CSE 1325

Week of 10/31/2022

Instructor : Donna French

Override vs Overload

When the method signature (name and parameters) are the same in the superclass and the child class, it's called overriding.

When two or more methods in the same class have the same name but different parameters, it's called overloading.

Override vs Overload

Overriding

- Occurs between superclass and subclass
- Have the same signature (name and method arguments)
- `toString()` and `getArea()` from `Shape` and `move()` from `Animal` and `draw()` from `SpaceObject`

Overloading

- Occurs between the methods in the same class
- Have the same name, but the parameters are different
- `addThem(int, int)` vs

  `addThem(double, double)`

Override vs Overload

Overriding

- It is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
- It is performed in two classes with inheritance relationships.
- Method overriding always needs inheritance.

Overloading

- It helps to increase the readability of the program.
- It occurs within the class.
- Method overloading may or may not require inheritance.

Override vs Overload

Overriding

- In method overriding, methods must have the same name and same signature.
- In method overriding, the return type must be the same.
- Argument list should be same in method overriding.

Overloading

- In method overloading, methods must have the same name and different signatures.
- In method overloading, the return type can or can not be the same, but we just have to change the parameter.
- Argument list should be different while doing method overloading.

Coding Assignment 4

- Add command line parameters
  - read in a file
    - IFILENAME=xxxxxx
  - write out a file
    - OFILENAME=xxxxx
- Parse file of pipe delimited Coke Machines information using `split()`
  - name|price|change|inventory
- Create and manipulate an `ArrayList` of Coke Machines objects
- Display menu of Coke Machines and allow operations on each machine
- Exception handling
- Default constructor
- Overload `toString()` to print object

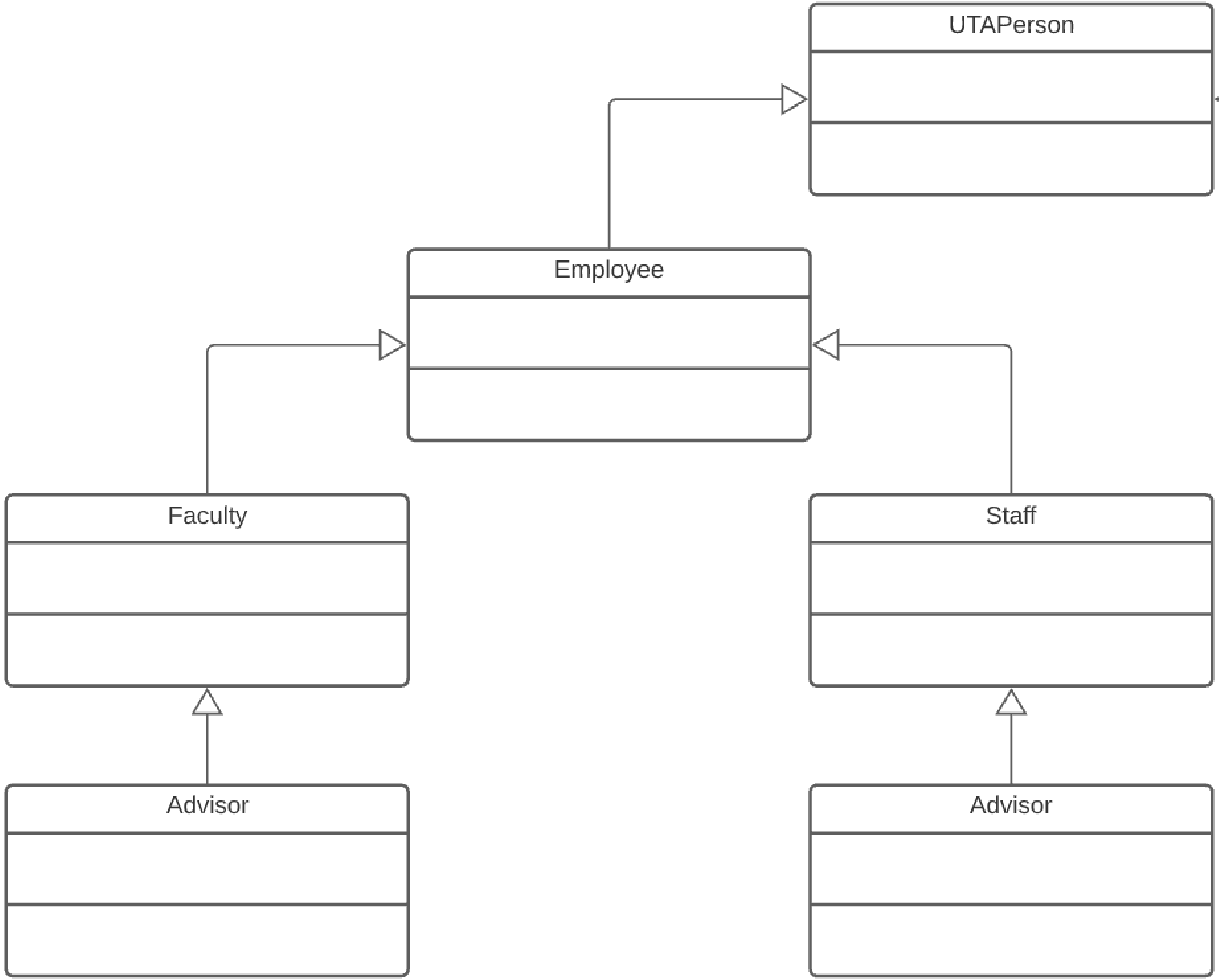
Coding Assignment 4

```
Machine Bugs Bunny|50|500|50
Machine Cecil Turtle|45|545|45
Machine Daffy Duck|40|540|1
Machine Elmer Fudd|100|1000|10
Machine Fog Horn|35|350|99
```

Coding Assignment 4

```
Pick a Coke Machine

0. Exit
1. Machine Bugs Bunny
2. Machine Cecil Turtle
3. Machine Daffy Duck
4. Machine Elmer Fudd
5. Machine Fog Horn
6. Add a new machine

Enter choice 1
```

```
0. Walk away

1. Buy a Coke

2. Restock Machine

3. Add change

4. Display Machine Info

5. Update Machine Name

6. Update Coke Price
```



Would a Faculty Advisor and Staff Advisor have things in common?

interfaces

Do Faculty Advisor and Staff Advisor have things in common?

Let's look at this question from a security aspect and information access…

A Faculty member needs access to their class information.

A Staff member needs access to departmental information.

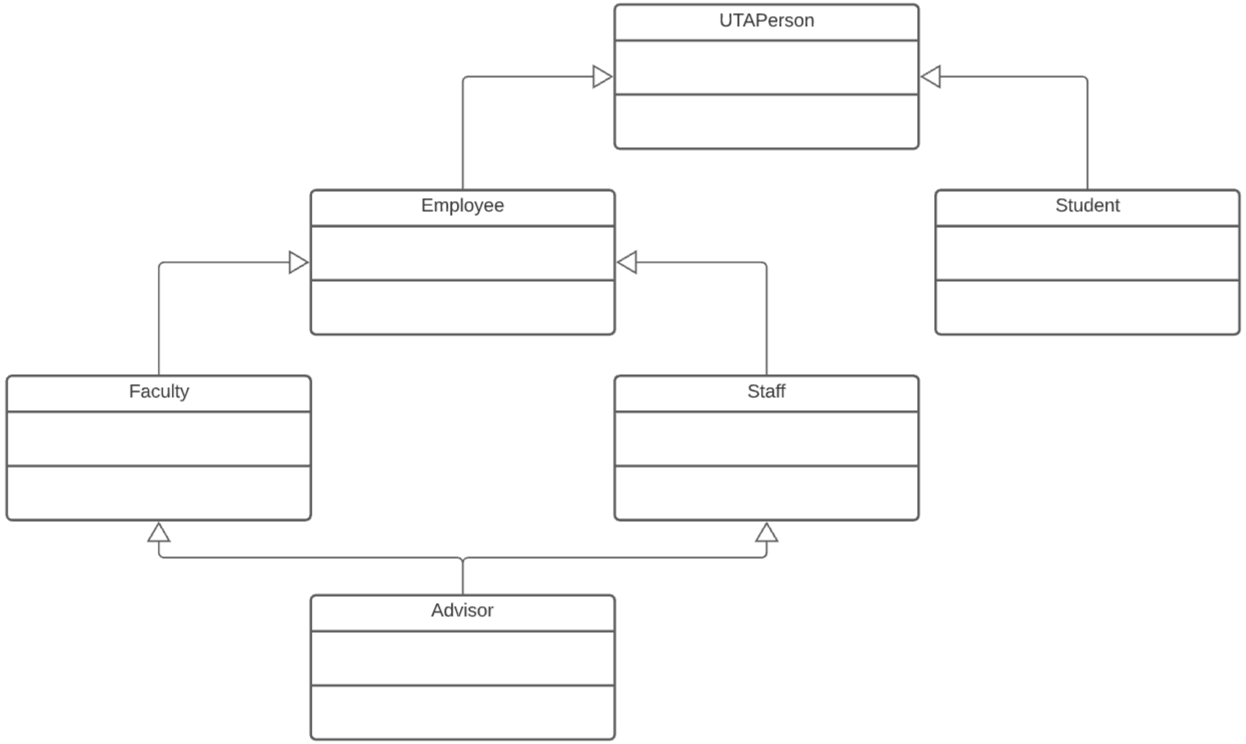
An Advisor needs access to student information.

A Faculty Advisor member needs access to both Faculty and Advisor information.

A Staff Advisor member needs access to both Staff and Advisor information.

interfaces

Seems like Advisor needs to inherit from both Faculty and Staff??



This is multiple inheritance – also know as Diamond Inheritance.

Multiple inheritance creates several issues because of inheriting more than one set of instance variables that have the same name.

Advisor would get Staff's Employee instance variables AND Faculty's Employee variables.

C++ allows this and has special code to fix the issues.

Java does not allow multiple inheritance.

interfaces

We do not actually need a class that inherits from Faculty and Staff.

We need the Faculty Advisor class and the Staff Advisor class to share a common set of methods. They have their own attributes (instance variables). They just need to be able to perform a common set of tasks.

