Java Update Part I

Language Updates in Java 7/8

Contents

- New Java 8 Date/Time API
- Java 7: Updates to Exception management
- Refresher Java 6/7:
  - Collections
  - Generics
  - Threads
- Java 7 Concurrency API
- Java 8 updates to Concurrency API
- Quiz

Date/Time API - Instant and Duration

- Java Time counts 86,400 seconds per day (no leap second)
- In Java, an Instant represents a point on the time line.
  - epoch is arbitrarily set at midnight of January 1, 1970 at the prime meridian that passes through the Greenwich Royal Observatory in London.
- Instant.now(); Duration.between();
- You can get the length of a Duration in conventional units by calling toNanos, toMillis, toSeconds, toMinutes, toHours, or toDays.

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Date/Time API - Arithmetics on Instant and Duration

- Instant and Duration are *immutable:* operations return new instances

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
// Or timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```

Date/Time API - Local Dates

- There are two kinds of human time in the new Java API, local date/time and zoned time. Local date/time has a date and/or time of day, but no associated time zone information.
- Do not use zoned time unless you really want to represent absolute time instances. Use Local date and time for schedule times.

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
// Uses the Month enumeration
```

Date/Time API - Date Adjusters

- For scheduling applications, you often need to compute dates such as “the first Tuesday of every month.”
  - DateAdjusters are static methods called with the .with() method

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Date/Time API - Local Time

- A LocalTime represents a time of day

```
LocalTime rightNow = LocalTime.now();

LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)
```

Date/Time API - Zoned Time

- When you have a conference call at 10:00 in New York, but happen to be in Berlin, you expect to be alerted at the correct local time. In this case Time Zones have to be considered.
- Each time zone has an ID, such as America/New_York or Europe/Berlin. To find out all available time zones, call *ZoneId.getAvailableIds*.
- ZonedDateTime builds a specific Instant with Time Zone:

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
ZoneId.of("America/New_York"));
// 1969-07-16T09:32-04:00[America/New_York]
```

- What happens at the begin/end hour of DST? And when you perform operations that cross a DST Zone?

Date/Time API - Formatting and Parsing

- The DateTimeFormatter class provides three kinds of formatters to print a date/time value:
  - Predefined standard formatters
  - Locale-specific formatters (SHORT, MEDIUM, LONG, FULL)
  - Formatters with custom patterns

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11launch);
// 1969-07-16T09:32:00-05:00[America/New_York]
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.
LONG);
String formatted = formatter.format(apollo11launch);
// July 16, 1969 9:32:00 AM EDT
```

Date/Time API - Conversions with Legacy API

- A set of conversion static methods is available
  - java.util.Date: Date.from(instant); date.toInstant()
  - java.util.GregorianCalendar:
    - GregorianCalendar.from(zonedDateTime)
    - cal.toZonedDateTime()
  - LocalDate:
    - Date.valueOf(localDate)
    - date.toLocalDate()
  - etc.

Date/Time API - Exercises

- Write a program that prints how many days you have been alive

- Your return flight leaves Frankfurt at 14:05 and arrives in Los Angeles at 16:40. How long is the flight?

Exceptions - Try-with-resources

- Resources can now implement the Autocloseable interface
- You can add more than one resource in the try() declaration
- Catch and finally are executed after closing the resources

```
open a resource
try {
work with the resource
}
finally {
close the resource
}
```

```
try (Resource res = ...) {
work with res
}
```

Exceptions - Implementing Autocloseable

- It is possible to create new resources using Autocloseable

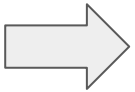
```
public class MyResource implements AutoCloseable {
        public void close()       {
                // take care of closing the resource
        }
}
```

Exercise: Create two autocloseable resources, using them in a try-with-resources, and apply also a catch and finally. Determine what is the order of the calls.

Exceptions - Multiple Catch

- Allow to overcome redundancy in catching exceptions
- You can’t use multiple names for the exceptions
- Order does not matter
- You have to ensure that only one exception is thrown at a time

```
try {
// access the database and write to a file }
catch (SQLException e) { handleErrorCase(e);
} catch (IOException e) { handleErrorCase(e) ;
}
```



```
try {
// access the database and write to a file
}  catch (SQLException | IOException e) {
        handleErrorCase(e);
}
```

Exceptions - Suppressed Exceptions

- In Java 6, an exception thrown in a finally clause discards the previous exception.
- The try-with-resources statement reverses this behavior. When an exception is thrown in a close method of one of the AutoCloseable objects, the original exception gets rethrown, and the exceptions from calling close are caught and attached as “suppressed” exceptions.

