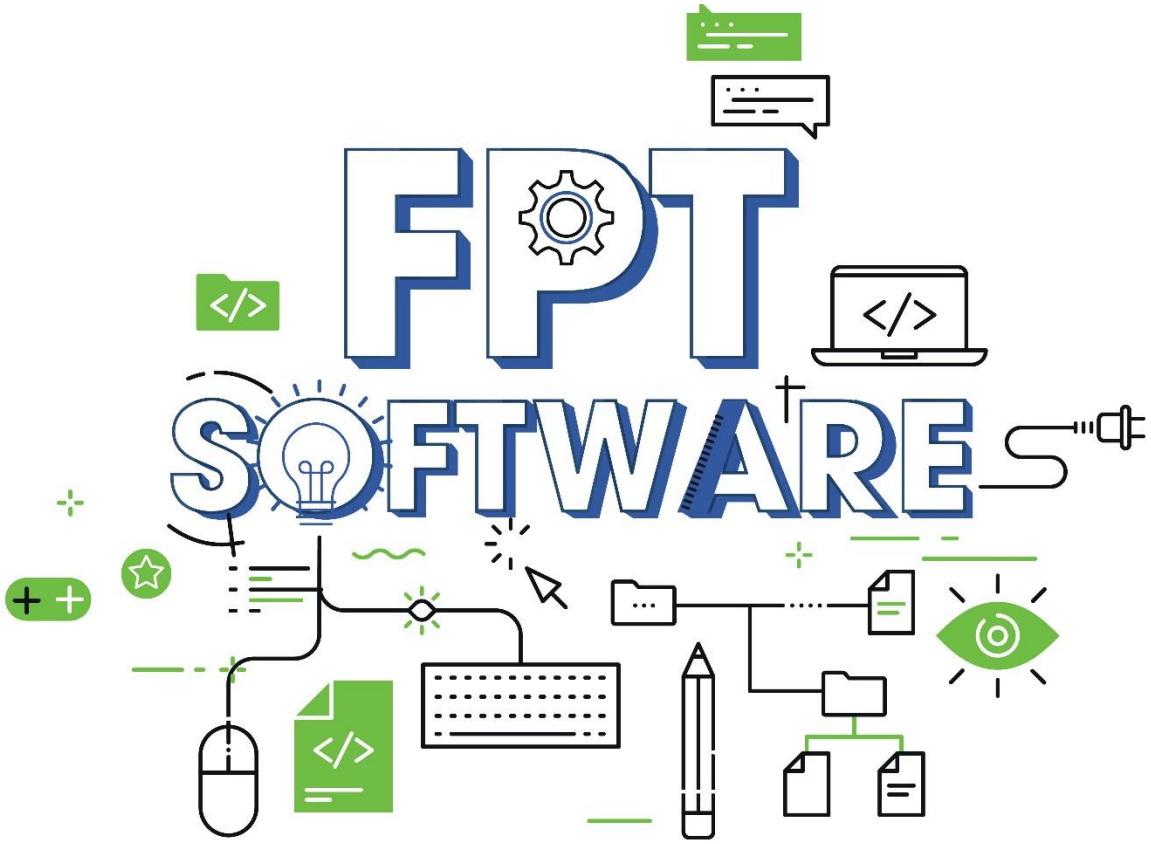
Multithread and Synchronization

Fsoft Academy



Content

- Introduction
- Thread Concepts
- Creating Tasks and Threads
- The Thread class
- Thread Pools
- Thread Synchronization
- Synchronization Using Locks
- Cooperation among Threads

Section 1

Thread Concepts

Introduction

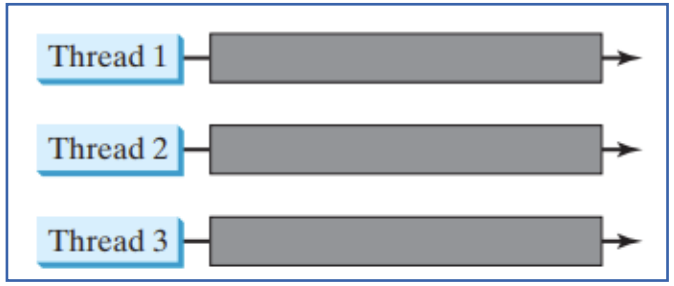
Multithreading enables multiple tasks in a program to be executed concurrently.

- One of the powerful features of Java is its built-in support for multithreading - the concurrent running of multiple tasks within a program.
- In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading.

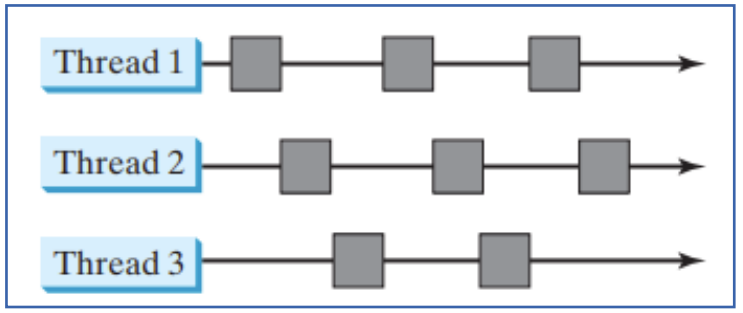
Thread Concepts

A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

- A thread provides the mechanism for running a task.
- With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multi-processor system



