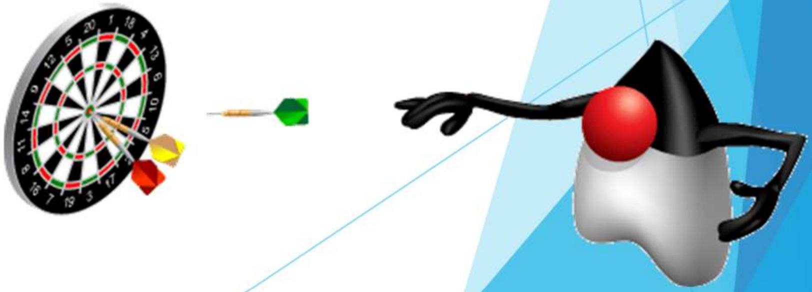
Multithreading

Objectives

At the end of this session, you will be able to

- Identify the need for multi threading
- Write simple programs using Multithreading
- Write multi threaded programs with Thread Synchronization



Agenda

- Introduction to Multithreading
- Implementing Multithreading
- Thread Life Cycle
- Using *sleep(), yield(), join()*
- Thread priorities
- Thread Synchronization
- Using *wait()* & *notify()*
- Daemon threads

Multitasking

- All modern operating systems allow the execution of concurrent tasks.
- This simultaneity can be real if the computer has more than one processor or a multi-core processor, or apparent if the computer has only one core processor.
- Two types of Multitasking:
  - Thread based
  - Process based

Process-based Multitasking

- A Process is a program under execution
- Process based multitasking allows to execute two or more programs concurrently
- In process based multitasking, a *program* is the smallest unit of code that can be dispatched by the scheduler
- Process is a heavy weight component which needs its own address space

Eg. You can read your e-mails while you listen to music in a computer

Thread based Multitasking

- Thread based Multi Tasking is Multithreading

- In thread based multitasking, a *thread* is a basic processing unit to which an operating system allocates processor time

- More than one thread can be executing code inside a process

What is a Thread?

- A Thread is an independent, concurrent path of execution through a program

- Threading is a facility to allow multiple activities to execute simultaneously within a single process

- Threads execute within the context of a process and shares the same resources allotted to the process by the kernel

- Sometimes referred to as lightweight processes

- A processor executes threads, not processes, so each application has at least one process, and a process always has at least one thread of execution, known as the primary thread or the main thread

Multithreading Example

- Consider your basic word processor
  - You have just written a large amount of text in MS Word editor and now hit the save button
  - It takes a noticeable amount of time to save new data to disk, this is all done with a separate thread in the background
  - Without threads, the application would appear to hang while you are saving the file and be unresponsive until the save operation is complete

Why Multithreading?

- Make the UI more responsive
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities
- Perform asynchronous or background processing

Multithreading in Java

- Java has excellent support for developing multithreaded applications.

- A developer can implement threads in his/her application without being aware of the lower level implementation details.

- Threads are objects

- 2 ways of creating threads:
  - *Runnable Interface*
  - *Thread Class*

Using Runnable interface

- Implement Runnable on your class.

```
class MyThreadClass implements Runnable{…}
```

- Keep the code to be executed by thread in void run()

```
public void run(){ // code to be executed comes here }
```

- Create an object of Thread class and pass a Runnable object (an object of your class) to it’s constructor.

```
Thread myThread = new Thread( new MyThreadClass() );
```

- Invoke start method on this object of Thread Class

```
myThread.start();
```

- Calling start() method twice will throw IllegalThreadStateException

Using Thread class

- Extend your class with Thread Class

```
class myThreadClass extends Thread{…}
```

- Override the run() method of Thread Class

```
public void run(){
    //Code to be executed by the thread
}
```