```
} catch (Exception e) {
                System.err.println(e.getMessage());
                for (Throwable t : e.getSuppressed())      {
                        System.err.println("suppressed:" + t);
                }
        }
```

- **Exercise**: create a main exception and at least a suppressed exception.

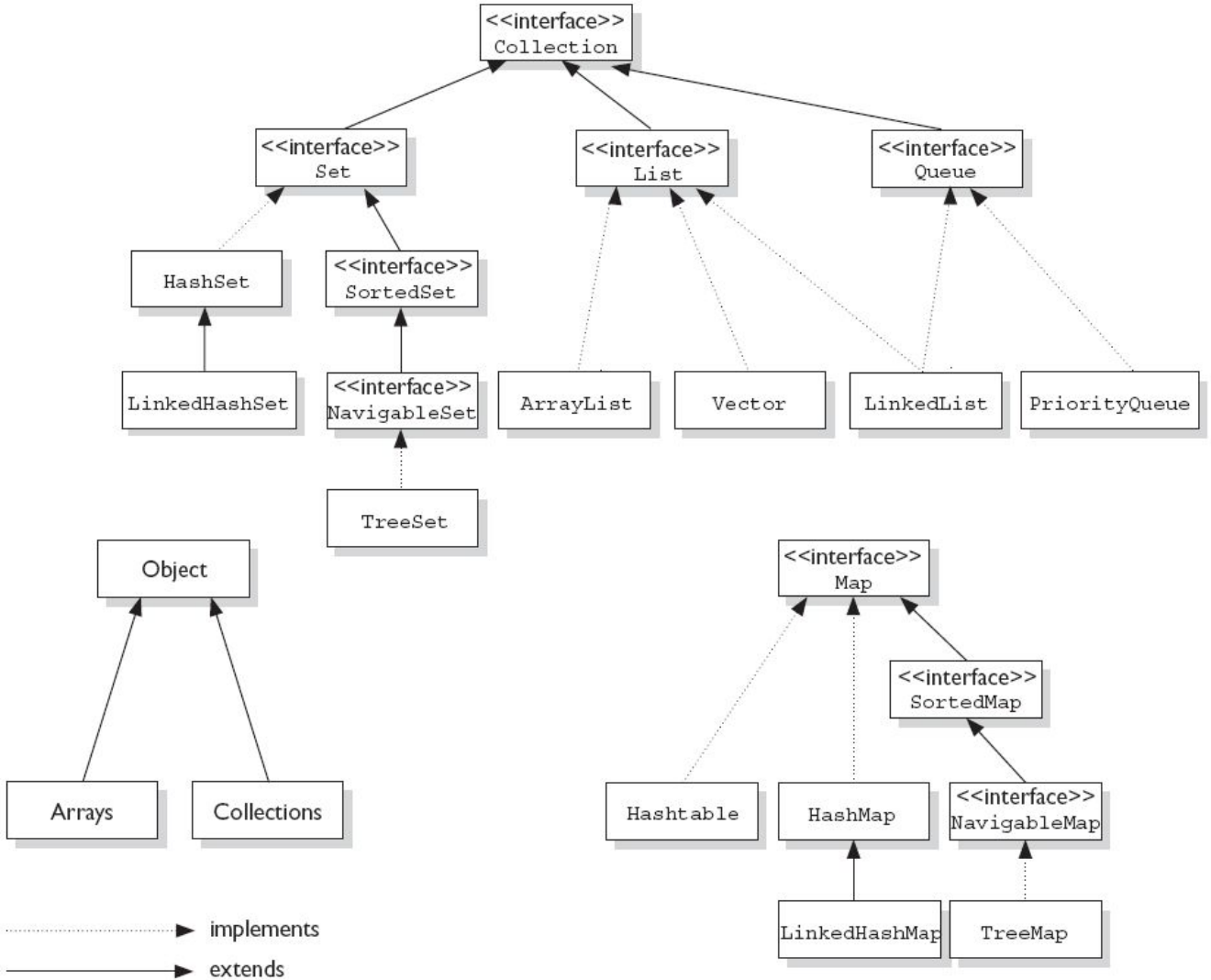
Collections - Overview

Interfaces

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

Classes (Sorted/Unsorted, Ordered/Unordered)

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
LinkedHashSet
<<interface>>
NavigableSet
ArrayList
Vector
LinkedList
PriorityQueue
TreeSet
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
HashMap
<<interface>>
NavigableMap
LinkedHashMap
TreeMap
implements
extends

Collections - List interface

- A **List** has an index
- **ArrayList**: fast iteration and fast random access. Ordered not Sorted. Use for fast iteration with no need to insert/delete.
- **Vector**: Like ArrayList, but synchronized for thread safety. Use only if necessary because of performance hit.
- **LinkedList**: ordered by index with elements doubly linked. Use for fast insertion/deletion. Slow in iteration.

Collections - Set interface

- A **Set** does not allow duplicates
- **HashSet**: unsorted, unordered Set. Uses the object hashCode, needs an efficient hash implementation.
- **LinkedHashSet**: ordered, unsorted Set. Iterates in the order of insertion.
- **TreeSet**: ordered, sorted Set. Elements are in ascending order, or you can implement your own Comparator.

Collections - Map Interface

- Map uses key, value pairs with unique keys.
- HashMap is an unsorted, unordered Map. Allows 1 null key. Allows null values.
- Hashtable: prehistoric. It is a synchronized HashMap. Does not allow null key or values.
- LinkedHashMap: Maintains insertion order.
- TreeMap: Sorted Map. Keys can implement Comparator.

Collections - Queue Interface

- A Queue is a list in FIFO order.
- Contains specific Queue methods.

Using Boxing (in Collections)

- Exercise: What happens here?

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");


Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

Collections - Diamond Syntax

- Collection type can be inferred with Diamond
- This is OK:

```
ArrayList<String> stuff = new ArrayList<>()
List<Dog> myDogs = new ArrayList<> () ;
Map<String, Dog> dogMap = new HashMap<>();
```

This is KO:

```
ArrayList<> stuff = new ArrayList<String>()
```

Collections - Sorting with Comparator

- Comparable: Used by Collections.sort() and java.util.Arrays.sort() to sort Lists and arrays
- Comparator: Used to sort any class in any collection
- Must implement compare():
  - return negative if thisObject < anotherObject
  - Zero if thisObject == anotherObject
  - Positive if thisObject > anotherObject
- A sorted collection can be searched with binarySearch(). A collection must be sorted first to be binary searchable.

EXERCISE: Create a DVD Object with Title, Genre, Price. Put some of them in an ArrayList. Create a Comparator which orders by Genre, Ascending. Use the Comparator to Sort the List. Perform a Binary Search.

Collections and Generics - Polymorphism

- Consider this:
  - List<Integer> myList = new ArrayList<Integer>();
- This is legal as ArrayList implements List

- Consider this:
  - class Parent { }
  - class Child extends Parent { }
  - List<Parent> myList = new ArrayList<Child>();

EXERCISE: Try if this works

Generics - Polymorphism

- Polymorphism in Collections with Generics does not work as in Arrays
- Exercise: Try Rewriting AnimalDoctor with a Typed ArrayList
  - Can you pass an ArrayList<Dog> into an ArrayList<Animal> argument parameter?
  - Can you add a Dog into an ArrayList<Animal>?
- Generics have Erasure
  - The Typing in generics does not exist anymore at runtime
  - At runtime ALL collection code (legacy and typed) looks exactly like the pregeneric version
  - This is done to support Legacy use of Collections (without Generics)

Generics - Polymorphism

- The reason it is dangerous to pass a collection of a subtype into a method that takes a collection of a supertype is because you might add something wrong.

