run

 public void run()

 {

 //tell the system how to run the custom thread

 }

} //end the TaskClass

* A task must be executed in the thread.
* The thread class contains the constructors for creating threads and many useful methods & controlling threads
* To create a task object:
	+ TaskClass task = new TaskClass();
* To create a thread for a task
	+ Thread thread = new Thread(task);
* You can then invoke the start method to tell the Java Virtual Machine that thread is ready to run

#  major steps for creating a task, a thread, and starting the thread

public class Client

{

 public void someMethod()

 {

 //create an instance of TaskClass

 TaskClass task = new TaskClass(…);

 //create a thread

 Thread thread = new Thread(task);

 //start the thread

 thread.start(); //JVM will execute the task by invoking the task’s run method

 …

 } //end someMethod

} //end class Client

# **Example**

Create a console program that has three tasks and three threads to run them.

* The first task prints the letter a 100 times
* The second task prints the letter b 100 times
* The third task prints the integers 1 to 100

## Example implementation

**//create a TaskClass called PrintChar**

public char PrintChar implements Runnable

{

 private char charToPrint;

 private int times;

 **//constructors**

 public PrintChar(char c, int t) {

 charToPrint =c;
 times =t; }

**//overriding the run method to tell the system what task to perform**

 public void run()

 {

for (int i = 1; i <= times; i++)

{

System.out.print(charToPrint + “ ”);
}

} //end class

//Create a TaskClass called PrintNum for printing numbers from 1 to n for a given n

public class PrintNum() implements Runnable

{

 private int lastNum;

 public printNum(int n) { lastNum = n;}

//tell thread to how to run

 public void run()

 {

 for (int ii = 1; ii <= lastNum; ii++)
 {System.out.print(ii + “ “);

 } //end method run

} //end class

//create a class to run the tasks in the main() method

public class Test

{

 \_\_\_\_\_\_\_\_\_ main()

 {

**//create tasks**

Runnable printA = new PrintChar(‘a’, 100);

Runnable printB = new PrintChar(‘b’, 100);

Runnable print100 = new PrintNum(100);

**//create threads run tasks**

Thread t1 = new Thread(printA);

Thread t2 = new Thread(printB);

Thread t3 = new Thread(print100);

//start threads

t1.start();

t2.start();

t3.start();

}

}

**Sample output:**

a a a 1 b 2 b a a a 3 4 …

# The thread class

* The thread class contains the constructors for creating threads for tasks, and methods for controlling threads.
* Import java.lang.Runnable to use the Thread class.

## Thread class

public class Thread
{

 **//constructors**

 public Thread(){…} //creates an empty thread

 public Thread(Runnable task) {…} //creates a task for a specific task

 **//methods**

 public void start(){…} /\*starts the thread that causes the run method to be invoked by the JVM\*/

 public Boolean isAlive() {…} //test whether the thread is currently running

 public void setPriority(int i) {…} /\*sets priority p (ranging from 1(lowest)j to 10 (highest)) for the thread\*/

 public void join() {…} //waits for this thread to finish

 public void sleep(long milliseconds) {…} /\*puts a thread to sleep for a specified time in milliseconds\*/

 public void yield() {…} /\*causes the thread to pause temporarily and allow other threads to execute\*/

 public void interrupt() {…} //interrupts this thread (will be discussed in upper div. classes)

}

Note these methods are unstable and should be avoided:

* stop()
* suspend()
* resume()

There’s is another way to implement multithreading using the class thread instead of the runnable

interface (not recommended)

# implementation of multiThreading using THREAD CLASs (Not recommended)

**//CustomThread class**

public class CustomThread extend Thread

{

 …

 Public CustomThread(…) {…}

 **//overriding the abstract method run**

 public void run() {…}

}

**//Client class**

public class Client

{

 …

 public void doSomething()

{

 …

CustomThread th = new CustomThread(…);

 th.start();

 …

}

}

## Using yield()

public void run()

{

 For (int i = 0; i< lastNum; i++)

 {

 System.out.print(“ “ + i);

 Thread.yield();

 }

}

Every time a number is printed, the thread of the print100 task is yielded. So each number is followed by

some characters.

## Using sleep( int milliseconds)

* puts the thread to sleep for a specified time in milliseconds to allow other threads to execute
* sleep method might throw an interruptexception

public void run()

{

 try

 {

 for (int i = 0; i < lastNum; i++)

 {

 System.out.print(“ “ + i);

 If (i >= 50)

 {Thread.sleep(1);}

 }

 } //end try block

 Catch(InterruptedException e)

 {

 }

} //end method

## Using join()

* forces one thread to wait for another to finish

public void run()

{
Thread t4 = new Thread(new PrintChar(‘c’, 40));

 t4.start();

 Try

 {

 for(int i = 0; i < lastNum; i++)

 {

 System.out.print(“ “ + i);

 if( i==50)

 {t4.join();}

 }

 } //end try

 Catch (InterruptedException e) {

 }

A new thread4 is created. It prints character c 40 times. The numbers from 50 to 100 are printed after

thread thread4 is finished.

# Thread Priority

* If all runnable threads have the same priority, each is assigned an equal portion of CPU in a circular queue. This is called a round-robin scheduling.
* You can increase and decrease thread priority using the setPriority(int) method.
	+ 1 (lowest) – 10 (highest)
	+ You can also use int constants
		- MIN\_PRIORITY = 1
		- NORM\_PRIORITY = 5
		- MAX\_PRIORITY = 10
	+ setPriority(5) and setPriority(NORM\_PRIORITY) are equivalent
	+ The JVM always picks up the current runnable thread with the highest priority. A lower priority thread can run only when no higher priority thread are running

# Thread Pool

* Creating tasks and threads we learned are not efficient

Runnable task1 = new Task(task);

Thread t = new Thread (task1);

t.start();

* This approach is convenient for a single task execution but it isn’t efficient for a large number of tasks because you have to create a thread for each task
	+ Starting a new thread for each task could limit the throughput and cause poor performance
* A thread pool is ideal to manage the tasks
	+ Java provides:
		- Executor interface- executing tasks in a thread pool
			* To create an executor object use the static methods in the Executors class
		- ExecutorService interface- managing and controlling tasks

# Executors Class

Methods

1. ExecutorService newFixedThreadPool (int numOfThreads)
2. ExecutorService newCachedThreadPool()

## newFixedThreadPool

Creates a thread pool with a fixed number of threads executing concurrently. A thread might be reused to execute another task after its current task is finished

## newcachedthreadpool

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they’re available

# using a thread pool

**Main**

//create a fixed thread pool with a maximum of three threads

ExecutorService executor = Executors.newFixedThreadPool(3);

//submit runnable tasks to executor

executor.execute(new PrintChar(‘c’, 100));

executor.execute(new PrintChar(‘a’, 100));

executor.execute(new PrintNum(100));

**Notes:**

* the executor creates three threads to execute three tasks concurrently
* if we change the executor to create only one thread in the thread pool

ExecutorService executor = Executors.newFixedThreadPool(1);

* Then the three runnable tasks will be executed *sequentially*

## if we use newCachedThreadPool

ExecutorService executor = Executors.newCachedThreadPool();

* New threads will be created for each waiting task, so all the tasks will execute concurrently
* To shut down the executor uses the method shutdown
executor.shutdown();