An Advisor has access to student records; therefore, both a Faculty Advisor and a Staff Advisor need access to student records.

Java has the ability to require classes to implement a set of common methods.

interfaces

Interfaces define and standardize the ways in which things interact with each other.

The interface specifies what operations an object must permit users to perform but does not specific how the operations are performed.

Remember our `SpaceObject` and our `Shape` classes? What if we need a `draw()` method that can draw either a `SpaceObject` (or its subclasses) or a `Shape` (or its subclasses)?

`SpaceObject` and `Shape` would not be in the same hierarchy, but we still want to be able to `draw()` either of them.

interfaces

An `interface` declaration begins with the keyword `interface` and contains **only** constants and abstract methods.

All `interface` members must be `public` and interfaces may not specify any implementation details.

No concrete methods declarations

No instance variables

All methods declared in an `interface` are implicitly `public abstract`.

interfaces

To use an `interface`, a concrete class must

specify that it **`implements`** the `interface`

declare each method in the `interface` with the signature specified in the `interface` declaration

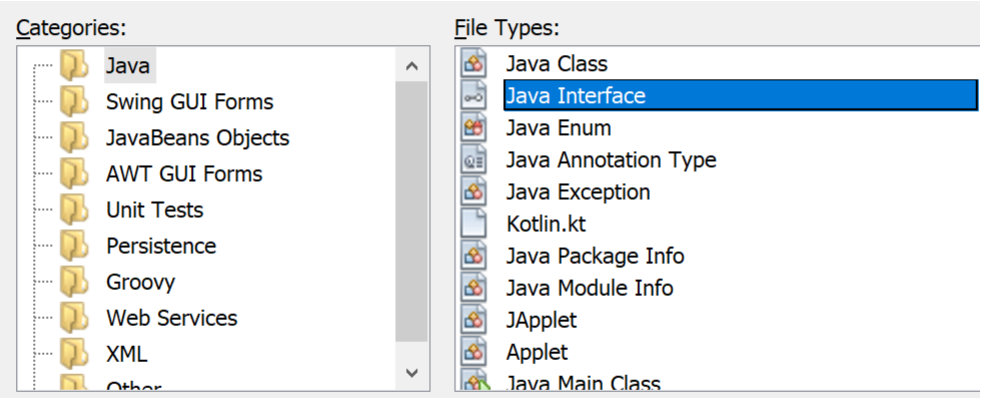
If a class does not implement ALL of the methods of the `interface`, then that class will be `abstract`.

interfaces vs abstract classes

An interface is often used instead of an abstract class where there's no default implementation to inherit.

no instance variables and no default method implementations

Instead of creating a Java Class file, pick a Java Interface file.

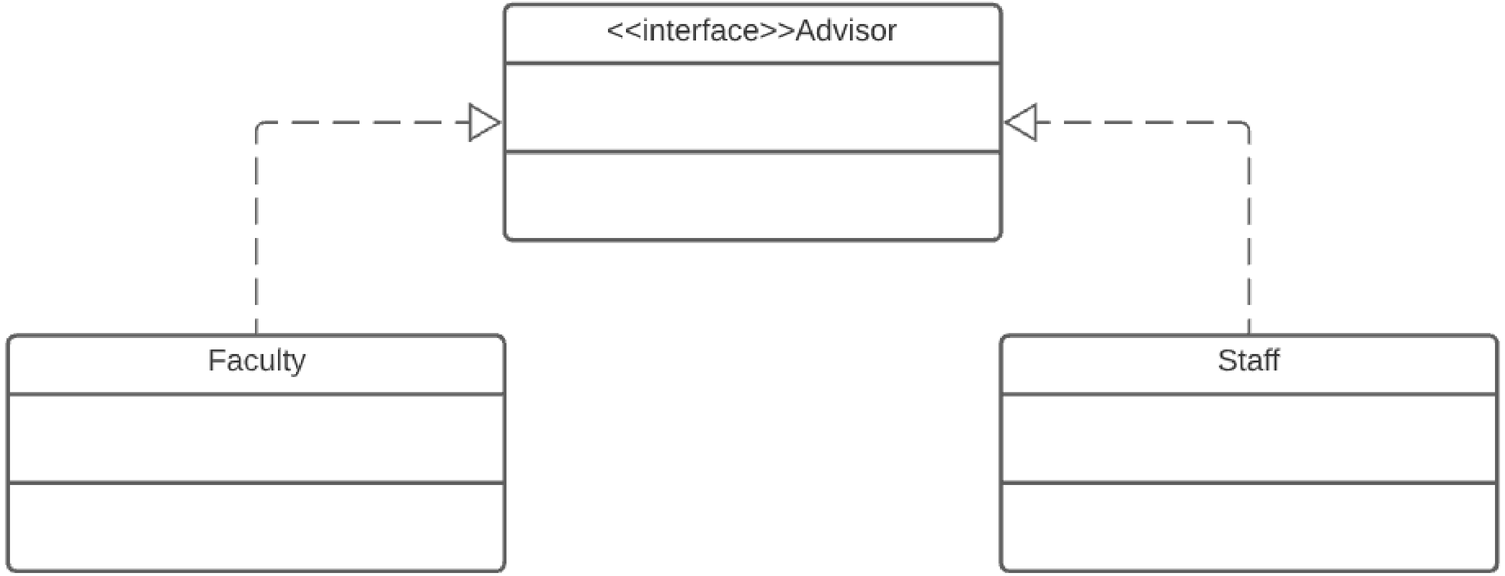



```
public interface NewInterface
{

}
```

interfaces

The UML for an interface expresses the relationship between the class and an interface through a relationship known as realization.

A class is said to *realize*, or `implement`, the methods of an `interface`.