```
public void foo() {
        Cat[] cats = {new Cat(), new Cat()};
        addAnimal(cats); // no problem, send the Cat[] to the method
}
public void addAnimal(Animal [] animals) {
        animals [0] = new Dog(); // Eeek.We just put a Dog  in a Cat array!
}
```

- It compiles for Arrays (doesn't work at Runtime); it does not compile for ArrayList and Generics, because Arrays have ArrayStoreException, while Collections have their type Erased at runtime

Generics - Using ? Wildcard

- The problem is because the Method is adding potentially wrong elements
- We can specify that the method will be read-only by using the ? Wildcard
  - public void readAnimal(List<? extends Animal> animals)
  - This can access elements from the collection, but cannot add.
  - ? extends refers to a superclass or an interface, without distinction
- You can use ? wildcard and ADD to the collection by using ? super
  - public void addAnimal(List<? super Dog> animals) {
    - animals.add(new Dog()); // you can add a Dog with super

EXERCISE: See Example 2

Generics - Declarations

- We can use generic types in our own classes and methods
- This is the same mechanism used to define Collections with Generics

```
public class TestGenerics<T> {      // as the class type
  T anInstance;                     // as an instance variable type
  T [] anArrayOfTs;                 // as an array type

  TestGenerics(T anInstance) {      // as an argument type
     this.anInstance = anInstance;
  }
  T getT() {                        // as a return type
     return anInstance;
  }
}
```

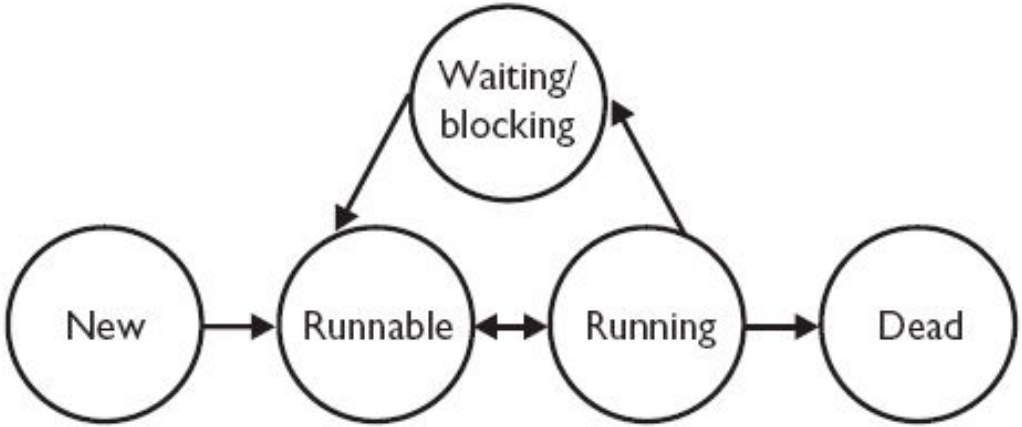
```
public <T> void makeArrayList(T t)

public <T extends Number> void makeArrayList(T t)
```

Threads

- The Thread class is used to start and managing Threads
  - run(); start(); sleep(); yield()
- You can use the Runnable interface to start a Thread (Example 1)
- Order execution is not guaranteed (Example 2) across thread (it is within the same thread)
- Each thread starts and runs to completion (of the run method)
- Once a thread has been started, it can never be started again. (call start() again)

Thread States



New: start() not called
Runnable: start() called and eligible to run
Running: run() is running
Waiting/Blocking/Sleeping: wait() called, or blocked for I/O or lock, or sleep() called
Dead: run() has completed

Threads - sleep(), yield(), join()

- Thread.sleep(millis). Another thread might Interrupt.
  - After sleep, thread gets back to Runnable, not to Running (sleep time might not be exact)
- Threads run with priorities (1-10). yield() puts back the thread to **Runnable (not Waiting)** to allow other threads with the same priority to get into Running.
- Non-static join() allows one thread to wait until the end of another thread.