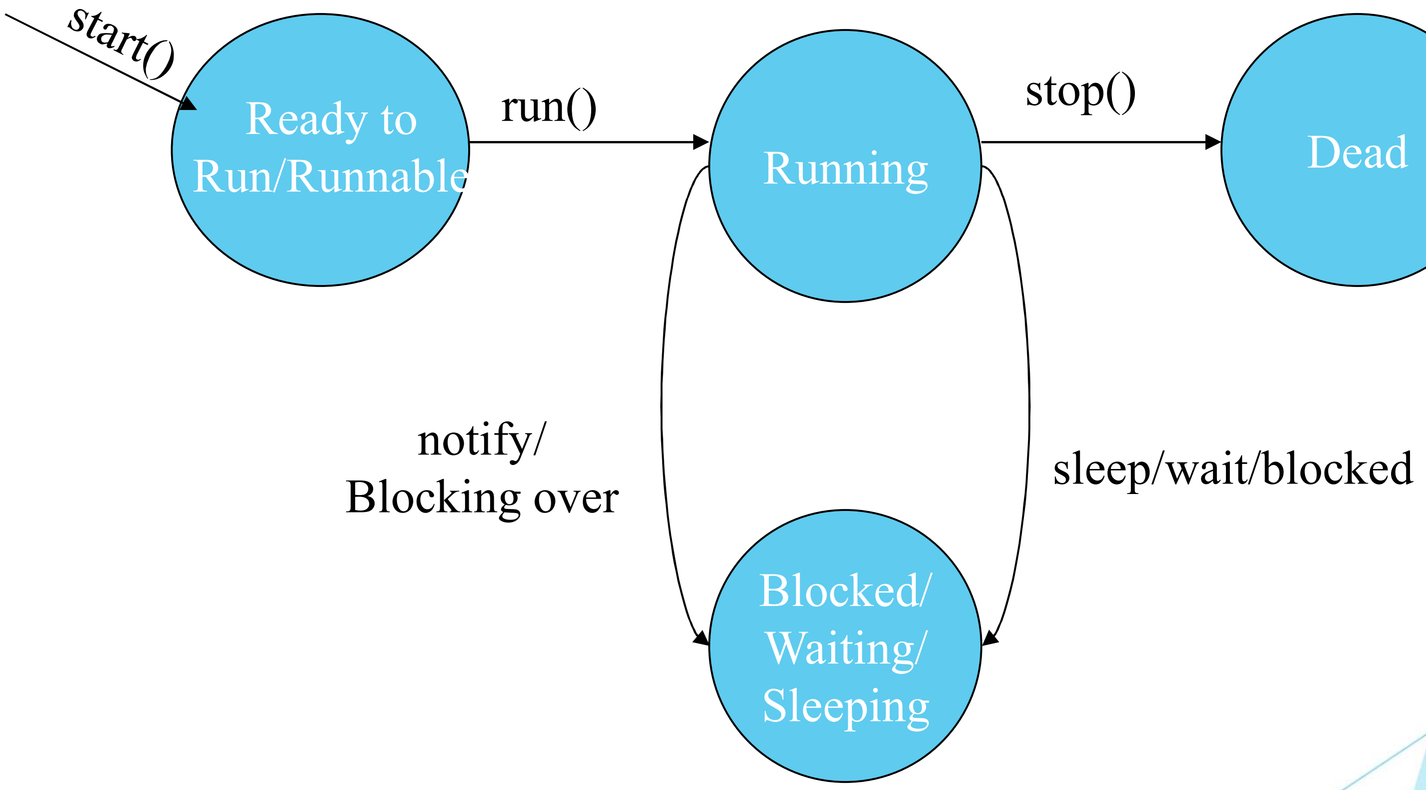
- Create an object of your class (to initialize the thread by invoking the superclass constructor)

```
myThreadClass myThread = new myThreadClass();
```