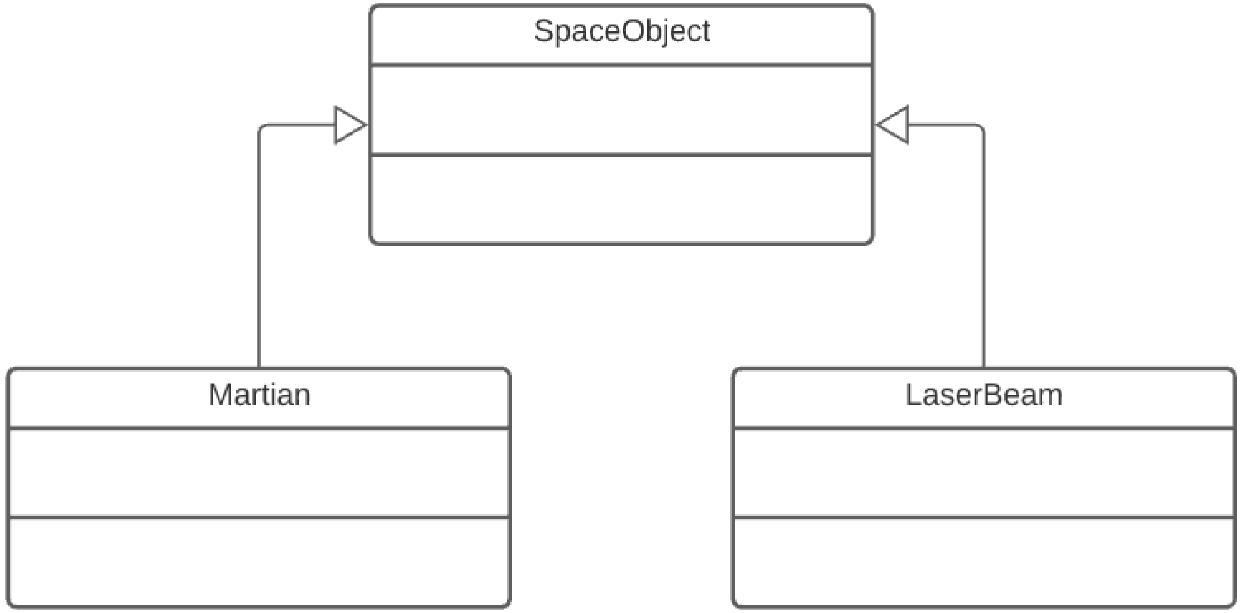


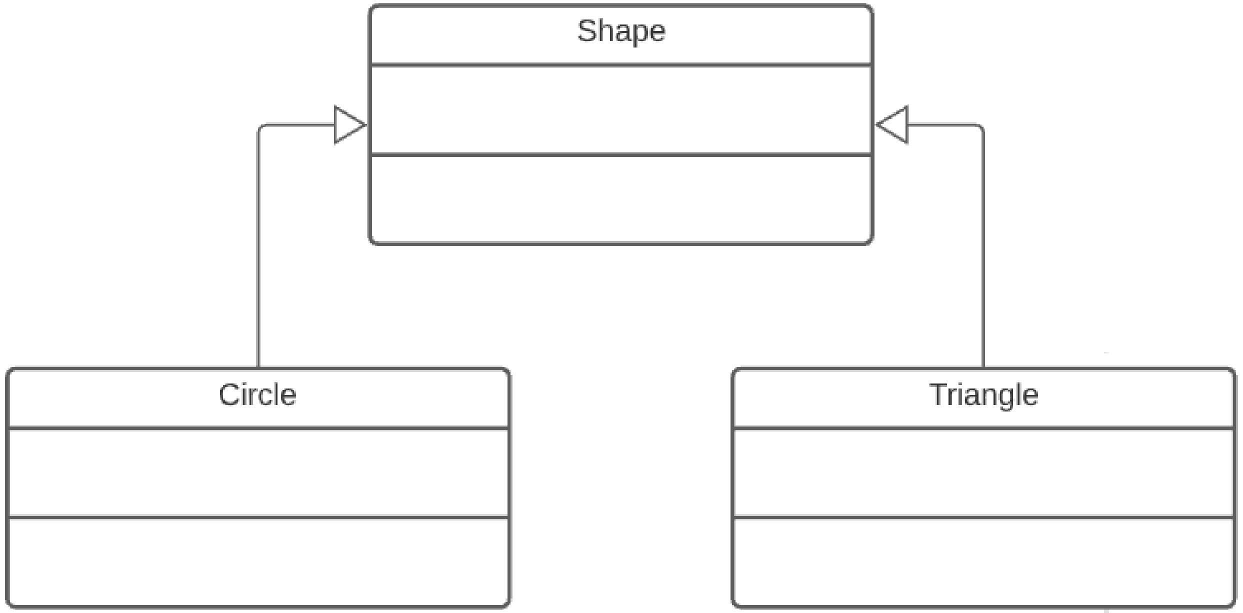
interfaces

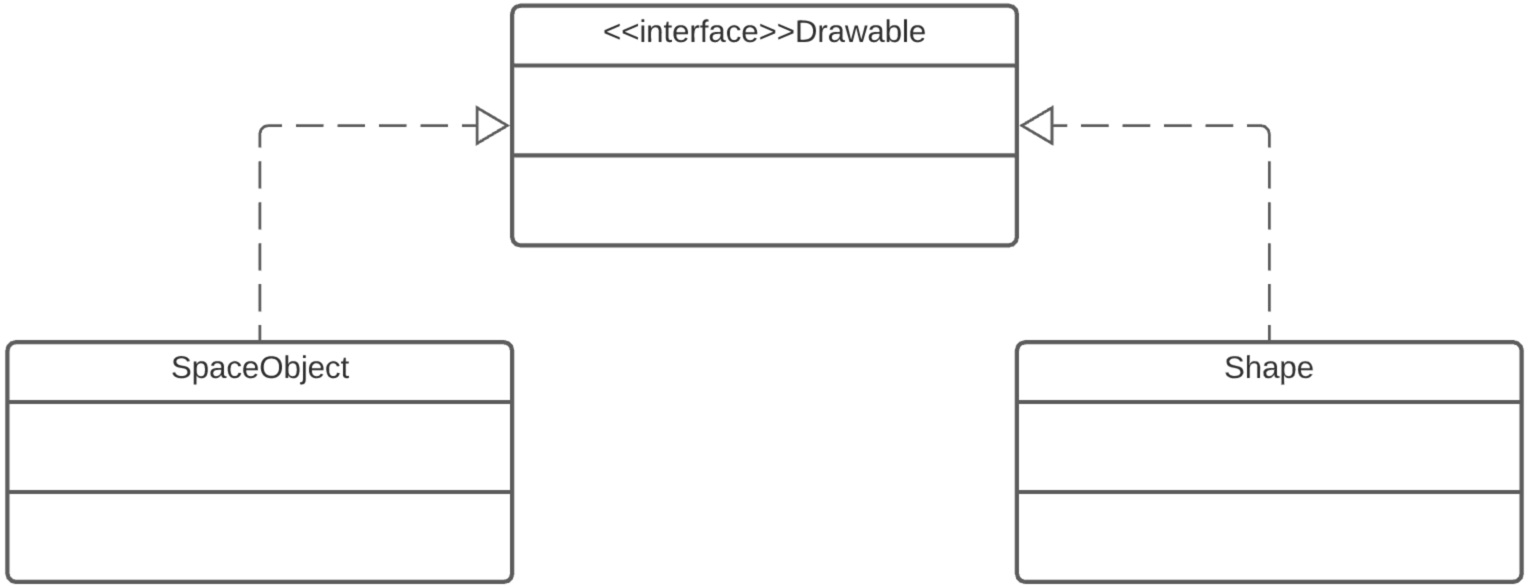
```
public interface Advisor
{
   public String readStudentRecord(String studentID);
}

public class FacultyAdvisor extends Faculty implements Advisor

public class StaffAdvisor extends Staff implements Advisor
```

interfaces

```
public class FacultyAdvisor extends Faculty implements Advisor
{
   @Override
   public String readStudentRecord(String studentID)
   {
      //actual code to read a student record using Faculty+Advisor permissions
   }
}

public class StaffAdvisor extends Staff implements Advisor
{
   @Override
   public String readStudentRecord(String studentID)
   {
      //actual code to read a student record using Staff+Advisor permissions
   }
}
```


SpaceObject
Martian
LaserBeam


Shape
Circle
Triangle


<<interface>>Drawable
SpaceObject
Shape

```
public interface Drawable
{
   public void draw();
}

public class SpaceObject implements Drawable

public class Shape implements Drawable


public class Martian extends SpaceObject

public class Circle extends Shape
```

interfaces

When a class implements an interface, the same *is-a* relationship provided by inheritance applies.

Class `SpaceObject` implements `Drawable` so we can say that `SpaceObject` is a `Drawable`.

Class `Shape` implements `Drawable` so we can say that `Shape` is a `Drawable`.

Objects of any class that extend `SpaceObject` are `Drawable` objects.

Objects of any class that extend `Shape` are `Drawable` objects.

interfaces

Now we can create a container to hold `Drawable` objects.

```
Drawable[] drawableObjects = new Drawable[100];

ArrayList <Drawable> drawableObjects = new ArrayList<>();

for (Drawable it : drawableObjects)
{
   it.draw();
}
```

polymorphism at work!!

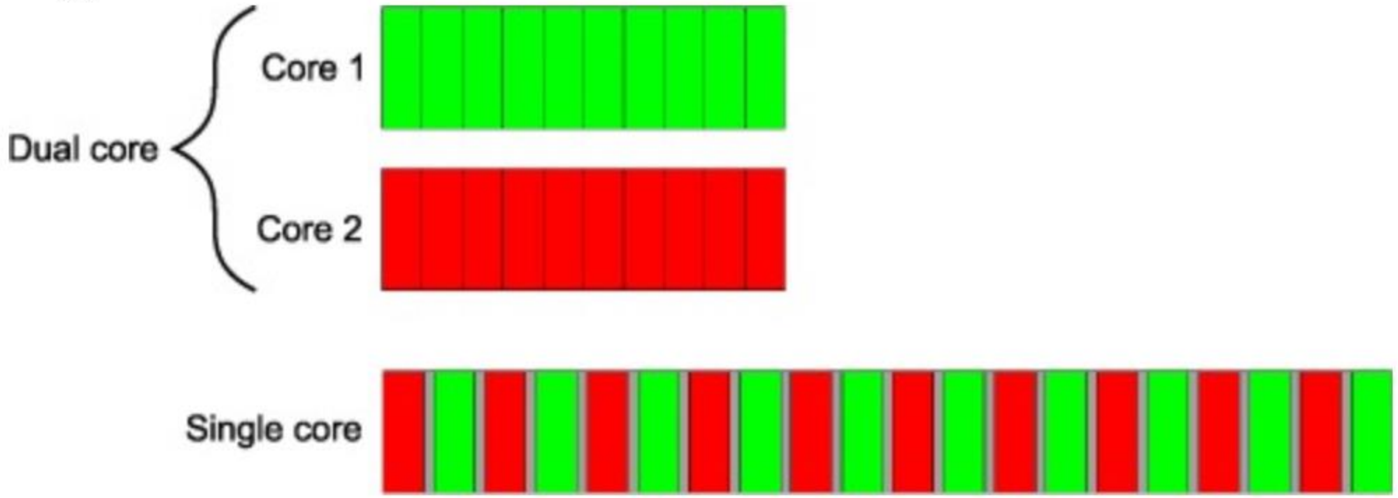
Multithreading

Most of today's computers, smartphones and tablets are typically multicore.

The most common level of multicore processor today

dual core

quad core



In multicore hardware systems, the hardware can put multiple processes to work simultaneously on different parts of your task; thereby, enabling the program to complete faster.

Multithreading

To take full advantage of multicore architecture, we need to write multithreaded applications.

When a program splits tasks into separate threads, a multicore system can run those threads in parallel.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background of your system.

Multithreading

Therefore, it is important to recognize that different runs of the same process may take different amounts of time and the various threads may run in different orders at different speeds.

There's also overhead inherent to multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not guarantee that it will run twice as fast.

Multithreading

There is not guarantee of which threads will execute when and how fast they will execute regardless of how the program is designed or how the processors are laid out.

Concurrent vs Parallel

Concurrent tasks

tasks that are making progress at once

Parallel

tasks are executing simultaneously

Parallelism is a subset of concurrency

Are you parallel or concurrent?

Concurrent vs Parallel

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

An application can be concurrent — but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at the same time instant.

An application can be parallel — but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at the same time

Java Concurrency

Java programs can have multiple threads of execution.

Each thread can execute concurrently with other threads while sharing with them application-wide resources like memory and file handles.

This sharing and executing concurrently is called multithreading.

The Java Virtual Machine (JVM) creates threads to run programs and threads to perform tasks like garbage collection.

Concurrent Programming Use

When streaming a video over the Internet, the user does not want to wait until the entire video downloads before starting the video.

Multiple threads are used

the producer downloads the video

the consumer plays the video

To avoid choppy playback, the producer thread and the consumer thread must be synchronized

The consumer thread does not start playback until a sufficient amount of video has been downloaded.

Producer and consumer threads share memory

Concurrent Programming

An experiment for you…

Open three books to page 1 and read the books concurrently….

Read a few words from the first book, then a few words from the second book and then a few words from the third book.

Loop.

Challenges of multithreading

- switching between books
- reading only a few words
- remembering your places in each book
- moving the book you're reading closer so you can see it and then pushing it away and replacing it with the next book

Concurrent Programming

Writing multithreaded programs can be challenging.

There's a reason why there are whole classes dedicated to Parallel Processing

CSE 4323. QUANTITATIVE COMPUTER ARCHITECTURE. 3 Hours.

Pipelined processors, parallel processors including shared and distributed memory, multicore, Very Long Instruction Word (VLIW) and graphics processors, memory and cache design, computer peripherals, and computer clusters.

CSE 4351. PARALLEL PROCESSING. 3 Hours.

Theory and practice of parallel processing, including characterization of parallel processors, models for memory, algorithms, and interprocess synchronization. Issues in parallelizing serial computations, efficiency and speedup analysis. Programming exercises using one or more concurrent programming languages, on one of more parallel computers.

Concurrent Programming

We will only be using the prebuilt classes of the concurrency APIs.

The goal of this class is to gain a better understanding of threads and concurrency.

We will do some simple multithreading coding but most of the focus will be on vocabulary and understanding the concepts on a high level.

Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Process

A self-contained execution environment including its own memory space.

