General

Spring Annotations

Annotation Summary

Big O Notation

Notation Summary

New Concurrency Features

New Java 7 Concurrency Features

Concurrency

Intrinsic Locks and Synchronization
Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquistion of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of `c1` from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.
High Level Concurrency Objects
So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new `java.util.concurrent` packages. There are also new concurrent data structures in the Java Collections Framework.

- Lock objects support locking idioms that simplify many concurrent applications.
- Executors define a high-level API for launching and managing threads. Executor implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
- Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
- `ThreadLocalRandom` (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

Design patterns

GoF Creational Patterns	
Abstract Factory	Sets of methods to make various objects.
Builder	Make and return one object various ways.
Factory Method	Methods to make and return components of one object various ways.
Prototype	Make new objects by cloning the objects which you set as prototypes.
Singleton	A class distributes the only instance of itself.
GoF Structural Patterns	
Adapter	A class extends another class, takes in an object, and makes the taken object behave like the extended class.
Bridge	An abstraction and implementation are in different class hierarchies.
Composite	Assemble groups of objects with the same signature.

Decorator	One class takes in another class, both of which extend the same abstract class, and adds functionality.
Facade	One class has a method that performs a complex process calling several other classes.
Flyweight	The reusable and variable parts of a class are broken into two classes to save resources.
Proxy	One class controls the creation of and access to objects in another class.
GoF Behavorial Patterns	
Chain Of Responsibility	A method called in one class can move up a hierarchy to find an object that can properly execute the method.
Command	An object encapsulates everything needed to execute a method in another object.
Interpreter	Define a macro language and syntax, parsing input into objects which perform the correct opertaions.
Iterator	One object can traverse the elements of another object.
Mediator	An object distributes communication between two or more objects.
Memento	One object stores another objects state.
Observer	An object notifies other object(s) if it changes.
State	An object appears to change its` class when the class it passes calls through to switches itself for a related class.
Strategy	An object controls which of a family of methods is called. Each method is in its` own class that extends a common base class.
Template	An abstract class defines various methods, and has one non-overridden method which calls the various methods.
Visitor	One or more related classes have the same method, which calls a method specific for themselves in another class.

Daemon versus User

Daemon threads in Java are like a service providers for other threads or objects running in the same process as the daemon thread. Daemon threads are used for background supporting tasks and are only needed while normal threads are executing. If normal threads are not running and remaining threads are daemon threads then the interpreter exits.

When a new thread is created it inherits the daemon status of its parent. Normal thread and daemon threads differ in what happens when they exit. When the JVM halts any remaining daemon threads are abandoned: finally blocks are not executed, stacks are not unwound – JVM just exits. Due to this reason daemon threads should be used sparingly and it is dangerous to use them for tasks that might perform any sort of I/O.

How to stop a thread

Make sure that the flag is thread safe by using a volatile variable or by using getter and setter methods which are synchronised with the variable being used as the flag.

```
import java.util.Timer;

import java.util.TimerTask;

class CanStop extends Thread {

    private volatile boolean stop = false;

    private int counter = 0;

    public void run() {

        while (!stop && counter < 10000) {

        System.out.println(counter++); }

        if (stop)

           System.out.println("Detected stop");

    }

    public void requestStop() {
```

```
        stop = true;
    }
}
public class Stopping {
    public static void main(String[] args) {
        final CanStop stoppable = new CanStop();
        stoppable.start();
        new Timer(true).schedule(new TimerTask() {
            public void run() {
                System.out.println("Requesting stop");
                stoppable.requestStop();
            }
       }, 350);
    }
}
```

Queues

JDK 5

In this example we will discuss about `java.util.concurrent.BlockingQueue`interface. `java.util.concurrent.BlockingQueue` was added in Java 1.5 along with all the other classes and interfaces of `java.util.concurrent` package. However, what is `BlockingQueue` and what is the difference with the simple `java.util.Queue`? How can we use `BlockingQueues`? Those questions will be answered in the following sections along with a simple example of `BlockingQueue's` usage.