```
Thread t = new Thread();
t.start();
t.join(); // this thread (main) waits until t completes
```

- EXERCISE: Create some Threads and experiment with sleep() and join()

Threads - Synchronization

- Synchronization is needed when different threads access the same method/object/field in a common class, to avoid race conditions (see Race Example)
- To avoid race conditions, operations which need consistency need to be made atomic (e.g. checking balance and withdrawal are atomic)
- You can't guarantee that a single thread will stay running throughout the entire atomic operation.
- However, you can guarantee that even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data.

Threads - Recipe for Synchronization

- *Mark the shared variables across Threads as private.*
- *Synchronize the code that modifies the variables. (synchronize)*

EXERCISE: Correct Race condition adding synchronize to makeWithdrawal and run again. Compare the outputs.

- Synchronization works with Locks. Every object has a Lock which is used in synchronized blocks.
- When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the *this* instance).

Threads - Synchronization with object lock

- Only methods (or blocks of code) can be synchronized, not variables or classes.
- Each object has just one lock.
- Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.
- If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call.
- If a class has both synchronized and non-synchronized methods, multiple threads can still access the other methods
- If a thread goes to sleep, it holds any locks it has
- A thread can acquire more than one lock (on different objects)

Threads - Exercise

In this exercise we will attempt to synchronize a block of code. Within that block of code we will get the lock on an object, so that other threads cannot modify it while the block of code is executing. We will be creating three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times, and then increment that letter by one. The object we will be using is StringBuffer.

We could synchronize on a String object, but strings cannot be modified once they are created, so we would not be able to increment the letter without generating a new String object. The final output should have 100 As, 100 Bs, and 100 Cs all in unbroken lines.

Threads - Exercise

1. Create a class and extend the Thread class.

2. Override the run() method of Thread. This is where the synchronized block of code will go.

3. For our three thread objects to share the same object, we will need to create a constructor that accepts a StringBuffer object in the argument.

4. The synchronized block of code will obtain a lock on the StringBuffer object from step 3.

5. Within the block, output the StringBuffer 100 times and then increment the letter in the StringBuffer.

6. Finally, in the main() method, create a single StringBuffer object using the letter A, then create three instances of our class and start all three of them.

Threads - Deadlock

- Deadlock occurs when two threads are blocked, with each waiting for the other's lock.
- Example 7: There is a very small chance that it deadlocks
- Deadlocks can be usually resolved by changing lock order.
- See Example 7 and propose an order change to the lock to avoid the deadlock.
-

Threads - Interaction

- Threads can interact with one another to communicate about their locking status. The **Object class** has three methods, wait(), notify(), and notifyAll() that help threads communicate
- wait(), notify(), and notifyAll() must be called from within a synchronized context. A thread can't invoke a wait or notify method on an object unless it owns that object's lock.
- In the same way that every object has a lock, every object can have a list of threads that are waiting for a notification from the object. A thread gets on this waiting list by executing wait() on the target object. From that moment, it doesn't execute any further instructions until the notify() method of the target object is called. If many threads are waiting on the same object, only one will be chosen (in no guaranteed order)

Concurrency

- Creating concurrent programming with barebone Threads can be challenging and error prone: java.util.concurrent provides a tool-kit for robust creation of concurrent applications
- util.concurrent provides a more flexible way to use Threads mechanism:
  - Atomic Variables
  - Locks (Lock, ReadWriteLock, ReentrantLock)
  - Concurrent Collections
  - Executors and ThreadPools
  - Parallel Fork/Join Framework

Concurrency - Atomic Variables

- When threads work on a common field variable, accessing the variable value and incrementing (modifying) it are not atomic operations, they should be manually synchronized (see Example 1)
- Or the variable could be wrapped as Atomic
  - for an int, use AtomicInteger and use its methods (e.g. getAndIncrement()) to modify it
- EXERCISE: replace the shared integer with AtomicInteger in Example 1
- There are Atomic wrappers and operations for Integer, Long, Double
- General wrapper for Objects: AtomicReference updates atomically a referenced object

Concurrency - Locks

- the Locks package allows to create more sophisticated Locks:
  - Obtain a lock in one method and release it in another
  - Multiple wait/notify pool. Threads can select which pool to wait on with Condition
  - Ability to try and acquire a lock and specify an alternate action if unsuccessful