Thread

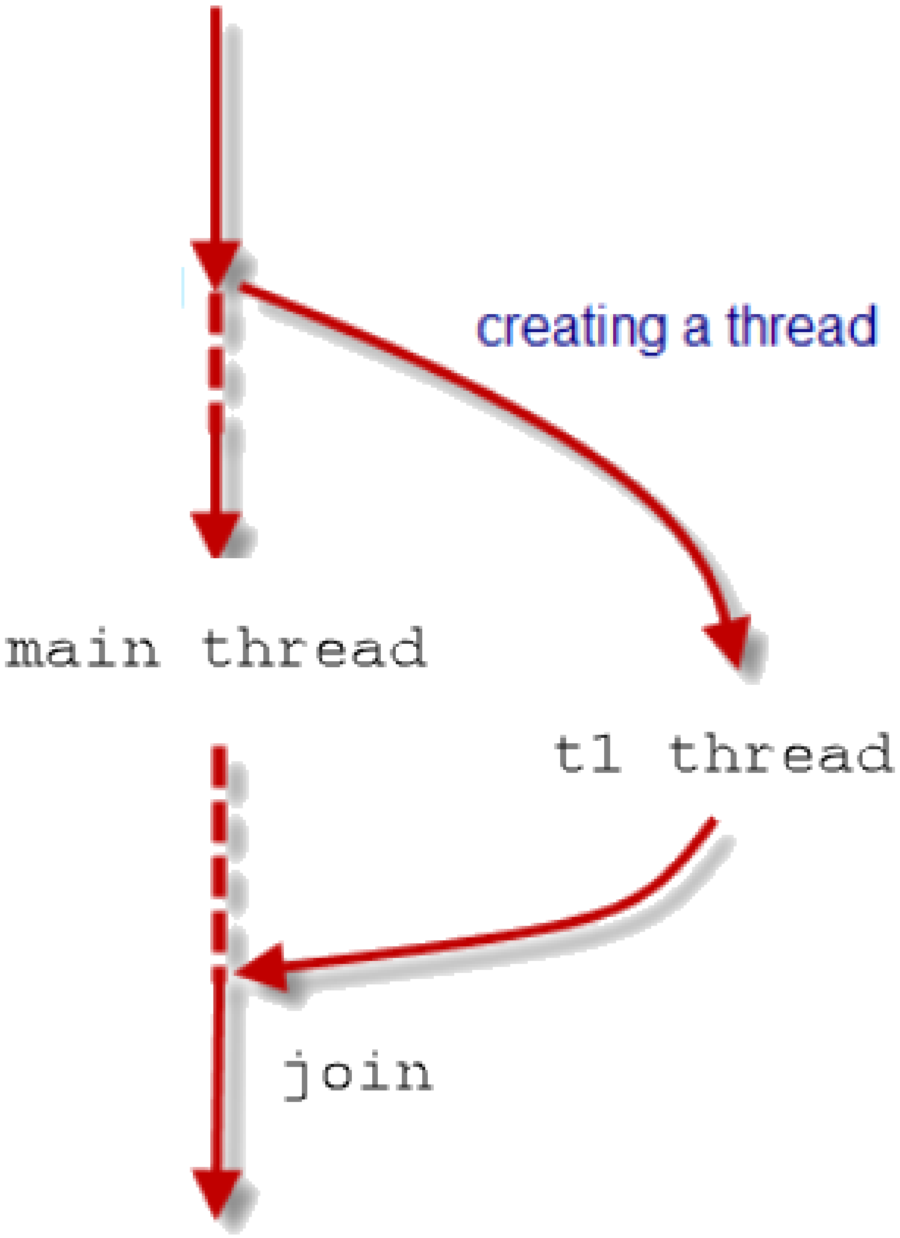
An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.

Threads

Class to represent individual threads of execution.

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

`main()` is a thread



Threads

Real World Examples of Threads

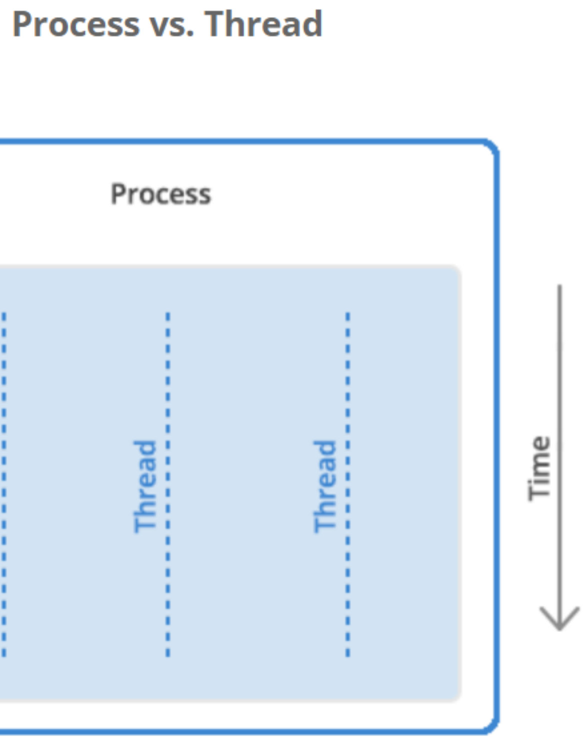
Text editor – one thread is accepting your typing, one thread is checking your spelling, one thread is occasionally saving your document. etc…

Video game – one thread is tracking your health, one thread is tracking your position, one thread is tracking your ammo, etc…

You – one thread is breathing, one thread is keeping your heart beating, one thread is falling asleep, one thread is halfway listening, etc…

Threads

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.

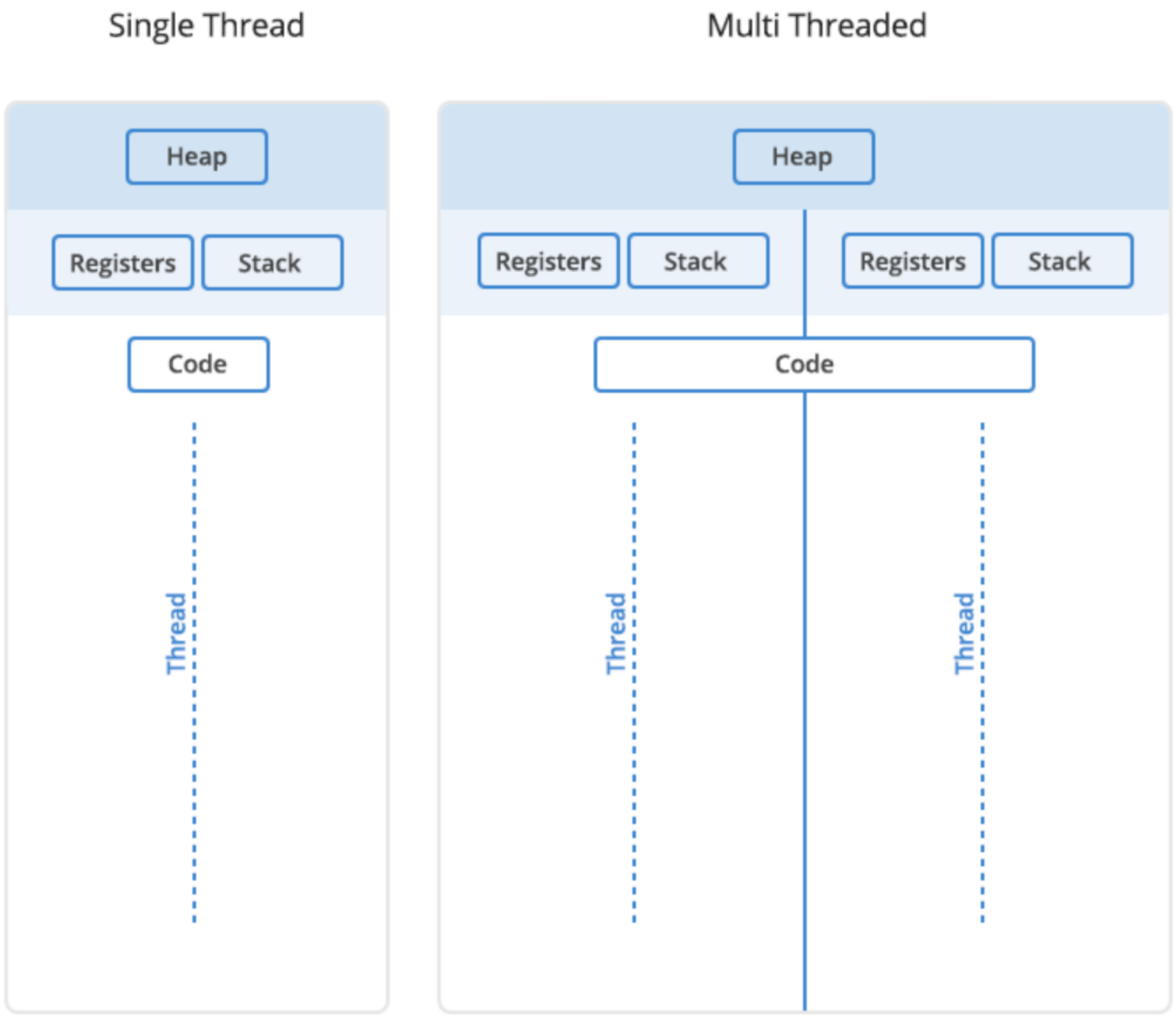


Threads

When a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources.

In single-threaded processes, the process contains one thread. The process and the thread are one and the same, and there is only one thing happening.

In multithreaded processes, the process contains more than one thread, and the process is accomplishing a number of things at the same time.