1. What is a BlockingQueue?

`BlockingQueue` is a queue which is thread safe to insert or retrieve elements from it. Also, it provides a mechanism which blocks requests for inserting new elements when the queue is full or requests for removing elements when the queue is empty, with the additional option to stop waiting when a specific timeout passes. This functionality makes `BlockingQueue` a nice way of implementing the Producer-Consumer pattern, as the producing thread can insert elements until the upper limit of `BlockingQueue` while the consuming thread can retrieve elements until the lower limit is reached and of course with the support of the aforementioned blocking functionality.

2. Queues vs BlockingQueues

A `java.util.Queue` is an interface which extends Collection interface and provides methods for inserting, removing or inspecting elements. First-In-First-Out (FIFO) is a very commonly used method for describing a standard queue, while an alternative one would be to order queue elements in LIFO (Last-In-First-Out). However, `BlockingQueues` are more preferable for concurrent development.

3. BlockingQueue methods and implementations

The classes that implement `BlockingQueue` interface are available in `java.util.concurrent` package and they are the following:

- `ArrayBlockingQueue`
- `DelayQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `SynchronousQueue`

For more information for each one of the above classes, you can visit the respective javadoc.

Also, `BlockingQueue` provides methods for inserting, removing and examining elements which are divided in four categories, depending on the way of handling the operations that cannot be satisfied immediately. Meanwhile in cases that the thread tries to insert an element in a full queue

or remove an element from an empty queue. The first category includes the methods that throw an exception, the second category includes the methods returning a special value (e.g. null or false), the third category is related to those methods that block the thread until the operation can be accomplished, and finally, the fourth category includes the methods that block the thread for a given maximum time limit before giving up. These methods are summarized below:

- **Methods related to insertion**
  1. Throws exception: `add(e)`
  2. Special value: `offer(e)`
  3. Blocks: `put(e)`
  4. Times out: `offer(e, time, unit)`
- **Methods related to removal**
  1. Throws exception: `remove()`
  2. Special value: `poll()`
  3. Blocks: `take()`
  4. Times out: `poll(time, unit)`
- **Methods related to examination**
  1. Throws exception: `element()`
  2. Special value: `peek()`
  3. Blocks: not applicable
  4. Times out: not applicable

JDK 6

JDK 6 queues

JDK 7

JDK 7 Queues

StringBuffer and StringBuilder

`StringBuffer` is synchronized, `StringBuilder` is not.

Generics introduces

Templates introduced

Hierarchy and classification[edit]

According to *Java Language Specification*:[3]

- A **type variable** is an unqualified identifier. Type variables are introduced by generic class declarations, generic interface declarations, generic method declarations, and by generic constructor declarations.
- A **class** is generic if it declares one or more type variables. These type variables are known as the type parameters of the class. It defines one or more type variables that act as parameters. A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. All of these parameterized types share the same class at runtime.
- An **interface** is generic if it declares one or more type variables. These type variables are known as the type parameters of the interface. It defines one or more type variables that act as parameters. A generic interface declaration defines a set of types, one for each possible invocation of the type parameter section. All parameterized types share the same interface at runtime.
- A **method** is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the method. The form of the formal type parameter list is identical to a type parameter list of a class or interface.
- A **constructor** can be declared as generic, independently of whether the class that the constructor is declared in is itself generic. A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor. The form of the formal type parameter list is identical to a type parameter list of a generic class or interface.

Motivation[edit]

The following block of Java code illustrates a problem that exists when not using generics. First, it declares an `ArrayList` of type `Object`. Then, it adds a `String` to the `ArrayList`. Finally, it attempts to retrieve the added `String` and cast it to an `Integer`.