Multiple threads are running on multiple CPUs



Multiple threads share a single CPU

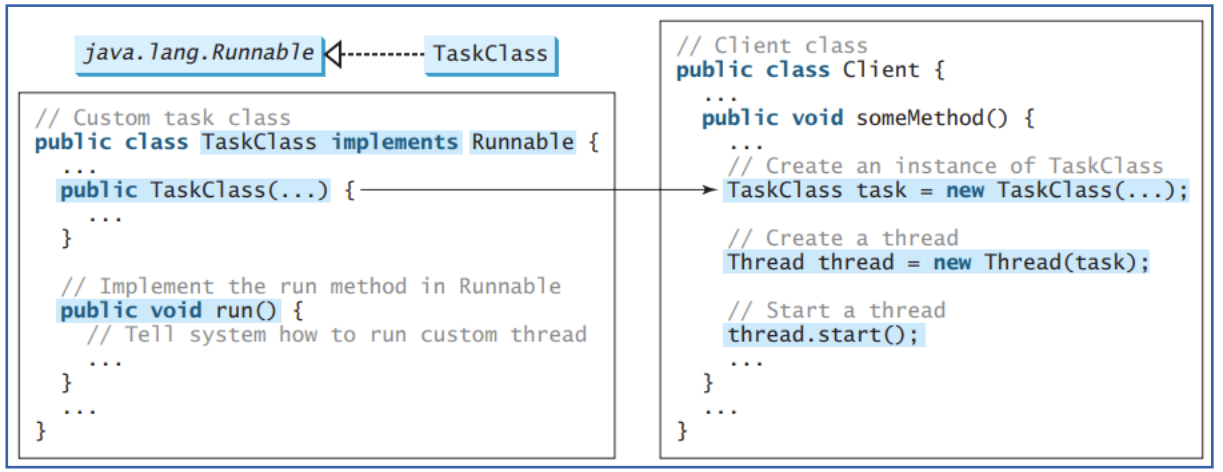
Thread Concepts

Multithreading can make your program more *responsive and interactive*, as well as enhance performance.

- **For example:** a good word processor lets you print or save a file while you are typing.
- In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems. Java provides exceptionally good support for **creating** and **running** threads and for locking resources to prevent conflicts.
- When your program executes as an application, the Java interpreter **starts a thread** for the main method.
  - ✓ You can create additional threads to run concurrent tasks in the program.
  - ✓ 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

A task class must implement the **Runnable interface**. A task must be run from a thread.



- Tasks are objects.
- To create tasks: you have to define a class for tasks, which implements the **Runnable** interface.
- The **Runnable** interface contains the run method. You need to implement this method to tell the system how your thread is going to run

Example

```
class PrintChar implements Runnable {
   private char charToPrint;
   private int times;

   public PrintChar(char c, int t) {
      charToPrint = c;
      times = t;
   }

   @Override
   public void run() {
      for (int i = 0; i < times; i++) {
         System.out.print(charToPrint);
      }
   }
```

```
class PrintNum implements Runnable {
   private int times;

   public PrintNum(int t) {
      times = t;
   }
   @Override
   public void run() {
      for (int i = 1; i <= times; i++) {
         System.out.print(i + " ");
      }
   }
}
```

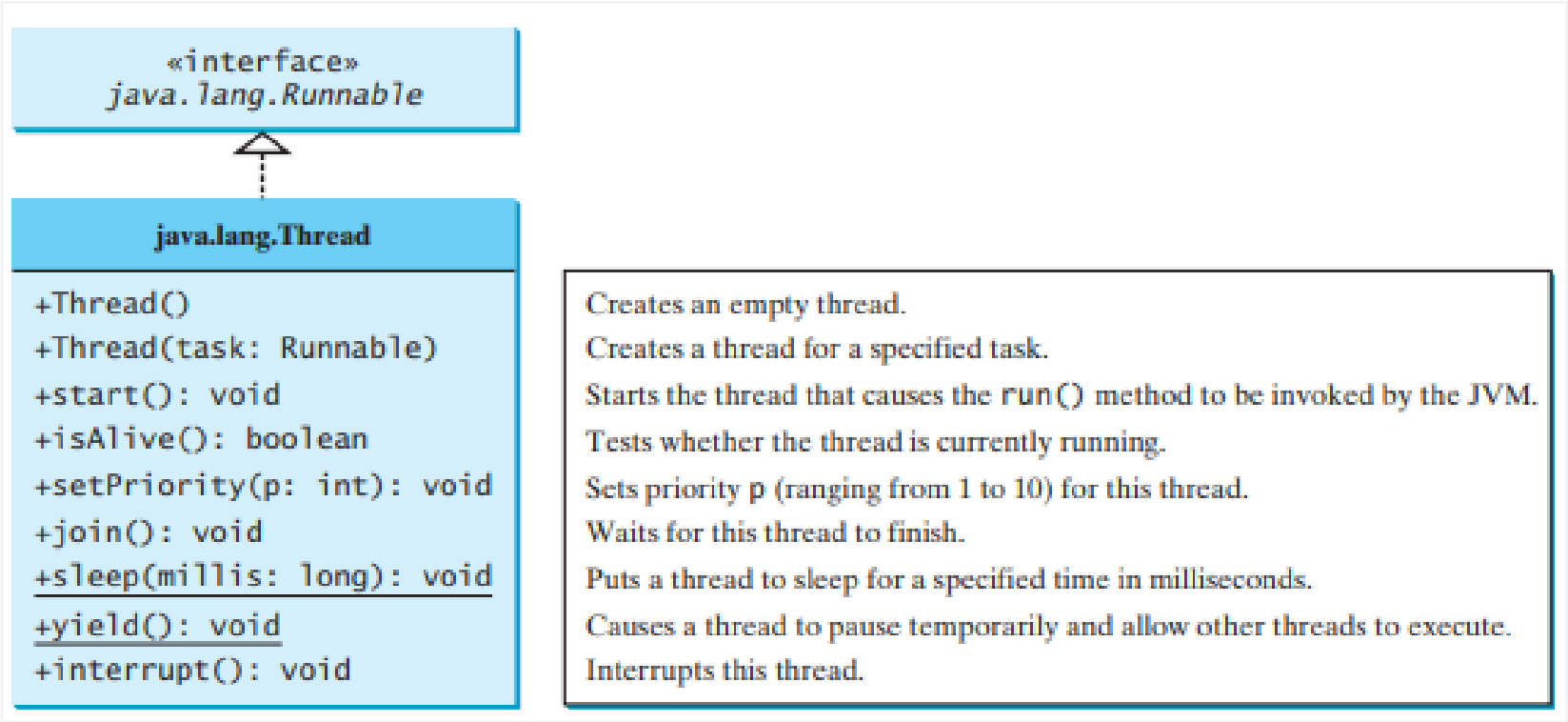
Example

```
public class TaskThreadDemo {
   public static void main(String[] args) {
      // Create tasks
      Runnable printA = new PrintChar('A', 100);
      Runnable printB = new PrintChar('B', 100);
      Runnable printNumbers = new PrintNum(100);

      // Create threads
      Thread threadA = new Thread(printA);
      Thread threadB = new Thread(printB);
      Thread threadNumbers = new Thread(printNumbers);

      // Start threads
      threadA.start();
      threadB.start();
      threadNumbers.start();
   }
}
```