Single Thread
Multi Threaded
Heap
Registers
Stack
Code
Thread
Heap
Registers
Stack
Registers
Stack
Code
Thread
Thread

Threads

Two types of memory are available to a process or a thread

the stack

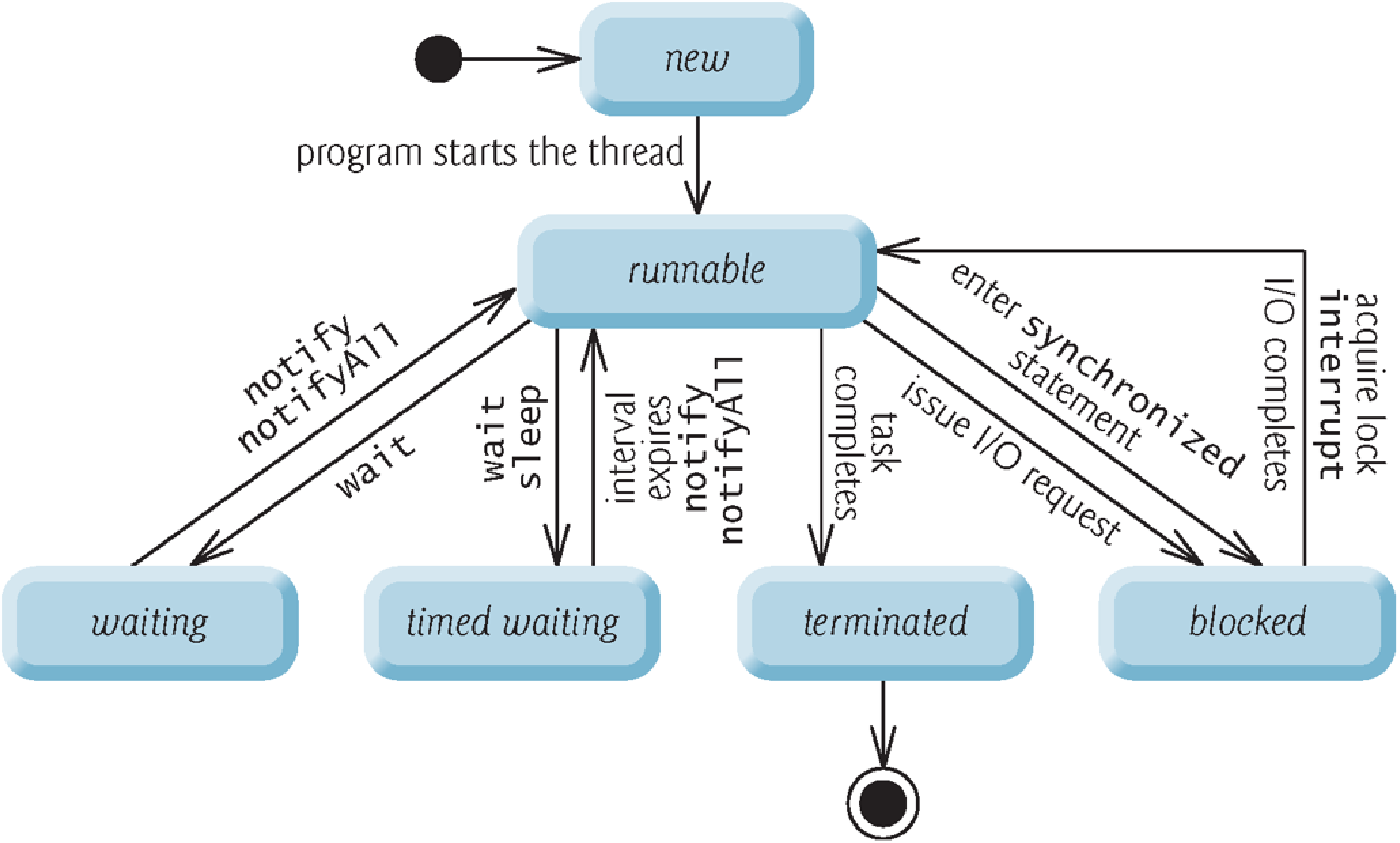
the heap

It is important to distinguish between these two types of process memory because

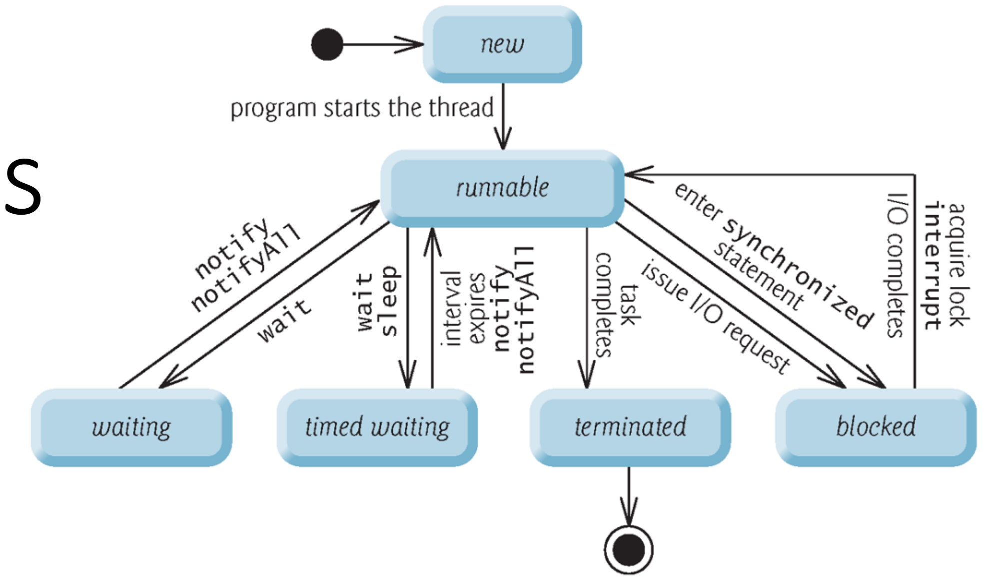
each thread will have its own stack

all the threads in a process will share the heap

Thread States



Thread States



New and Runnable States

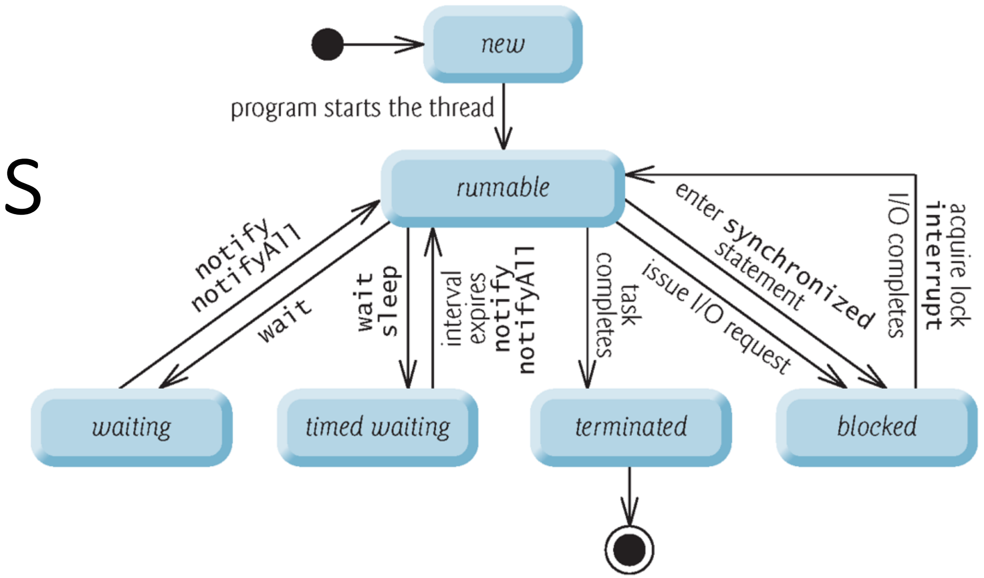
A new thread starts the life cycle in the *new* state.

It remains in this state until the program starts the thread.

When a thread is started, it moves to the *runnable* state.

A thread in a *runnable* state is executing its task.

Thread States

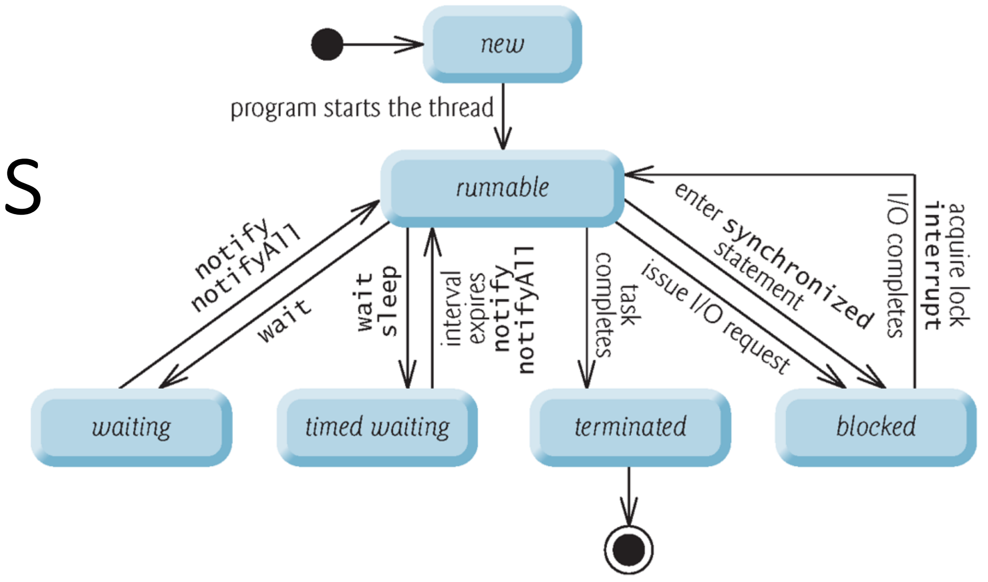


Waiting State

Sometimes a ***runnable*** thread transitions to the ***waiting*** state while it waits for another thread to perform a task.

A ***waiting*** thread transitions back to the ***runnable*** state only when another thread notifies it to continue executing.