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Run time error
```

Although the code is compiled without error, it throws a runtime exception (`java.lang.ClassCastException`) when executing the third line of code. This type of problem can be avoided by using generics and is the primary motivation for using generics.

Using generics, the above code fragment can be rewritten as follows:

```
List<String> v = new ArrayList<>();
v.add("test");
Integer i = v.get(0); // (type error) compilation-time error
```

The type parameter `String` within the angle brackets declares the `ArrayList` to be constituted of `String` (a descendant of the `ArrayList`'s generic `Object` constituents). With generics, it is no longer necessary to cast the third line to any particular type, because the result of `v.get(0)` is defined as `String` by the code generated by the compiler.

Compiling the third line of this fragment with J2SE 5.0 (or later) will yield a compile-time error because the compiler will detect that `v.get(0)` returns `String` instead of `Integer`. For a more elaborate example, see reference.[4]

Here is a small excerpt from the definitions of the interfaces `List` and `Iterator` in package **`java.util`**:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Type wildcards[edit]

Main article: Wildcard (Java)

A type argument for a parameterized type is not limited to a concrete class or interface. Java allows the use of *type wildcards* to serve as type arguments for parameterized types. Wildcards are type arguments in the form "?", possibly with an upper or lower bound. Given that the exact type represented by a wildcard is unknown, restrictions are placed on the type of methods that may be called on object of the parameterized type.

As an example of an unbounded wildcard, `List<?>` indicates a list which has an unknown object type. Methods which take such a list as a parameter will accept any type of list as argument. Reading from the list will return objects of type `Object`, and adding non-null elements to the list is not allowed, since the element type is not known.

To specify the upper bound of a type wildcard, the `extends` keyword is used, which indicates that the type argument is a subtype of the bounding class. So `List<? extends Number>`means that the given list contains objects of some unknown type which extends the `Number` class. For example, the list could be `List<Float>` or `List<Number>`. Reading an element from the list will return a `Number`, while adding non-null elements is once again not allowed.

The use of wildcards above adds flexibility since there is not any inheritance relationship between any two parameterized types with concrete type as type argument. Neither `List<Number>` nor `List<Integer>` is a subtype of the other, even though `Integer` is a subtype of `Number`. So, any method that takes `List<Number>` as a parameter does not accept an argument of `List<Integer>`. If it did, it would be possible to insert a `Number` that is not an `Integer` into it, which violates type safety. Here is sample code that explains the contradiction it brings if `List<Integer>` were a subtype of `List<Number>`:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(2);
List<Number> nums = ints;  // valid if List<Integer> were a subtype of List<Number> according to
substitution rule.
nums.add(3.14);
Integer x=ints.get(1); // now 3.14 is assigned to an Integer variable!
```

The solution with wildcards works because it disallows operations that would violate type safety.

```
List<? extends Number> nums = ints;  // it is OK
nums.add(3.14); // it is prohibited
```

To specify the lower bounding class of a type wildcard, the `super` keyword is used. This keyword indicates that the aforementioned parameterized type is with a type argument which is a super-type of said bounding type. So, `List<? super Number>` could represent `List<Number>` or `List<Object>`. Reading from a list defined as `List<? super Number>`returns elements of type `Object`. Adding to such a list requires elements of type `Number` or any sub type of `Number`.

The mnemonic PECS (Producer Extends, Consumer Super) from the book **Effective Java** by Joshua Bloch gives an easy way to remember when to use wildcards (corresponding to Covariance and Contravariance) in Java.

Generic class definitions[edit]

Here is an example of a generic Java class, which can be used to represent individual entries (key to value mappings) in a map:

```
public class Entry<KeyType, ValueType> {

    private final KeyType key;
    private final ValueType value;

    public Entry(KeyType key, ValueType value) {
        this.key = key;
        this.value = value;
    }

    public KeyType getKey() {
        return key;
    }

    public ValueType getValue() {
        return value;
    }

    public String toString() {
        return "(" + key + ", " + value + ")";
    }

}
```

This generic class could be used in the following ways, for example:

```
Entry<String, String> grade = new Entry<String, String>("Mike", "A");
Entry<String, Integer> mark = new Entry<String, Integer>("Mike", 100);
System.out.println("grade: " + grade);
System.out.println("mark: " + mark);

Entry<Integer, Boolean> prime = new Entry<Integer, Boolean>(13, true);
if (prime.getValue()) System.out.println(prime.getKey() + " is prime.");
else System.out.println(prime.getKey() + " is not prime.");
```

It outputs:

```
grade: (Mike, A)
mark: (Mike, 100)
13 is prime.
```

Java Collections

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.

Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.

8	public boolean contains(object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

How to select type of collection

Java Data Types

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s = 10000, short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.

- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68; char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100; int octal = 0144; int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World" "two\nlines" "\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001'; String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)

\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Decimal Hex Octal

Because hex corresponds much more closely to bits that decimal numbers. Each hex digit corresponds to 4 bits (a nibble). So, once you've learned the bitmask associated with each hex digit (0-F), you can do something like "I want a mask for the low order byte":

`0xff`

or, "I want a mask for the bottom 31 bits":

`0x7fffffff`

Just for reference:

```
HEX BIN
```

- `0 -> 0000`
- `1 -> 0001`
- `2 -> 0010`
- `3 -> 0011`
- `4 -> 0100`
- `5 -> 0101`
- `6 -> 0110`
- `7 -> 0111`
- `8 -> 1000`
- `9 -> 1001`
- `A -> 1010`
- `B -> 1011`
- `C -> 1100`
- `D -> 1101`
- `E -> 1110`
- `F -> 1111`

Converting hex to decimal

Bitwise Operators

The following simple example program demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {

   public static void main(String args[]) {

      int a = 60; /* 60 = 0011 1100 */

      int b = 13; /* 13 = 0000 1101 */

      int c = 0; c = a & b; /* 12 = 0000 1100 */

      System.out.println("a & b = " + c ); c = a | b; /* 61 = 0011 1101 */

      System.out.println("a | b = " + c ); c = a ^ b; /* 49 = 0011 0001 */
```

```
        System.out.println("a ^ b = " + c ); c = ~a; /*-61 = 1100 0011 */
        System.out.println("~a = " + c ); c = a << 2; /* 240 = 1111 0000 */
        System.out.println("a << 2 = " + c ); c = a >> 2; /* 215 = 1111 */
        System.out.println("a >> 2 = " + c ); c = a >>> 2; /* 215 = 0000 1111 */
        System.out.println("a >>> 2 = " + c ); }
}
```

This would produce the following result:

```
a & b = 12
a | b = 61
a ^ b = 49
 ~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

When to Use

You will use bitwise operators if you compare Bitmask enumerations. For example, you have an enumeration of states and an object that can be in more than one of those states. In this case, you'll do a bitwise or to assign more than one state to your object.

for example `state = CONNECTED | IN_PROGRESS` where

`CONNECTED could be 0x00000001` and

`IN_PROGRESS 0x00000010`

- **Bit fields (flags)**
  They're the most efficient way of representing something whose state is defined by several "yes or no" properties. ACLs are a good example; if you have let's say 4 discrete permissions (read, write, execute, change policy), it's better to store this in 1 byte rather than waste 4. These can be mapped to enumeration types in many languages for added convenience.
- **Communication over ports/sockets**
  Always involves checksums, parity, stop bits, flow control algorithms, and so on, which usually depend on the logic values of individual bytes as opposed to numeric values, since the medium may only be capable of transmitting one bit at a time.
- **Compression, Encryption**
  Both of these are heavily dependent on bitwise algorithms. Look at the deflate algorithm for an example - everything is in bits, not bytes.
- **Finite State Machines**
  I'm speaking primarily of the kind embedded in some piece of hardware, although they can be found in software too. These are combinatorial in nature - they might literally be getting "compiled" down to a bunch of logic gates, so they have to be expressed as `AND`, `OR`, `NOT`, etc.
- **Graphics** There's hardly enough space here to get into every area where these operators are used in graphics programming. `XOR` (or ^) is particularly interesting here because applying the same input a second time will undo the first. Older GUIs used to rely on this for selection highlighting and other overlays, in order to eliminate the need for costly redraws. They're still useful in slow graphics protocols (i.e. remote desktop).

More examples

Wait Notify

When to use wait()/notify() in Java?