If you don't see the effect of these three threads running concurrently, increase the number of characters to be printed

The Thread class

The Thread class contains the **constructors** for creating threads for tasks and the methods for controlling threads



Example

```
public class CountdownTimer {
    public static void main(String[] args) {
        Thread countdownThread = new Thread(new Countdown());
        countdownThread.start();
    }
}
```

```
class Countdown implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 10; i >= 0; i--) {
                System.out.println("Time left: " + i + " seconds");
                Thread.sleep(1000); // Sleep for 1 second (1000 milliseconds)
            }
            System.out.println("Time's up!");
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Thread Pool

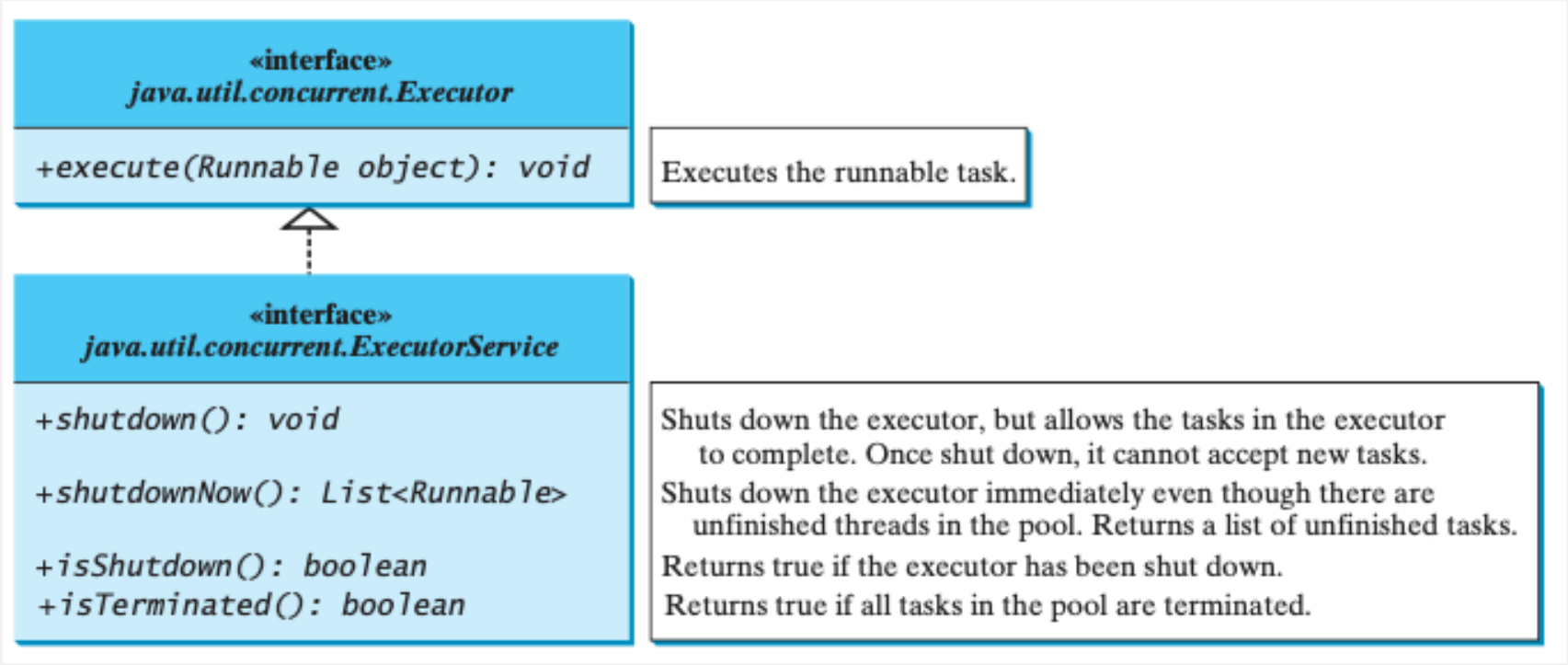
- You learned how to define a task class by implementing **java.lang.Runnable**, and how to create a thread to run a task like this:

```
Runnable task = new TaskClass(task);
new Thread(task).start();
```

- It is convenient for a single task execution, but it is not 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 throughput and cause poor performance.
- Using a **thread pool** is an ideal way to manage the number of tasks executing concurrently.

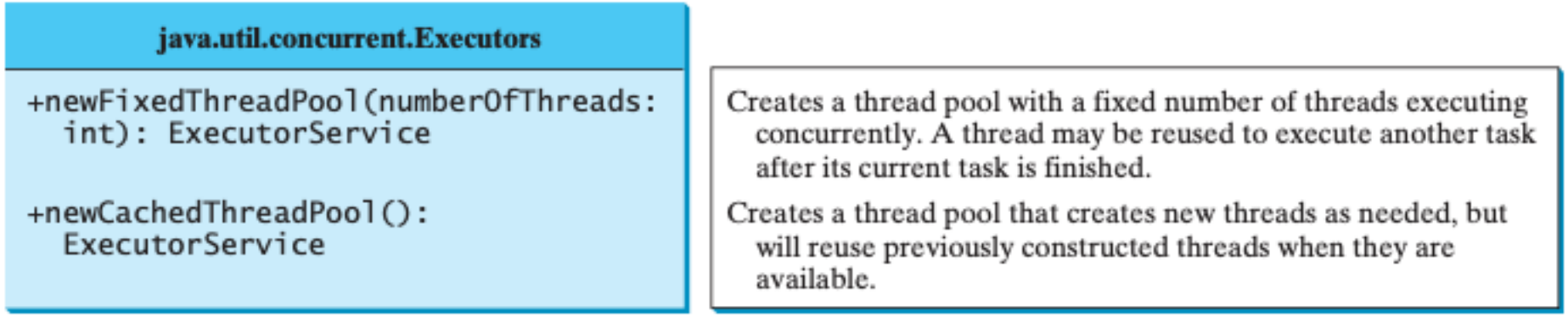
Thread Pool

- Java provides the **Executor** interface for executing tasks in a thread pool and the **ExecutorService** interface for managing and controlling tasks.
- **ExecutorService** is a sub-interface of **Executor**



Thread Pool

- To create an **Executor** object, use the static methods in the **Executors** class



- The **newFixedThreadPool**(int) method creates a fixed number of threads in a pool.
  - ✓If a thread completes executing a task, it can be reused to execute another task.
  - ✓If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution.
- The **newCachedThreadPool**() method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

Thread Pool

- If you need to create a thread for just one task, use the **Thread** class. If you need to create threads for multiple tasks, it is better to use a thread pool.

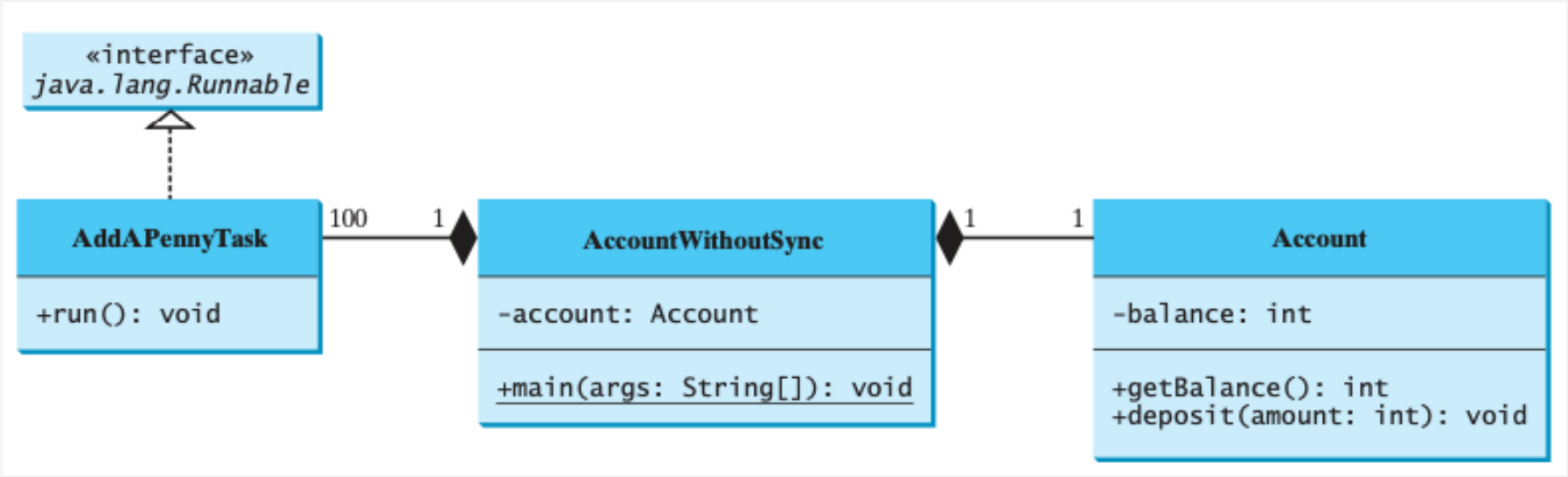
```
public class ExecutorDemo {
    public static void main(String[] args) {
        // Create a fixed thread pool with a maximum of three threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit runnable tasks to the executor
        executor.execute(new PrintChar('a', 100));
        executor.execute(new PrintChar('b', 100));
        executor.execute(new PrintNum(100));