Thread States



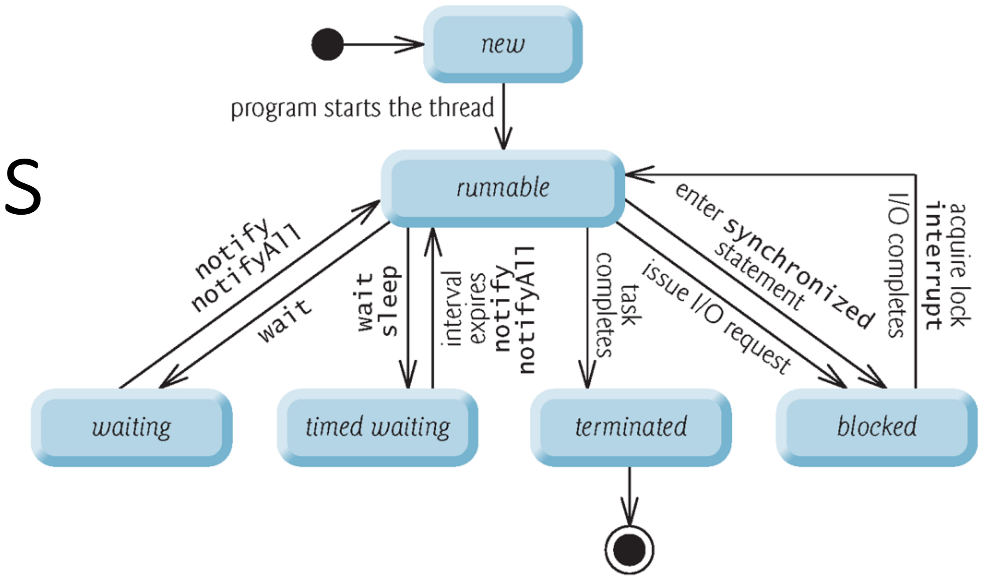
Timed Waiting State

A ***runnable*** thread can enter the ***timed waiting*** state for a specified interval of time.

It transitions back to ***runnable*** when the time interval expires.

Placing a ***runnable*** thread into the ***timed waiting*** state for a designated period of time is called putting the thread to ***sleep***

Thread States



Sleeping Thread

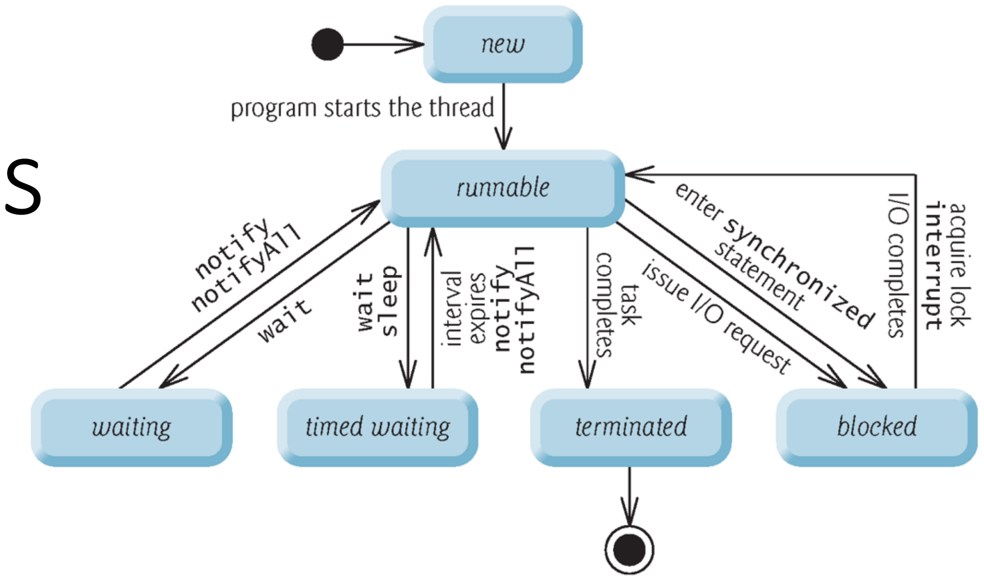
A sleeping thread remains in the ***timed waiting*** state for a designated period of time called the ***sleep interval***.

When it wakes up, the thread returns to the ***runnable*** state.

Threads sleep when they momentarily do not have work to do.

Timed waiting threads and waiting threads cannot use a processor even if one is available.

Thread States



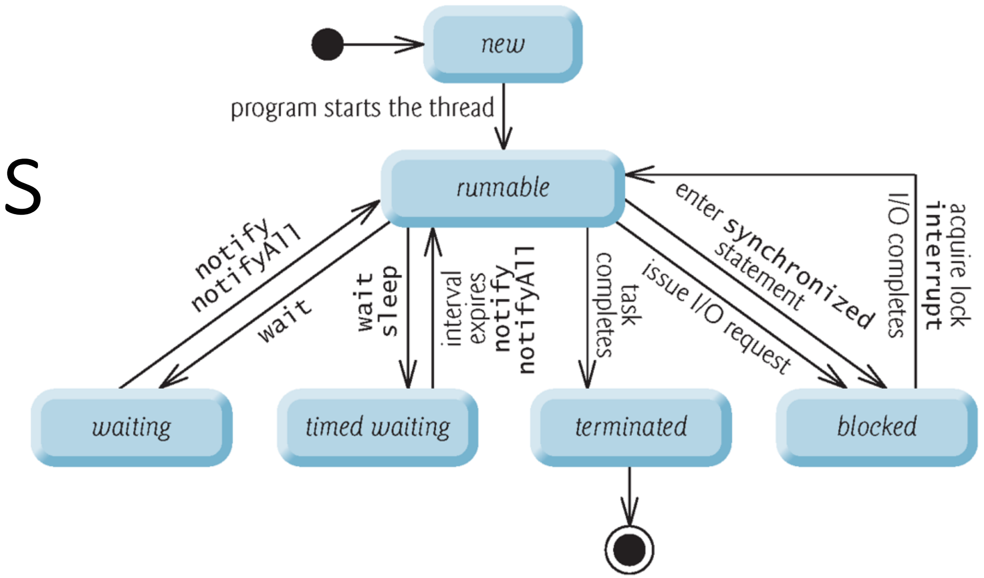
Blocked Thread

A ***runnable*** thread transitions to the ***blocked*** state when it attempts to perform a task that cannot be completed immediately.

The thread must temporarily wait until the task that kept it from completing completes and then the thread can complete its task.

I can't cross the street until the car goes by. I am blocked from finishing my task of crossing the street while another task (the car driving by) completes.

Thread States



Terminated Thread

A ***runnable*** thread enters the terminated state (sometimes called the dead state) when it successfully completes its task or otherwise terminates.

Something unrecoverable can happen to a thread to cause it to terminate (remember `ThreadDeath` from the `Error` exception hierarchy?)

Executor

We are going to perform concurrent tasks in our programs using **`Executor`**s and **`Runnable`** objects.

`Runnable` is an interface.

The **`Runnable`** interface declares a single method **`run`**

`run` contains the code that defines the task that a **`Runnable`** object should perform

Executor

To allow a **`Runnable`** (a class that implemented **`Runnable`** and overrode **`run()`**) to perform its task, we must execute it.

An `Executor` object executes **`Runnable`**s.

An `Executor` object creates and manages a group of threads called a **thread pool**.

When an **`Executor`** begins executing a **`Runnable`**, the **`Executor`** calls the **`Runnable`** object's **`run`** method which executes in the new thread.

Executor

The **`Executor`** class declares a single method named **`execute`** which accepts a **`Runnable`** as an argument.

Every **`Runnable`** is assigned a thread from the **`Executor`**'s thread pool.

If there are no available threads, then the **`Executor`** will either create a new one or it will wait for one of the existing ones to become available.

`Executor` manages the thread pool so that we don't have to (we don't want to).

Executor

The **ExecutorService** interface *extends* **Executor**

declares various methods for handling an **Executor**

You obtain an **ExecutorService** object by calling one of the `static` methods declared in class **Executors**.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

Executor

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

Can this statement throw an exception? Do we need to add a `try-catch`?

newCachedThreadPool

`public static ExecutorService newCachedThreadPool()`

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to `execute` will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using `ThreadPoolExecutor` constructors.

Returns:

the newly created thread pool

newCachedThreadPool

`public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)`

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new threads when needed.

Parameters:

`threadFactory` - the factory to use when creating new threads

Returns:

the newly created thread pool

Throws:

`NullPointerException` - if threadFactory is null

Executor

```
public static void main(String[] args)
{
    ExecutorService executorService = Executors.newCachedThreadPool();

    System.out.println(executorService);
}
```

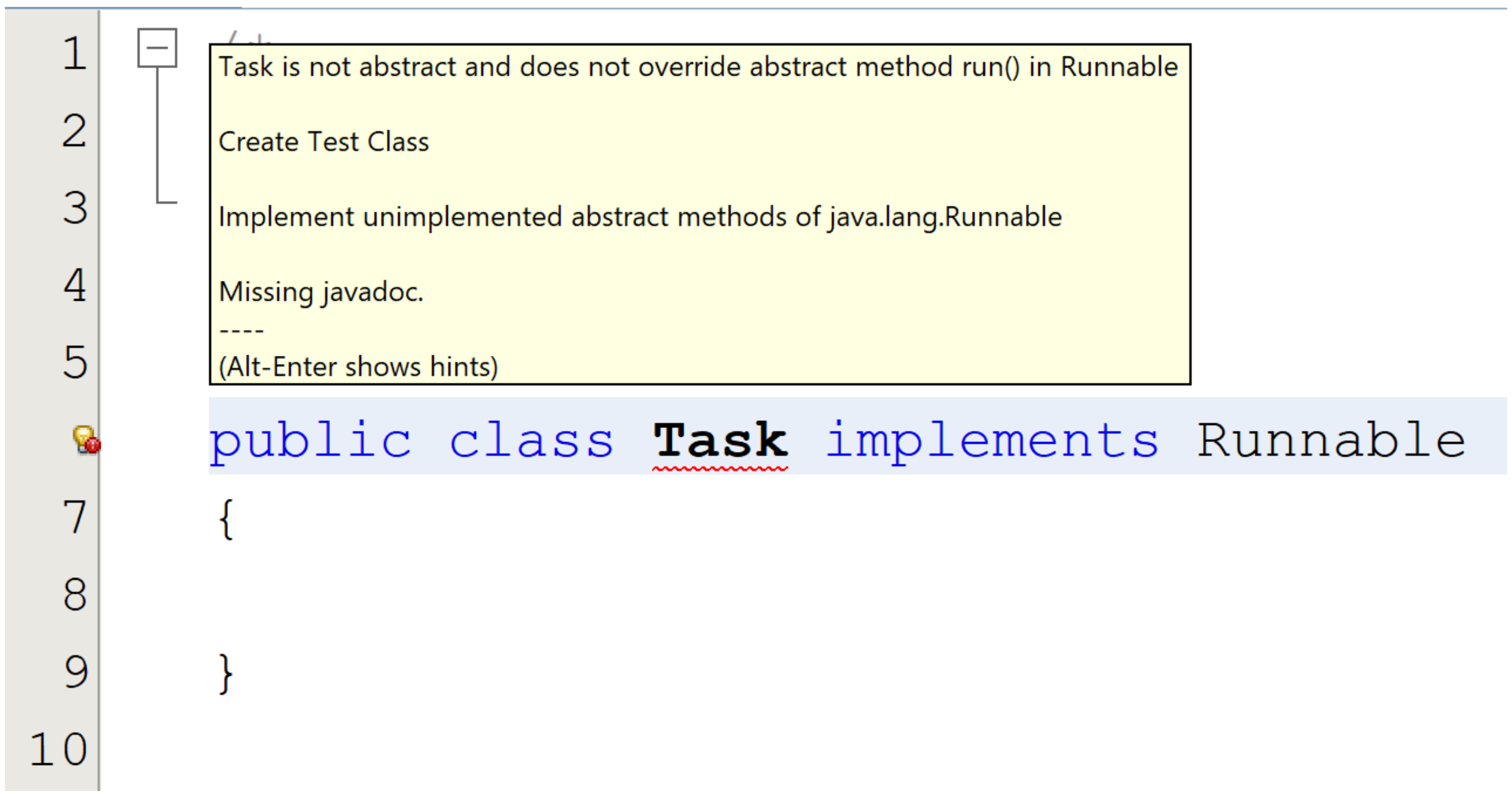
```
java.util.concurrent.ThreadPoolExecutor@30f39991[Running, pool size = 0, active
threads = 0, queued tasks = 0, completed tasks = 0]
```