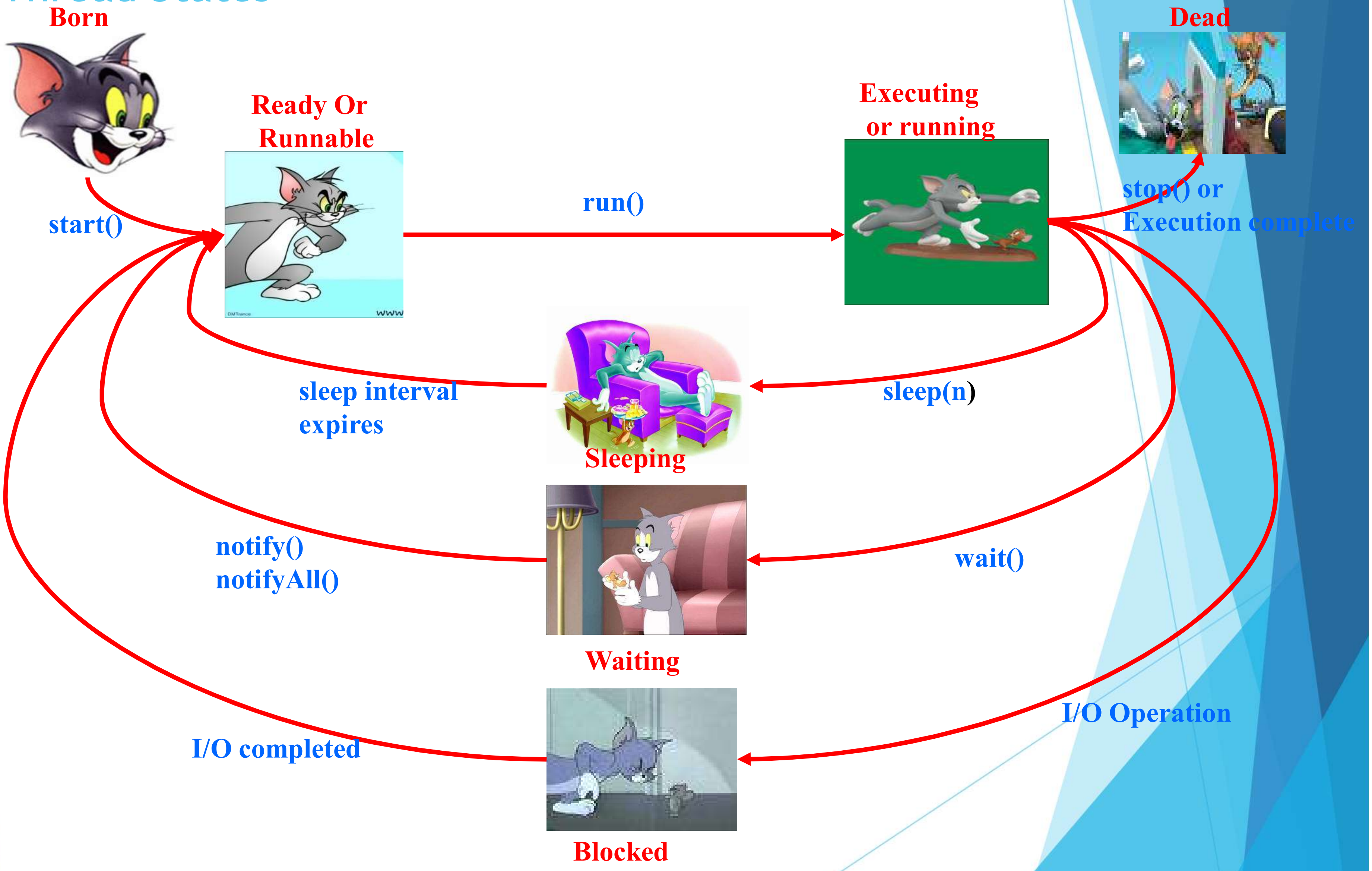
- Invoke the inherited start() method which makes the thread eligible for running

```
myThread.start();
```

Thread Life Cycle



Thread States



Born

start()

Ready Or Runnable

run()

Executing or running

stop() or Execution complete

Dead

sleep(n)

Sleeping

sleep interval expires

wait()

Waiting

notify()
notifyAll()

I/O Operation

Blocked

I/O completed

Using *sleep()*, *yield()*

- Once a thread gains control of the CPU, it will execute until one of the following occurs:
  - Its *run()* method exits
  - A higher priority thread becomes *runnable* & pre-empts it
  - Its time slice is up (on a system that supports time slicing)
  - It calls *sleep()* or *yield()*

yield()	the current thread paused its execution temporarily and has allowed other threads to execute
sleep()	the thread sleeps for the specified number of milliseconds, during which time any other thread can use the CPU

sleep() method

- Thread.sleep() is a static method. It delays the executing thread for a specified period of time (milliseconds or milliseconds plus nanoseconds)

- Thread.sleep() throws a checked InterruptedException if it is interrupted by another Thread.

```
try {
   Thread.sleep ( 1000 );
} catch ( InterruptedException e ) {
   e.printStackTrace();
}
```

- When a thread is asleep, or otherwise blocked on input of some kind, it doesn't consume CPU time or compete with other threads for processing.