        // Shut down the executor
        executor.shutdown();
    }
}
```

Section 2

Thread Synchronization

Thread Synchronization

- **Thread synchronization** is to coordinate the execution of the dependent threads.
  - ✓ A shared resource may become corrupted if it is accessed simultaneously by multiple threads.
- Suppose that you create and launch **100 threads**, each of which adds a penny to an account. Define a class named Account to model the account, a class named AddAPennyTask to add a penny to the account, and a main class that creates and launches threads.



Example

```
public class AccountWithoutSync {
    private static Account account = new Account();

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

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }

        executor.shutdown();

        // Wait until all tasks are finished
        while(!executor.isTerminated()){

        }
        System.out.println("What is the balance? " + account.getBalance());
    }
```

Example (cont)

```
public static class Account {
    private int balance = 0;

    public int getBalance() {
         return balance;
    }
    public void deposit(int amount) {
          int newBalance = balance + amount;
          // Simulate some processing time
        try {
           Thread.sleep(5);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
           balance = newBalance;
        }
    }
    static class AddAPennyTask implements Runnable {
        @Override
        public void run() {
            account.deposit(1);
        }
    }
}
```

What is the balance? 6

What is the balance? 7

What is the balance? 3

What is the balance? 6

What caused the error in this program?

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

Task 1 and Task 2 both add 1 to the same balance.

- In Step 1, Task 1 gets the balance from the account.
- In Step 2, Task 2 gets the same balance from the account.
- In Step 3, Task 1 writes a new balance to the account.
- In Step 4, Task 2 writes a new balance to the account.

The synchronized Keyword

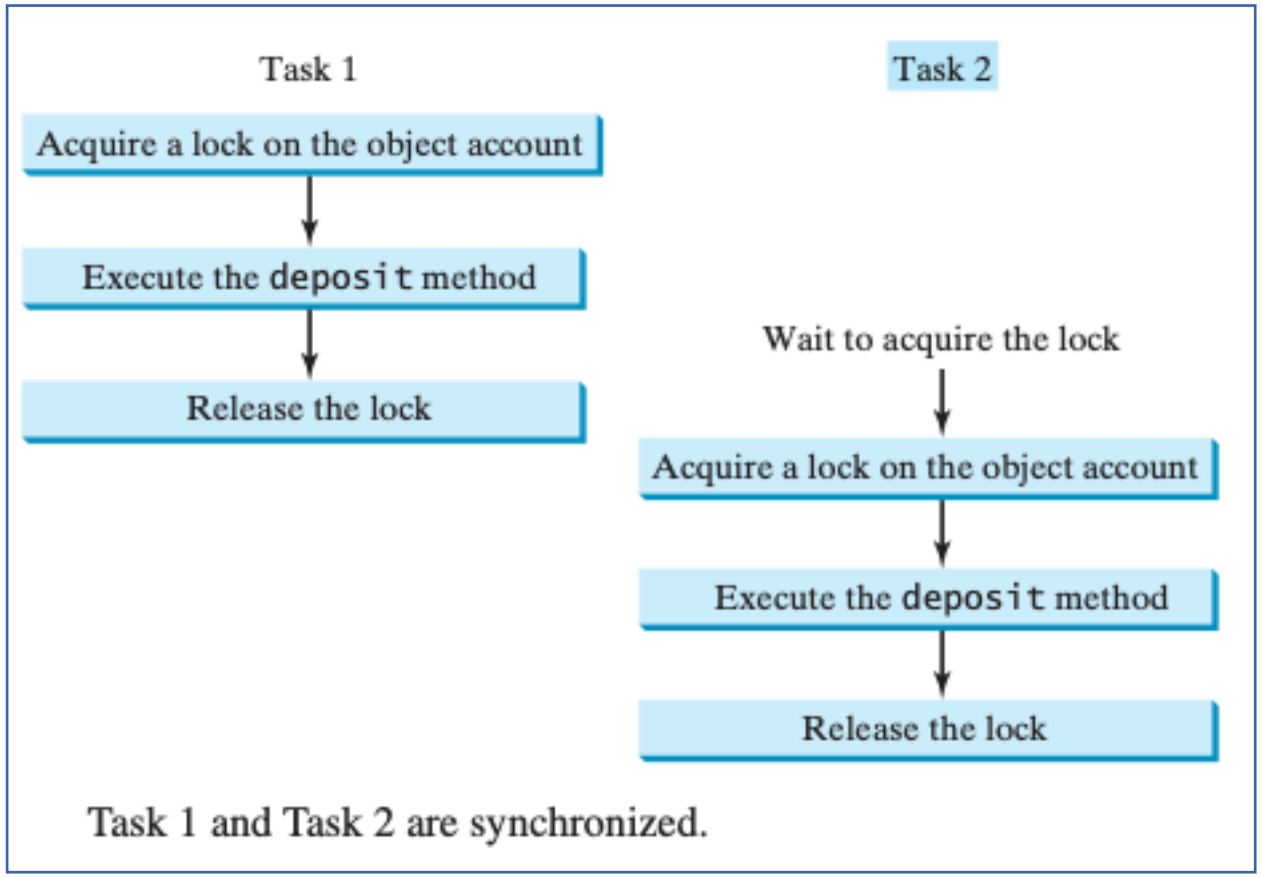
- To avoid **race conditions**, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the **critical region**.
- The critical region in the **AccountWithoutSync** class is the entire deposit method. You can use the keyword **synchronized** to synchronize the method so that only one thread can access the method at a time.