Don’t forget these

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

Executor

Before we go any further with the `Executor`, we need to create a `Runnable` object.



Runnable

```
public class Task implements Runnable
{
     Random rn = new Random();
     private final int sleepTime;
     private final String name;
}
```

> variable sleepTime not initialized in the default constructor
> ----
> (Alt-Enter shows hints)

> variable name not initialized in the default constructor
> ----
> (Alt-Enter shows hints)

If you have an instance variable whose value will NOT change once it is set, then it is a good idea to make it **`final`** so that it CANNOT be changed by anyone ever.

`final`

Always declare data fields that you do not expect to change as **`final`**.

Primitive variables that are declared as **`final`** can safely be shared across threads.

An object reference that's declared as **`final`** ensures that the object it refers to will be fully constructed and initialized before it's used by the program and prevents the reference from pointing to another object.

Strings

Runnable

```
public class Task implements Runnable
{
    Random rn = new Random();
    private final int sleepTime;
    private final String name;

    public Task(String taskName)
    {
        name = taskName;
        sleepTime = rn.nextInt(5000);
    }
}
```

The value assigned to `sleepTime` represents the number of milliseconds the thread will sleep.

Constructor will set `name` to passed in value and `sleepTime` to a random number between 0 and 4999

Runnable

```
public class Task implements Runnable
{
    Random rn = new Random();
    private final int sleepTime;
    private final String name;

    public Task(String taskName)
    {
        name = taskName;
        sleepTime = rn.nextInt(5000);
    }

    public void run()
    {
        System.out.printf("%s is going to sleep for %d milliseconds\n",
                                   name, sleepTime);
        Thread.sleep(sleepTime);
    }
}
```

`sleep()` is a `static` method of the class `Thread`.

`sleep()` will place the thread in the ***timed waiting*** state for the specified number of milliseconds.

The thread loses the processor and the system will allow another thread to execute.

When the time is up, the thread wakes up and enters the ***runnable*** state.

Runnable

Do we need a `try-catch` for `Thread.sleep()`?

Let's check...

sleep

```
public static void sleep(long millis)
                  throws InterruptedException
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

Parameters:

`millis` - the length of time to sleep in milliseconds

Throws:

`IllegalArgumentException` - if the value of `millis` is negative

`InterruptedException` - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

Class InterruptedException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.InterruptedException

All Implemented Interfaces:

Serializable

```
public class InterruptedException
extends Exception
```

Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity. Occasionally a method may wish to test whether the current thread has been interrupted, and if so, to immediately throw this exception. The following code can be used to achieve this effect:

```
  if (Thread.interrupted())  // Clears interrupted status!
      throw new InterruptedException();
```

unreported exception InterruptedException; must be caught or declared to be thrown
----
(Alt-Enter shows hints)

```
public void run() throws InterruptedException
```

```
public void run() throws InterruptedException
```

run() in Task cannot implement run() in Runnable
  overridden method does not throw InterruptedException

When you override a method, the `throws` may contain only the same exception types or a subset of the exception types declared in the original method's `throws` clause.

`Runnable` method `run()` does not have a `throws` clause in its original declaration; therefore, we cannot add one.

```
public void run()
{
    System.out.printf("%s is going to sleep for %d milliseconds\n",
                       name, sleepTime);
    try
    {
        Thread.sleep(sleepTime);
    }
    catch (InterruptedException e)
    {
        Thread.currentThread().interrupt();
    }
    System.out.printf("%s is done sleeping\n", name);
}
```

```
catch (InterruptedException e)
{
    Thread.currentThread().interrupt();
}
```

There's not much we can do with an `InterruptedException`.

What we can do is get a reference to the current thread (`Thread.currentThread()`) and use `Thread`'s `interrupt()` method to set the `Thread`'s interrupted flag back to true (throwing the `InterruptedException` set it to false).

Any process later looking at the thread will be able to detect that it was interrupted.

Executor

Now that we have a **Runnable** object – `Task` – to give to our `Executor` to run, let's instantiate those objects in our `Executor` program

```
Task task1 = new Task("task1");
Task task2 = new Task("task2");
Task task3 = new Task("task3");
```

And then get our pool of threads…

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

Executor

```
ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(task1);
executorService.execute(task2);
executorService.execute(task3);
```

`execute` will execute its `Runnable` argument some time in the future.

The task may execute in one of the threads in the `ExecutorService`'s thread pool or in a new thread or in the thread that called the `execute` method.

`ExecutorService` manages those details.

Executor

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

```
executorService.execute(task1);
executorService.execute(task2);
executorService.execute(task3);
```

`execute` returns immediately from each invocation – the program does not wait for each `Task` to finish.

Output - ConcurrencyDemo (run)

```
ant -f C:\\Users\\frenc\\Documents\\NetBeansProjects\\ConcurrencyDemo -Dnb.internal.action.name=run run
init:
Deleting: C:\Users\frenc\Documents\NetBeansProjects\ConcurrencyDemo\build\built-jar.properties
deps-jar:
Updating property file: C:\Users\frenc\Documents\NetBeansProjects\ConcurrencyDemo\build\built-jar.properties
compile:
run:
```

ding ConcurrencyDemo (run)... ConcurrencyDemo (run) 22:30 INS Windows (CR...

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool()
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to `execute` will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using `ThreadPoolExecutor` constructors.

Returns:

the newly created thread pool

```java
public static void main(String[] args)
{
    Task task1 = new Task("task1");
    Task task2 = new Task("task2");
    Task task3 = new Task("task3");

    System.out.println("Starting Executor");

    ExecutorService executorService = Executors.newCachedThreadPool();

    executorService.execute(task1);
    executorService.execute(task2);
    executorService.execute(task3);

    executorService.shutdown();

    System.out.printf("Tasks started, main ends.");
}
```

Tells `ExecutorService` to stop accepting new tasks but continue to execute tasks that have already been submitted.

Once all previously submitted **`Runnable`**s complete, `ExecutorService` terminates allowing the program to complete.

```
Starting Executor
Tasks started, main ends.

task1 is going to sleep for 2234 milliseconds

task2 is going to sleep for 17 milliseconds

task3 is going to sleep for 3407 milliseconds
task2 is done sleeping
task1 is done sleeping
task3 is done sleeping
BUILD SUCCESSFUL (total time: 4 seconds)
```

`main` Thread

The code in main() executes in the main thread which is created by the JVM.

The code in the **run** method of `Task` executes whenever the `Executor` starts each `Task` (not when each `Task` is instantiated).

When `main` terminates, the program itself continues running because there are still tasks that must finish executing.

The program will not terminate until these tasks are complete.

Starting Executor
Tasks started, main ends.
task2 is going to sleep for 2700 milliseconds
task1 is going to sleep for 413 milliseconds
task3 is going to sleep for 4957 milliseconds
task1 is done sleeping
task2 is done sleeping
task3 is done sleeping

Starting Executor
Tasks started, main ends.
task1 is going to sleep for 3214 millisec
task3 is going to sleep for 2796 millisec
task2 is going to sleep for 1897 millisec
task2 is done sleeping
task3 is done sleeping
task1 is done sleeping

Starting Executor
Tasks started, main ends.
task3 is going to sleep for 339 milliseconds
task2 is going to sleep for 3985 milliseconds
task1 is going to sleep for 1647 milliseconds
task3 is done sleeping
task1 is done sleeping
task2 is done sleeping

Starting Executor
Tasks started, main ends.
task1 is going to sleep for 1917 milliseconds
task2 is going to sleep for 3446 milliseconds
task3 is going to sleep for 1534 milliseconds
task3 is done sleeping
task1 is done sleeping
task2 is done sleeping

We cannot predict the order in which the tasks will start executing even if we know the order in which they were created and started.

Thread Synchronization

When multiple threads share an object and it's modified by one or more threads, indeterminate results may occur.

Access to a shared object must be managed properly.

If one thread is in the process of updating a shared object and another thread also tries to update it, it's uncertain which thread's update takes effect.

If one thread is in the process of updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will read the old value or the new one.

Thread Synchronization

The problem can be solved by giving only one thread **exclusive access** to the code that access the shared object.

During that time, other threads wanting to access the object are kept waiting.

When the thread with **exclusive access** finishes accessing the object, one of the waiting threads is allowed to proceed.

This process is called ***thread synchronization***.

By synchronizing threads, you can ensure that each thread accessing a shared object excludes all other threads from having access.

This is called ***mutual exclusion***.