Thread Priorities

- The Thread scheduler which is part of JVM chooses which thread to run according to a “fixed priority algorithm”

- Threads with higher priorities are run to completion before Threads with lower priorities get a chance of CPU time

- The algorithm is *preemptive*, so if a lower priority thread is running, and a higher priority thread becomes runnable, the high priority thread will pre-empt the lower priority thread

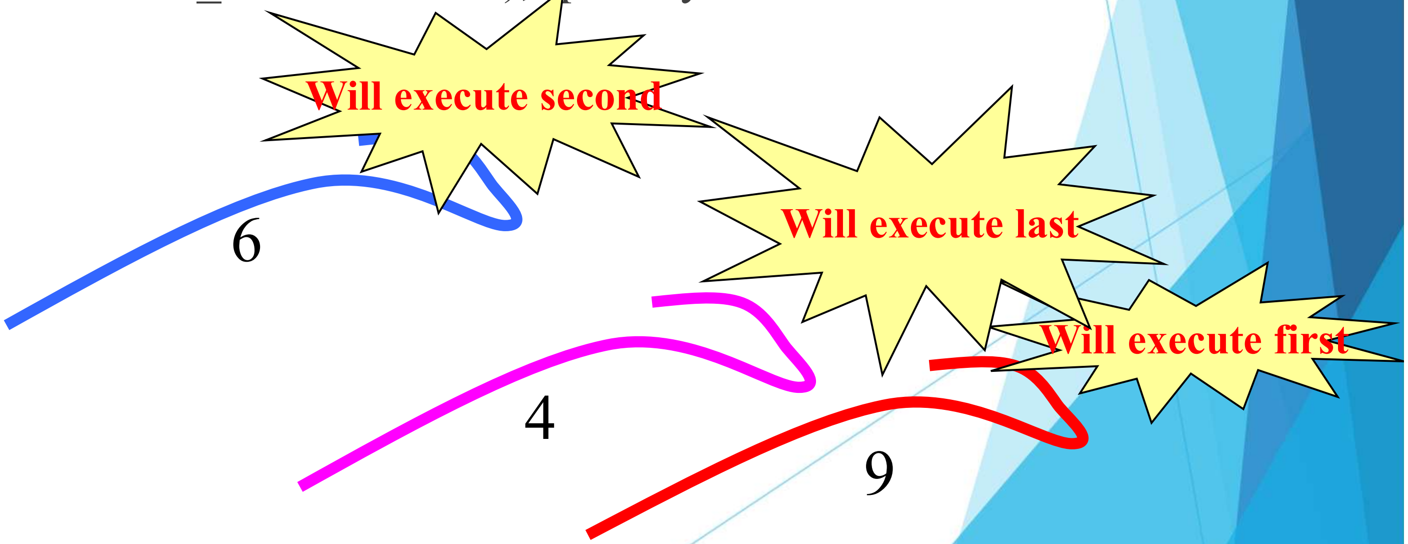
- All Java threads have a priority in the range 1-10.
  - Thread.MIN_PRIORITY - 1
  - Thread.MAX_PRIORITY - 10
  - Thread.NORM_PRIORITY - 5

Thread Priority

- When a new Java thread is created it has the same priority as the thread which created it.

- Thread priority can be changed by the setPriority() method.

```
t1.setPriority(Thread.NORM_PRIORITY + 1);  //priority 6
t2.setPriority(Thread.NORM_PRIORITY -1);//priority 4
t3.setPriority(Thread.MAX_PRIORITY - 1);//priority 9
```



```
t1.start();
t2.start();
t3.start();
}
```

Joins

- The join method allows one thread to wait for the completion of another.

- If t1 is a Thread object whose thread is currently executing, t1.join(); causes the current thread to pause execution until the thread t1 terminates.

- Join is dependent on the OS for timing, so do not assume that join will wait exactly as long as you specify.