```
Object obj = new Object();
synchronized(obj) {    // traditional locking, blocks until acquired
                       // work
}                      // releases lock automatically
```

No Alternate

```
Lock lock = new ReentrantLock();
lock.lock();          // blocks until acquired
try {
    // do work here
} finally {           // to ensure we unlock
  lock.unlock();      // must manually release
}
```

With try

```
Lock lock = new ReentrantLock();
boolean locked = lock.tryLock(); // try without waiting
if (locked) {
  try {
      // work
  } finally {                    // to ensure we unlock
    lock.unlock();
  }
}
```

Concurrency - Concurrent Collections

- The java.util.concurrent package provides several types of collection that are thread-safe, without using coarse-grained synchronization (see Example 2)
- It is recommended to use these Concurrent Collections when you develop collections that are maintained by multiple threads
- One of the basic mechanisms is CopyOnWrite: every time the collection is altered, a new immutable copy is created (this guarantees thread safety)
    - They can be scanned only via iterator (for-each), as the iterator is bound to the original copy
    - They can be used only when the Collection undergoes only minimal changes, otherwise the performance hit is unacceptable

Concurrency - Concurrent Collections

- ConcurrentHashMap; ConcurrentSkipListSet
  - Provide a safe-thread mechanism which is not CopyOnWrite to solve the performance problem.
  - Iterator is weakly consistent (it may point to the original list or to a later version of the list)
- Blocking Queues
  - A useful system to transfer data between threads in a safe manner
  - Based on producer-consumer scenario, using the concept of a Queue
  - Allows to consume objects from a destination thread with order guarantee (see Example)

Concurrency - Executors and Thread Pools

- Executors (and the ThreadPools used by them) help meet two of the same needs that Threads do:
  - Creating and scheduling some Java code for execution and
  - Optimizing the execution of that code for the hardware resources you have available (using all CPUs, for example). Executor helps to adjust the optimal number of threads we want to run on the current OS to get optimum performance.
  - Introduces the concept of Task, which can be run on one (or more) Threads

```
Runnable r = new MyRunnableTask();
ExecutorService ex = Executors.newCachedThreadPool(); // subtype of Executor
ex.execute(r);
```

FixedThreadPool executor is more common - you emipirically find a fixed optimal number of threads to use to execute the task.

Concurrency - Executors with Callable

- Unlike the Runnable interface, a Callable may return a result upon completing execution and may throw a checked exception.
- An ExecutorService can be passed a Callable instead of a Runnable.
- The primary benefit of using a callable is the ability to return a result.
- Because an ExecutorService may execute the Callable asynchronously (just like a Runnable), Future is used to obtain the status and result of a Callable.
- Future.get() waits for task to get completed from the Executor, greatly simplifying synchronization when consuming the results (Example 3)

Concurrency - Fork/Join Framework

- With Fork/Join, instead of running several parallel small tasks, we parallelize a single big task (e.g. Map/Reduce)
- The Fork-Join ExecutorService implementation is java .util. concurrent . ForkJoinPool.
- You will typically submit a single task to a ForkJoinPool and await its completion.
- The ForkJoinPool and the task itself work together to divide and conquer the problem. Any problem that can be recursively divided can be solved using Fork/Join.
- With the Fork-Join Framework, a java.util.concurrent.ForkJoinTask instance is created to represent the task that should be accomplished.
- RecursiveTask is used if no result has to be returned.

Concurrency - Fork/Join Framework

```
class ForkJoinPaintTask {
  compute() {
    if(isFenceSectionSmall()) {  // is it a manageable amount of work?
      paintFenceSection();       // do the task
    } else {                     // task too big, split it
      ForkJoinPaintTask leftHalf = getLeftHalfOfFence();
      leftHalf.fork();           // queue left half of task
      ForkJoinPaintTask rightHalf = getRightHalfOfFence();
      rightHalf.compute();       // work on right half of task
      leftHalf.join();           // wait for queued task to be complete
    }
  }
}
```