```
public class DollarThread implements Runnable
{
    private final String name;

    public DollarThread(String taskName)
    {
        name = taskName;
    }
```

```java
public void run()
{
    Random rn = new Random();

    try
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.printf("$%s$\n", Thread.currentThread().getName());
            Thread.sleep(rn.nextInt(50));
        }
        System.out.println();
    }
    catch (InterruptedException e)
    {
        Thread.currentThread().interrupt();
    }

    System.out.printf("%s is done sleeping\n", name);
}
```

```
DollarThread task1 = new DollarThread("task1");
DollarThread task2 = new DollarThread("task2");
DollarThread task3 = new DollarThread("task3");
```

Each task is writing its name to `stdout`.

The thread writes out its name and then goes to sleep.

Do other threads get access to `stdout` while a given thread is asleep (but still executing its for loop)?

```
Starting Executor
Tasks started, main ends.
$pool-1-thread-2$
$pool-1-thread-1$
$pool-1-thread-3$
$pool-1-thread-2$
$pool-1-thread-2$
$pool-1-thread-1$
$pool-1-thread-1$
$pool-1-thread-3$
$pool-1-thread-2$
$pool-1-thread-3$
$pool-1-thread-3$
$pool-1-thread-1$
$pool-1-thread-3$
$pool-1-thread-2$
$pool-1-thread-1$
task2 is done sleeping
task3 is done sleeping
task1 is done sleeping
```

```
DollarThread task1 = new DollarThread("task1");
DollarThread task2 = new DollarThread("task2");
DollarThread task3 = new DollarThread("task3");
AsterickThread task4 = new AsterickThread("task4");
AsterickThread task5 = new AsterickThread("task5");
AsterickThread task6 = new AsterickThread("task6");
```

`DollarThread` now just prints a $ and `AsterickThread` just prints an * to `stdout`

What will the output look like?

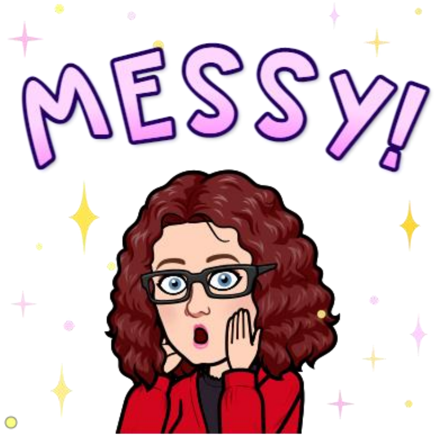
$ * * * $ $ $ * * $ $ * $ * * $ * $ $ $ * * * $ * * $ $ $ *

$ $ $ * * * * * $ * $ * $ $ * * * $ $ * $ $ $ $ * $ * * $ *

$ * * $ * $ $ $ $ * * $ * $ * * * $ $ * * * $ $ $ $ * * $ *

$ * * $ * $ * * * $ * * $ * * $ * $ $ * $ $ * * $ $ $ $ $ *

Can you see why file reading and writing (for example) would be "indeterminate"?



Threads

Threads don't do a very good job of sharing a resource.

`AsteriskThread` and `DollarThread` were sharing the standard output stream and were not able to evenly or consistently take turns.

What if those two threads had been sharing a file and were tasked with updating that file?

The data in the file would be in a different order every time the process ran.

Monitors

A common way to perform synchronization is to use Java's builtin monitors.

Every object has a monitor and a monitor lock.

The monitor ensures that its object's lock is held by a maximum of only one thread at any time.

Monitors and monitor locks can enforce mutual exclusion.

Monitors

If an operation requires the executing thread to hold a lock while the operation is performed, then a thread must acquire the lock before the operation.

Other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock.

One of the blocked threads will then acquire its own lock.

Monitors

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a **`synchronized`** statement.

```
synchronized (object)
{
   statements that need mutual exclusion
}
```

```
try
{
    synchronized (System.out)
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.print("$");
            Thread.sleep(rn.nextInt(50));
        }
    }
}
```

```
try
{
    synchronized (System.out)
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.print("*");
            Thread.sleep(rn.nextInt(50));
        }
    }

}
```

**********$$$$$*****$$$$$$$$$$

$$$$$*****$$$$$**********$$$$$

$$$$$*****$$$$$**********$$$$$

**********$$$$$$$$$$*****$$$$$

```java
public class AlphabetThread implements Runnable
{
    private char letter;

    public AlphabetThread(char letter)
    {
        this.letter = letter;
    }

    public void run()
    {
        System.out.print(letter);
        System.out.print(letter);
    }
}
```

```java
public static void main(String[] args)
{
    ArrayList <AlphabetThread> AT = new ArrayList<>();

    for (int i = 0; i < 26; i++)
    {
        AT.add(new AlphabetThread((char)(i + 65)));
    }
```

```java
ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 26; i++)
{
    executorService.execute(AT.get(i));
}

executorService.shutdown();
```

AADDEECCBBGGFFHHIIQQMMLLKKUUVVRRTTSSJJPPNNOOZZWWXXYY

AADDBEECCBFFGGINOOKKHHNIJJMLLVVMTSSRQRTXXQYYPPZZUUWW

BBDCCAAEEDJJNNFIHHGGSSOOURRMLLKTTKXMUVPPIQFQWWZZVXYY

AAEEDDBBCFFCGGHHIILRRLSSQQPPXXOOMKKNNMTVUUJVTWJWZZYY

```
public void run()
{
    synchronized (System.out)
    {
        System.out.print(letter);
        System.out.print(letter);
    }
}
```

```
AAEEDDCCBBFFGGHHIIJJPPSSOOLLTTRRNNQQMMKKWWYYUUVVZZXX
BBEEDDCCAAHHGGFFIILLQQPPOOJJNNUUMMKKZZXXWWVVTTSSRRYY
CCDDBBAAFFEEGGHHJJOOQQPPNNMMIIKKLLRRZZXXYYVVUUTTWWSS
BBEEDDCCAAHHFFMMQQSSUUNNPPZZTTYYKKGGJJWWIIVVLLRROOXX
```

```java
public class NumberThread implements Runnable
{
    private int number;
    static int counter = 1;

    public NumberThread(int number)
    {
        this.number = number;
    }

    public void run()
    {
        System.out.printf("%d-", number);

        if (counter++ % 10 ==  0)
        {
            System.out.println();
        }
    }
}
```

```java
public static void main(String[] args)
{
    ArrayList <NumberThread> NT = new ArrayList<>();

    for (int i = 10; i < 100; i++)
    {
        NT.add(new NumberThread(i));
    }

    System.out.println("Starting Executor");

    ExecutorService executorService = Executors.newCachedThreadPool();

    for (int i = 0; i < 90; i++)
    {
        executorService.execute(NT.get(i));
    }

    executorService.shutdown();
}
```

11-99-98-97-96-95-94-93-92-91-
90-89-88-87-86-85-84-83-82-81-
80-79-78-77-76-75-74-73-72-71-
70-69-68-67-66-65-64-63-62-61-
60-59-58-57-56-55-54-53-52-51-
50-49-48-47-46-45-44-43-42-41-
40-39-38-37-36-35-34-33-32-31-
30-29-28-27-26-25-24-23-22-21-
20-19-18-17-16-15-14-13-12-10-

12-99-98-97-95-96-92-94-93-91-
89-90-88-85-86-87-84-83-82-81-
80-79-78-77-76-75-74-73-72-71-
70-69-68-67-66-65-64-63-62-61-
60-59-58-57-56-55-54-53-52-51-
50-49-48-47-46-45-44-43-42-41-
40-39-38-37-36-35-34-33-32-31-
30-29-28-27-26-25-24-23-22-21-
20-19-18-17-16-15-14-13-11-10-

10-99-98-97-96-89-92-85-94-93-87-90-95-86-88-91-84-83-82-81-
80-79-78-77-76-75-74-73-72-71-
70-69-68-67-66-65-64-63-62-61-
60-59-58-57-56-55-54-53-52-51-
50-49-48-47-46-45-44-43-42-41-
40-39-38-37-36-35-34-33-32-31-30-29-28-27-26-25-24-20-23-22-
21-16-17-18-19-15-14-13-12-11-

```java
public void run()
{
    System.out.printf("%d-", number);

    if (counter++ % 10 ==  0)
    {
        System.out.println();
    }
}
```

```
1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
```

```java
public void run()
{
    System.out.printf("%d-", counter);

    if (counter++ % 10 == 0)
    {
        System.out.println();
    }
}
```

```
1-1-1-1-1-1-1-8-1-1-10-1-1-1-1-1-1-18-1-1-
1-22-23-1-1-1-1-28-29-1-
1-31-1-1-1-35-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-
1-38-37-36-36-35-34-34-33-33-
32-32-32-31-27-24-24-22-22-18-
16-16-13-13-
11-10-9-9-8-8-
```

```java
public void run()
{
    System.out.printf("%d-", counter);

    if (counter++ % 10 == 0)
    {
        System.out.println();
    }
}
```

```
1-2-3-4-5-6-7-8-9-10-
11-12-13-14-15-16-17-18-19-20-
21-22-23-24-25-26-27-28-29-30-
31-32-33-34-35-36-37-38-39-40-
41-42-43-44-45-46-47-48-49-50-
51-52-53-54-55-56-57-58-59-60-
61-62-63-64-65-66-67-68-69-70-
71-72-73-74-75-76-77-78-79-80-
81-82-83-84-85-86-87-88-89-90-
```

```java
public void run()
{
    synchronized (System.out)
    {
        System.out.printf("%d-", counter);

        if (counter++ % 10 == 0)
        {
            System.out.println();
        }
    }
}
```

```
1-2-3-4-5-6-7-8-9-10-
11-12-13-14-15-16-17-18-19-20-
21-22-23-24-25-26-27-28-29-30-
31-32-33-34-35-36-37-38-39-40-
41-42-43-44-45-46-47-48-49-50-
51-52-53-54-55-56-57-58-59-60-
61-62-63-64-65-66-67-68-69-70-
71-72-73-74-75-76-77-78-79-80-
81-82-83-84-85-86-87-88-89-90-
```