```
public synchronized void deposit(double amount)
```

The synchronized Keyword

- A synchronized method acquires a **lock** before it executes. A lock is a mechanism for exclusive use of a resource.
  - ✓ In the case of an **instance** method, the lock is on the object for which the method was invoked.
  - ✓ In the case of a **static** method, the lock is on the class.
- If one thread invokes a **synchronized instance method** (respectively, static method) on an object:
  - ✓ the lock of that object (respectively, class) is acquired **first**,
  - ✓ **then** the method is executed,
  - ✓ and **finally** the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

The synchronized Keyword



Task 1 and Task 2 are synchronized.

Synchronization Using Locks

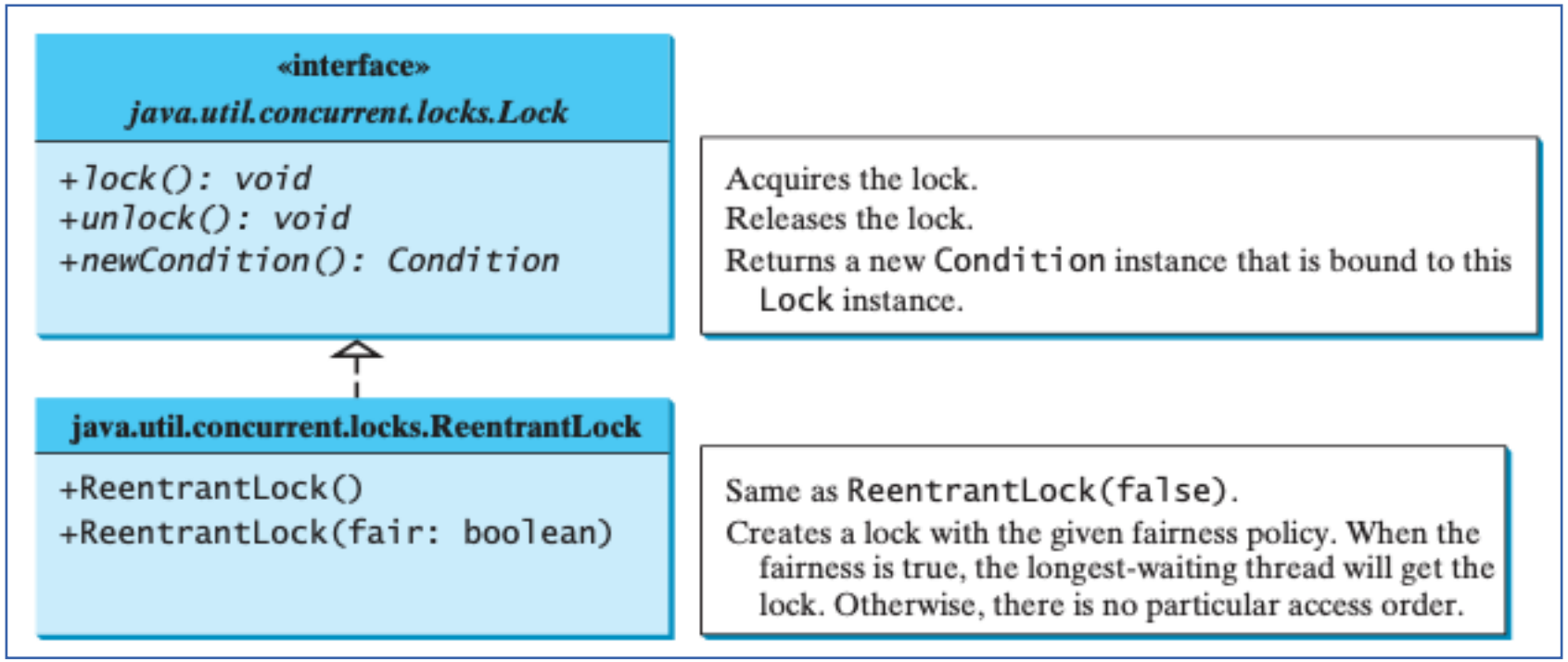
- Explicitly use locks and conditions to synchronize threads.
- In the example, 100 concurrent tasks depositing to the same account caused conflicts.
- To prevent conflicts, the deposit method uses the synchronized keyword:

public synchronized void **deposit**(double amount)

- A synchronized instance method implicitly acquires a lock on the instance.
- Explicit Locks for Thread Coordination:
  - ✓ Java allows explicit lock acquisition, providing more control for thread coordination.
  - ✓ Locks, instances of the **Lock** interface, define methods for acquiring and releasing locks.

Synchronization Using Locks

- A lock may also use the **newCondition**() method to create any number of **Condition** objects, which can be used for thread communications.



Synchronization Using Locks

- **ReentrantLock** is a concrete implementation of Lock for creating mutually exclusive locks.
- You can create a lock with the specified **fairness policy**.
  - ✓**True fairness** policies guarantee that the longest-waiting thread will obtain the lock first.
  - ✓**False fairness** policies grant a lock to a waiting thread arbitrarily.
- Programs using fair locks accessed by many threads may have poorer overall performance than those using the default setting, but they have smaller variances in times to obtain locks and prevent starvation.

Synchronization Using Locks