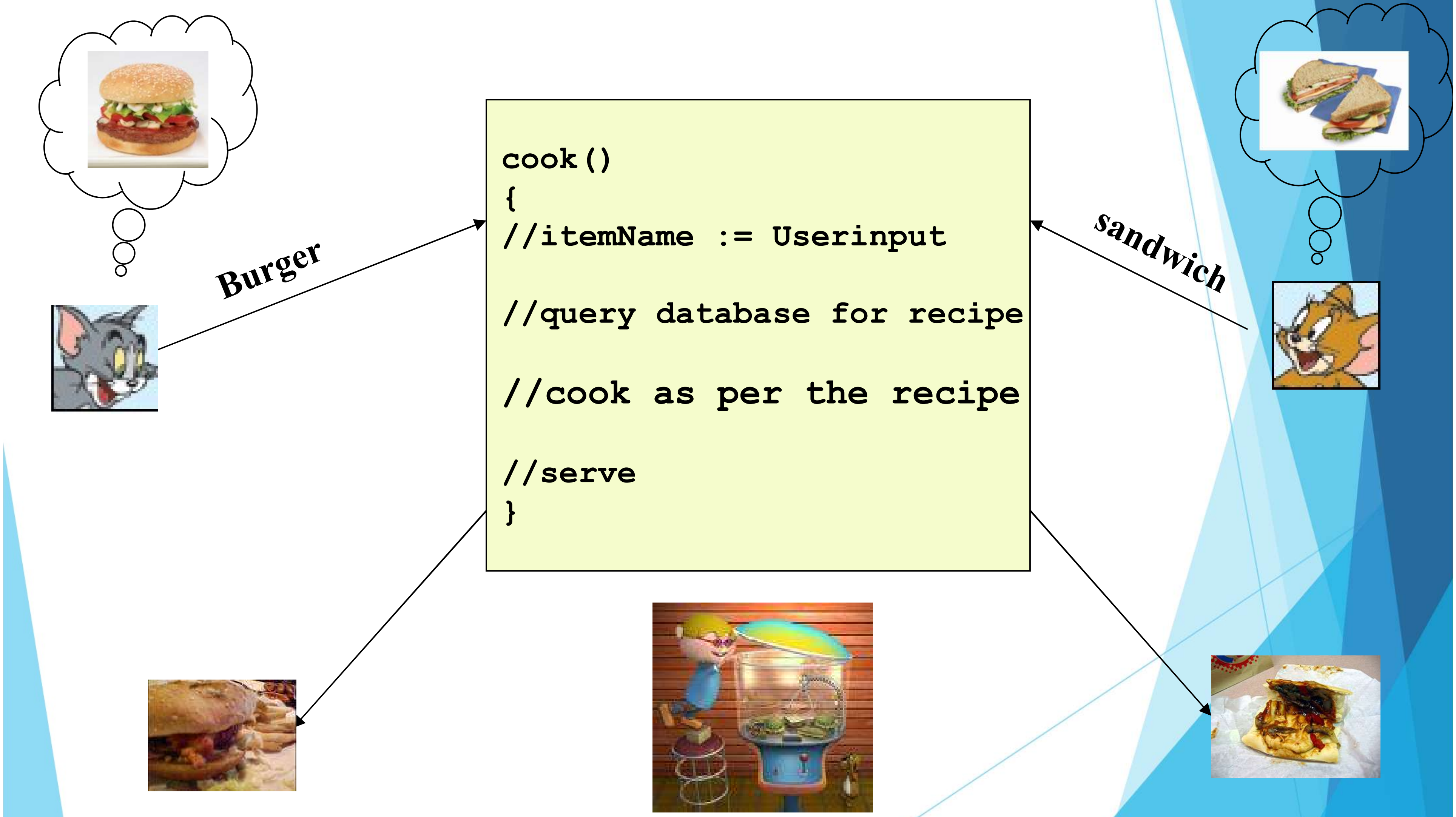
```
public class ThreadExampleMain
{
  public static void main(String[] args) {
     Thread myThread = new ThreadExample(“my data”);
    myThread.start();
    System.out.println(“I am the main thread”);

    myThread.join();
    System.out.println(“This line is printed after myThread finishes execution”);
  }
}
```

Synchronization

- Sometimes, multiple threads may be accessing the same resources concurrently
  - Reading and / or writing the same file
  - Modifying the same object / variable
- Synchronization controls thread execution order
- Synchronization eliminates data races
- Java has built in primitives to facilitate this coordination