If you are using a version prior to Java 5, then the wait/notify mechanism can be a key part of thread programming. It is generally used in situations where you need to **communicate between two threads**. This includes cases such as the following:

- For **controlling shared resources** ("pooling") such as database connections. If all resources are currently in use, one thread can *wait* to be *notified* that a resource has become available.

- For *background execution* or **coordinating** *multi-threaded execution*: the controlling thread can *wait* for other threads to *notify* it of completion of a task.
- For creating **thread pools** or **job queues** on a server: a fixed number of threads would sit waiting to be *notified* that a new job had been added to the list.

Wait()/notify() in Java 5

As of Java 5, there is less need for programmers to use wait()/notify(), since other classes are available in the Java concurrency package (java.util.concurrent) to handle these common situations. For example:

- In a common producer-consumer pattern, such as a logging thread, it is generally preferable to use a blocking queue;
- To **coordinate threads**, for example to parallelise a task, it is generally more convenient to use a countdown latch.

Processes and Threads

Most implementations of the Java virtual machine run as a single process and in the Java programming language, concurrent programming is mostly concerned with threads (also called lightweight processes). Multiple processes can only be realized with multiple JVMs.

Thread objects[edit]

Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication. Every application has at least one thread called the main thread. The main thread has the ability to create additional threads as `Runnable` or `Callable` objects. (The `Callable` interface is similar to `Runnable`, in that both are designed for classes whose instances are potentially executed by another thread. A `Runnable`, however, does not return a result and cannot throw a checked exception.)

Each thread can be scheduled on a different CPU core or use time-slicing on a single hardware processor, or time-slicing on many hardware processors. There is no generic solution to how Java threads are mapped to native OS threads. Every JVM implementation can do it in a different way.

Each thread is associated with an instance of the class Thread. Threads can be managed either directly using Thread objects or using abstract mechanisms such as `Executors` and `java.util.concurrent` collections.

Starting a thread[edit]

Two ways to start a thread:

Provide a runnable object[edit]

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from thread!");
    }
    public static void main(String[] args) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Subclass thread[edit]