```
public class AccountWithSyncUsingLock {
    private static Account account = new Account();
    private static final Lock lock = new ReentrantLock();

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

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }
        executor.shutdown();
        // Wait until all tasks are finished
        while(!executor.isTerminated()){
        }
        System.out.println("What is the balance? " + account.getBalance());
    }
```

Synchronization Using Locks

```
static class AddAPennyTask
      implements Runnable {
        @Override
        public void run() {
            account.deposit(1);
        }
    }
}
```

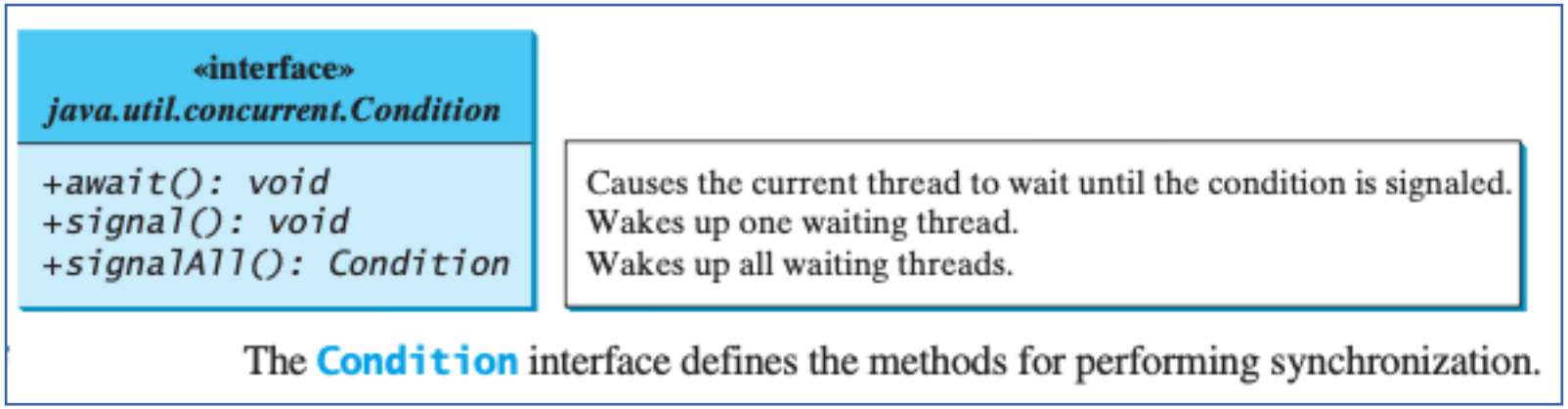
```
static class Account {
    // Create a lock
    private static Lock lock = new ReentrantLock();
    private int balance = 0;