Synchronization



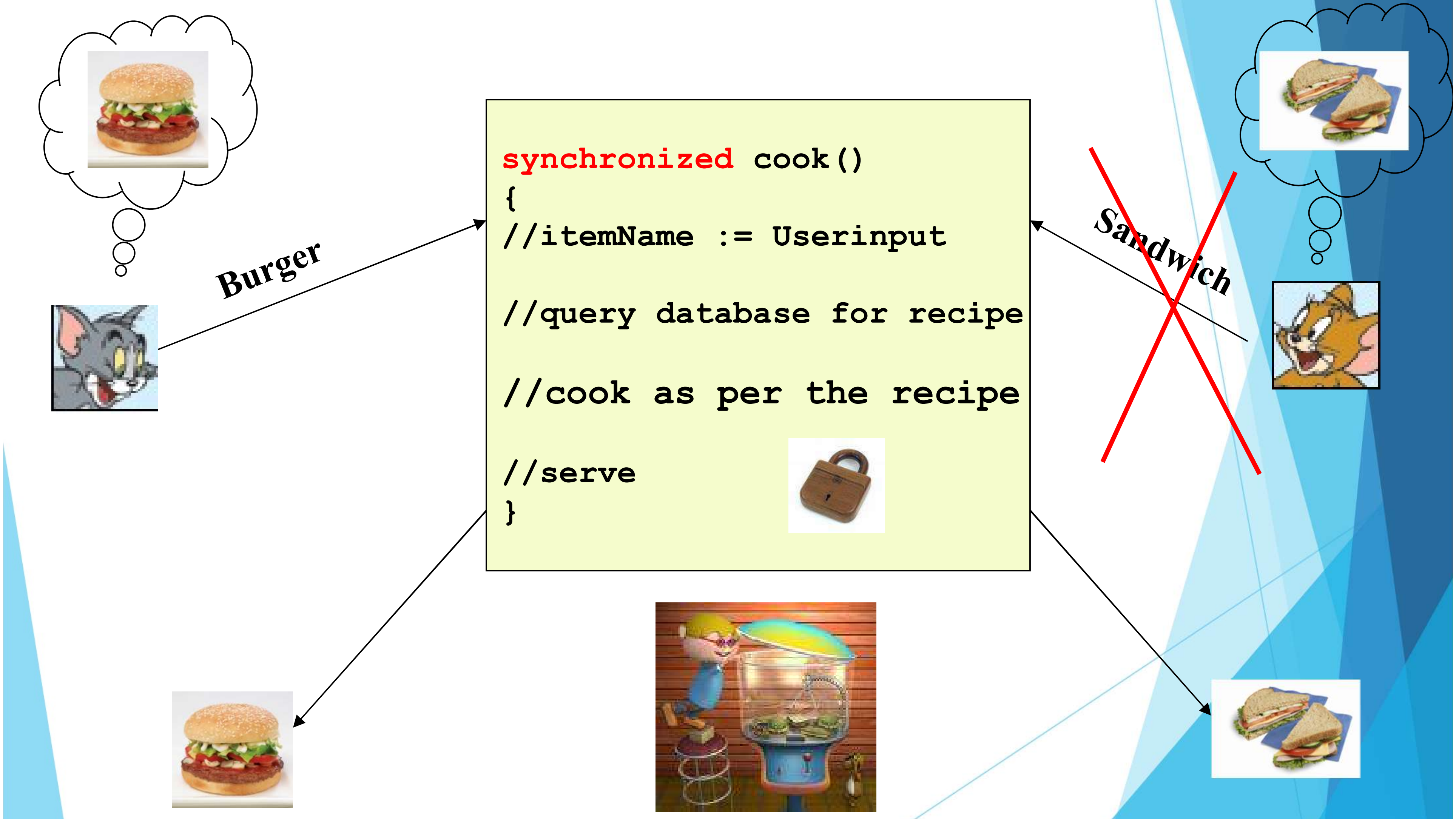
```
cook()
{
//itemName := Userinput

//query database for recipe

//cook as per the recipe

//serve
}
```

Synchronization



```
synchronized cook()
{
//itemName := Userinput

//query database for recipe

//cook as per the recipe

//serve
}
```

Synchronization

- Every object in Java has a lock.