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from thread!");
    }
    public static void main(String[] args) {
        (new HelloThread()).start();
    }
}
```

Interrupts[edit]

An interrupt is an indication to a thread that it should stop what it is doing and do something else. A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking `Thread.interrupt` sets this flag. By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Joins[edit]

The `Thread.join` methods allow one thread to wait for the completion of another.

Exceptions[edit]

Uncaught exceptions thrown by code will terminate the thread. The `main` thread prints exceptions to the console, but user-created threads need a handler registered to do so.[1][2]

Memory model[edit]

The Java memory model describes how threads in the Java programming language interact through memory. On modern platforms, code is frequently not executed in the order it was written. It is reordered by the compiler, the processor and the memory subsystem to achieve maximum performance. The Java programming language does not guarantee linearizability, or even sequential consistency, when reading or writing fields of shared objects, and this is to allow for compiler optimizations (such as register allocation, common subexpression elimination, and redundant read elimination) all of which work by reordering memory reads—writes.[3]

Synchronization[edit]

Threads communicate primarily by sharing access to fields and the objects that reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.

Reorderings can come into play in incorrectly synchronized multithreaded programs, where one thread is able to observe the effects of other threads, and may be able to detect that variable accesses become visible to other threads in a different order than executed or specified in the program. Most of the time, one thread doesn't care what the other is doing. But when it does, that's what synchronization is for.

To synchronize threads, Java uses monitors, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks; there is a lock associated with each object.

Synchronization has several aspects. The most well-understood is mutual exclusion—only one thread can hold a monitor at once, so synchronizing on a monitor means that once one thread enters a synchronized block protected by a monitor, no other thread can enter a block protected by that monitor until the first thread exits the synchronized block.

But there is more to synchronization than mutual exclusion. Synchronization ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads which synchronize on the same monitor. After we exit a synchronized block, we release the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we acquire the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release.

Reads—writes to fields are linearizable if either the field is volatile, or the field is protected by a unique lock which is acquired by all readers and writers.

Locks and synchronized blocks[edit]

A thread can achieve mutual exclusion either by entering a synchronized block or method, which acquires an implicit lock, or by acquiring an explicit lock (such as the ReentrantLock from the java.util.concurrent.locks package). Both approaches have the same implications for memory behavior. If all accesses to a particular field are protected by the same lock, then reads—writes to that field are linearizable (atomic).

Volatile fields[edit]

When applied to a field, the Java `volatile` guarantees that:

1. *(In all versions of Java)* There is a global ordering on the reads and writes to a volatile variable. This implies that every thread accessing a volatile field will read its current value before continuing, instead of (potentially) using a cached value. (However, there is no guarantee about the relative ordering of volatile reads and writes with regular reads and writes, meaning that it's generally not a useful threading construct.)
2. *(In Java 5 or later)* Volatile reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.[4] This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.

Volatile fields are linearizable. Reading a volatile field is like acquiring a lock: the working memory is invalidated and the volatile field's current value is reread from memory. Writing a volatile field is like releasing a lock: the volatile field is immediately written back to memory.

Final fields[edit]

A field declared to be final cannot be modified once it has been initialized. An object's final fields are initialized in its constructor. If the constructor follows certain simple rules, then the correct value of any final fields will be visible to other threads without synchronization. The rule is simple: the `this` reference must not be released from the constructor before the constructor returns.

History[edit]

Since JDK 1.2, Java has included a standard set of collection classes, the Java collections framework

Doug Lea, who also participated in the Java collections framework implementation, developed a concurrency package, comprising several concurrency primitives and a large battery of collection-related classes.[5] This work was continued and updated as part of JSR 166 which was chaired by Doug Lea.

JDK 5.0 incorporated many additions and clarifications to the Java concurrency model. The concurrency APIs developed by JSR 166 were also included as part of the JDK for the first time. JSR 133 provided support for well-defined atomic operations in a multithreaded/multiprocessor environment.

Both the Java SE 6 and Java SE 7 releases introduced updated versions of the JSR 166 APIs as well as several new additional APIs.

TLS and SSL

SSL versus TLS

TLS (Transport Layer Security) and SSL (Secure Sockets Layer) are protocols that provide data encryption and authentication between applications and servers in scenarios where that data is being sent across an insecure network, such as checking your email (How does the Secure Socket Layer work?). The terms SSL and TLS are often used interchangeably or in conjunction with each other (TLS/SSL), but one is in fact the predecessor of the other — SSL 3.0 served as the basis for TLS 1.0 which, as a result, is sometimes referred to as SSL 3.1. With this said though, *is there actually a practical difference between the two?*

Which is more secure – SSL or TLS?

It used to be believed that TLS v1.0 was marginally more secure than SSL v3.0, its predecessor. However, SSL v3.0 is getting very old and recent developments, such as the POODLE vulnerability have shown that SSL v3.0 is now completely insecure (especially for web sites using it). Even before the POODLE was set loose, the US Government had already mandated that SSL v3 not be used for sensitive government communications or for HIPAA-compliant communications. If that was not enough … POODLE certainly was. In fact, as a result of POODLE, SSL v3 is being disabled on web sites all over the world and for many other services as well.

SSL v3.0 is effectively “dead” as a useful security protocol. Places that still allow its use for web hosting as placing their “secure web sites” at risk; Organizations that allow SSL v3 use to persist for other protocols (e.g. IMAP) should take steps to remove that support at the soonest software update maintenance window.

Subsequent versions of TLS — v1.1 and v1.2 are *significantly more secure* and fix many vulnerabilities present in SSL v3.0 and TLS v1.0. For example, the BEAST attack that can completely break web sites running on older SSL v3.0 and TLS v1.0 protocols. The newer TLS versions, if properly configured, prevent the BEAST and other attack vectors and provide many stronger ciphers and encryption methods.

Unfortunately, even now a majority of web sites do not use the newer versions of TLS and permit weak encryption ciphers. Check how well your favorite web site is configured.