    public int getBalance() {
       return balance;
    }


    public void deposit(int amount) {
       lock.lock();
       try {
          int newBalance = balance + amount;
          // Simulate some processing time
          Thread.sleep(5);
          balance = newBalance;
       } catch (InterruptedException e) {
          e.printStackTrace();
       } finally {
          lock.unlock();
       }
    }
}
```

Cooperation among Threads

- Synchronization prevents *race conditions* by ensuring mutual exclusion in critical regions.
- Conditions can be used to facilitate communications among threads.
- Threads can specify actions under certain conditions.
- Conditions are created using **newCondition**() on a **Lock** object.

Cooperation among Threads

- Once a condition is created, you can use its **await**(), **signal**(), and **signalAll**() methods for thread communications.
- The **await**() method causes the current thread to wait until the condition is signaled.
- The **signal**() method wakes up one waiting thread, and the signalAll() method wakes all waiting



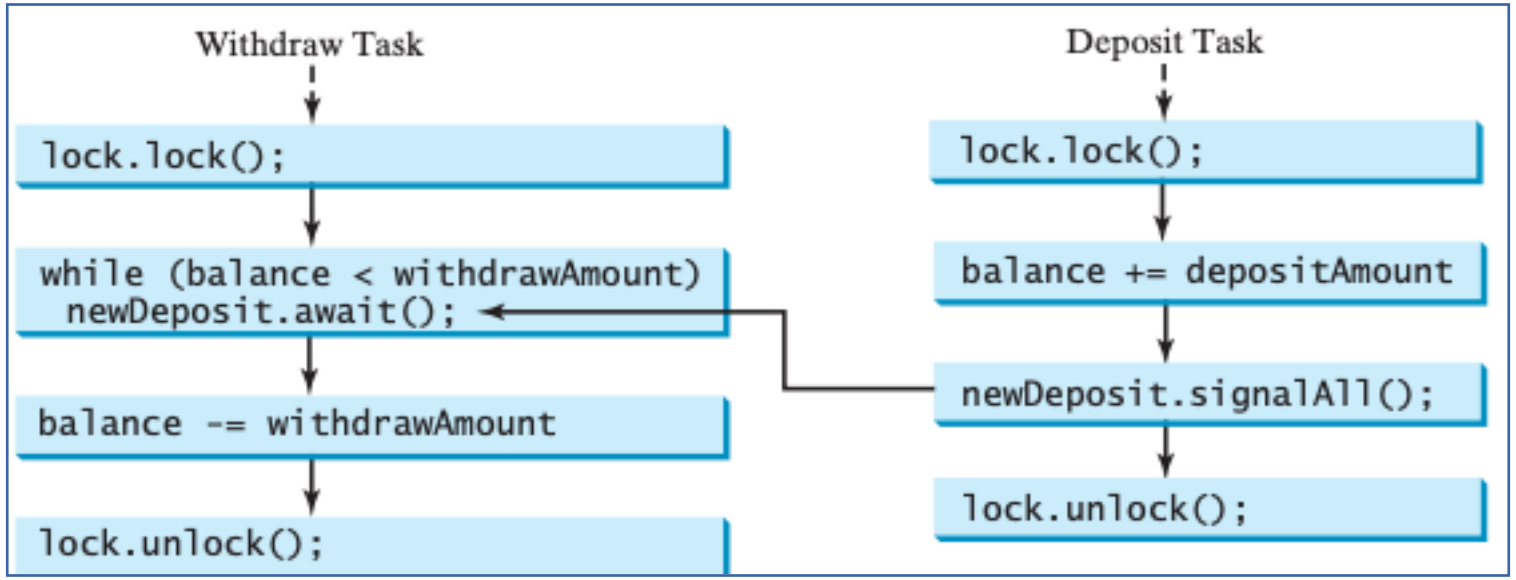
The **Condition** interface defines the methods for performing synchronization.

Example

- Suppose that you create and launch two tasks: one that deposits into an account, and one that withdraws from the same account.
  - ✓ The withdraw task has to wait if the amount to be withdrawn is more than the current balance.
  - ✓ Whenever new funds are deposited into the account, the deposit task notifies the withdraw thread to resume. If the amount is still not enough for a withdrawal, the withdraw thread has to continue to wait for a new deposit.
- To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account).
  - ✓ If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition.
  - ✓ When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.

Example

- The condition newDeposit is used for communications between the two threads.



Example

```
public class ThreadCooperationDemo {
  private  static Account account = new Account();
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WidthdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");

    try {
      // Wait until all tasks are finished
      executor.awaitTermination(1, TimeUnit.MINUTES);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("What is the balance? " + account.getBalance());
  }
  public static class DepositTask implements Runnable{
  }

public static class WidthdrawTask implements Runnable{
  }
  //An inner class for account
  private static class Account {
  }
}
```

```
public static class DepositTask implements Runnable{
  @Override
  public void run() {
    try{
      while (true){
        account.deposit((int) (Math.random() * 10) + 1);
        Thread.sleep(1000);
      }
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
public static class WidthdrawTask implements Runnable{
  @Override
  public void run() {
    while (true){
      account.withdraw((int) (Math.random() * 10) + 1);
    }
  }
}
```

Account class

```
//An inner class for account
private static class Account {
   private  static Lock lock = new ReentrantLock();
   private static Condition newDeposit = lock.newCondition();
   private int balance = 0;

   public int getBalance() {
      return balance;
   }

   public void withdraw(int amount) {
   }

   public void deposit(int amount) {
   }
}
}//end of ThreadCooperationDemo
```

```
public void withdraw(int amount) {
   lock.lock(); //acquire the lock
   // Simulate some processing time
   try {
      while(balance<amount){
         System.out.println("\t\t\tWait for a deposit");
         newDeposit.await();
      }
      balance -= amount;
      System.out.println("\t\t\tWidthdraw " + amount + "\t\t" + getBalance());

   } catch (InterruptedException e) {
      e.printStackTrace();
   }
   finally {
      lock.unlock(); //release the lock
   }

}
```

```
public void deposit(int amount) {
   lock.lock();

   try {
      balance += amount;
      System.out.println("Deposit " + amount + "\t\t\t\t\t" + getBalance());
      //Signal thread waiting on the condition
      newDeposit.signalAll();
   }
   finally {
      lock.unlock();
   }
}
```

Summary

- Task Execution in Threads:
  - Tasks are instances of the **Runnable** interface.
  - Threads, facilitating task execution, are created by implementing the Runnable interface and wrapping it using a Thread constructor.
- Starting and Controlling Threads:
  - After creating a thread object, use **start**() to initiate it.
  - Utilize **sleep**(long) to pause a thread, allowing others to run.
- Thread Execution and Overrides:
  - The run method in a task class is invoked by the JVM during execution.
  - Override the run method to specify the thread's behavior.

- Thread Safety with Synchronization:
  - Use synchronized methods or blocks to prevent thread interference.
  - Acquire locks before executing synchronized methods; for instance methods, lock is on the object, and for static methods, on the class.
- Synchronized Statements and Locks:
  - Synchronized statements can acquire locks on any object, not just "this".
  - Useful for executing synchronized blocks of code.
- Communication and Locks:
  - Utilize explicit locks and conditions for inter-thread communication.
  - The built-in monitor for objects can also be employed.

THANK YOU!

Any questions?