- Using *synchronization* enables the lock and allows only one thread to access that part of code.

- Synchronization can be applied to:

  - A method

    ```
    public synchronized doStuff(){…}
    ```

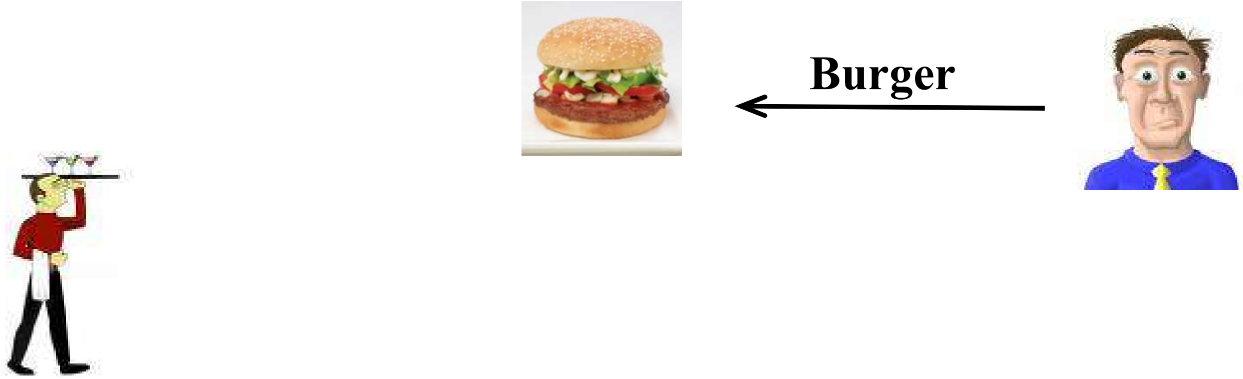
  - A block of code

    ```
    synchronized (objectReference){…}
    ```

wait() & notify()

- Threads can communicate using wait() and notify() methods of Object class.
- Both of them can be invoked only within a synchronized context
- When a thread calls the *wait()* method:
  - The thread releases the lock for the object.
  - The state of the thread is set to be blocked.
  - The thread is placed in the wait set for the object.
- When a thread calls the *notify()* method:
  - An arbitrary thread is picked from the list of threads in the wait set.
  - Moves the selected thread from the wait set to the entry set.
  - Sets the state of the selected thread from blocked to runnable.
  - notifyall() will move all waiting threads to Runnable state

wait() & notify()



```
synchronized takeOrder()
{
try{
if(!orderPrepared) {
wait();
}
orderTaken = true;
orderPrepared=false;
notify();
} catch(InterruptedException ie) {
ie.printStackTrace();
}
}
```

```
synchronized prepareOrder()
{
if(!orderTaken) {
wait();
}
orderPrepared = true;
orderTaken=false;
notify();
}catch(InterruptedException ie) {
ie.printStackTrace();
}
}
```

Thread-safe classes

- When a class has been carefully synchronized to protect its data, it is said to be thread-safe
- It is safe to use them in a multi-threaded environment
- Many classes in Java API use synchronization internally to make it thread-safe
- Eg. StringBuffer

Daemon Threads

- Low priority threads that work in the background as service providers for normal (also called user) threads running in the same program

- Executes when no other thread of the same program is running

- When daemon threads are the only threads running in a program, the JVM ends the program finishing these threads.

- By default a thread is not a daemon thread. *setDaemon(true)* turns a thread into daemon thread.

```
public final void setDaemon(boolean isDaemon);
public final boolean isDaemon();
```

- Eg: the clock handler thread, the garbage collector thread, the screen updater thread etc.

Try it out

Q1. What is the signature of run method?

Q2. Can variables and classes be synchronized as like methods and blocks?

Q3. A thread releases the locks it holds on calling wait() method but keeps the lock on calling sleep() method. (True/ False)

Q4. sleep() is a static method in________ class and wait() is a non-static method in __________class

Q5. The thread that completes its run() method moves to _________ state

Summary

- In this session, we have covered:
  - Implementing Multithreading
  - Thread Life Cycle
  - Using *sleep()*, *yield()*, *join()*
  - Thread priorities
  - Thread Synchronization
  - Using *wait()* & *notify()*
  - Daemon threads

Thank you