Prepare Java Interview

Notes on prepare java interview

Test Live Writer

June 1st, 2010

This is a test for Windows Live Writer.

Here is a code section:

```
if (liveWrite == “good”) {
   useIt();
} else {
   discardIt();
}
// end of code
```

More stuff here…

bla bla.

Posted in Uncategorized | 10 Comments »

Prepare Java Interview 9: Java Design Patterns

March 30th, 2010

Design Pattern is a general reusable solution to a commonly occurring problem in software design.

Creational Patterns

Singleton

One instance of a class or one value accessible globally in an application.

```
public class A  { // do not implements Serializable, Cloneable

    private static final A instance = new A();

        // the only one instance

    private A(){}

        // private constructor to prevent instantiate outside

        //   either calling constructor: new A()
```

```
        //   or calling reflection: newInstance();
    public static A getInstance() { return instance;}
        // expose to the world
}

// enum singleton - the preferred approach
public enum A {
    INSTANCE;
    // other stuff
}

// lazy initialized singleton with "Demand Holder Idiom"
// no synchronization needed
public class A {
    // define a Demand Holder class
    private static class HolderA {
        private static A inst = new A();
    }
    // provide public api
    public static A getInstance() {
        return A.HolderA.inst;
    }
    // private constructor and other stuff
}
```

About final modifier for singleton class:

* Some said “May be extensible by the subclass (if not define singleton class as file)” – Not ture – it’ll not be allowed to subclass if the only constructor is private.

* Some said “ensure unique instance by defining class final to prevent cloning” – Not true – you cannot call clone if the class does not implement Cloneable (and not extends any Cloneable class) because inherited (Object) clone() is a pretected method.

Static Factory methods

(Not the same as Factory Method of GoF)

Static Factory methods are static methods that return an instance of the class.

- have names, unlike constructors, which can clarify code.
- not required to create a new object each time, unlike constructor, objects can be cached and reused, if necessary.
- can return a subtype of their return type – in particular, *can return an object whose implementation class is unknown to the caller.*

Disadvantage: cannot be subclass (private constructor); not distinguishable from other static methods.

Example

```
public static Boolean valueOf(boolean b) {

    return (b ? TRUE : FALSE);

}

public static Calendar getInstance() {

    Calendar cal = createCalendar(TimeZone.getDefaultRef(),

                                  Locale.getDefault());

    cal.sharedZone = true;

    return cal;

}
```

Builder

Consider a builder when faced with many constructor parameters.

Telescoping constructor pattern: provide with only the required parameters, another with one optional parameter, a third with two optional parameters, and so on.

```
class Foo {

      public Foo(int a){this.a=a;this.x=0;this.y=0;}

      public Foo(int a, int x){this.a=a;this.x=x;this.y=0;}

      public Foo(int a, int x, int y){this.a=a;this.x=x;this.y=y;}

}
```

The builder is a static member class of the class it builds.

```
class Foo {
      private final int a, x, y; // a reqired; x,y optional
      public static class FooBuilder {
            private int a, x, y;
            public FooBuilder(int a){this.a=a; }
            public FooBuilder x(int x) {this.x=x; return this;}
            public FooBuilder y(int y) {this.y=y; return this;}
            public Foo build() { return new Foo(this);}
      }
      private Foo(FooBuilder bd){
            this.a=bd.a; x=bd.x; this.y=bd.y;
      }
}

// caller is flexible in feeding the optional parameters
Foo f = new Foo.FooBuilder(1).x(2).y(3).build();
Foo f2 = new Foo.FooBuilder(1).x(2).build();
```

Good in creating classes with many parameters.

Prototype

Clone complicated object.

The type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

Structural Patterns

Adapter (Wrapper)

Convert the existing interfaces to a new interface.

- Try to match an interface.
- Compatibility.
- Reusable.
- Delegate objects.
- Achieve the goal by inheritance or by composition

Bridge

Decouple an abstraction (interface) from its implementation so that the two can vary independently.

The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

- Separate abstraction and implementation
- Hide implementation details from clients

Composite

Composite allows a group of objects to be treated in the same way as a single instance of an object.

The intent of composite is to “compose” objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.

- Want to represent a part-whole relationship like tree folder system
- Group components to form larger components, which in turn can be grouped to form still larger components.

Composite can be used when clients should ignore the difference between compositions of objects and individual objects.

java.io.File is a composite pattern, it handles both directory and file at the same time. It is an abstract representation of file and directory pathnames. getName, getPath, getParent, etc are the same.

XML node is another example.

Decorator

One solution is make a mutable object become immutable is to use the Decorator pattern as a wrapper around each of the list’s items to make them immutable.

```
class Cart {
  private final List items;

  // this is NOT immutable!
  public List Cart(List items) { this.items=items; }
  public List getItems() { return items; }

  // This is immutable constructor and getter
```

```
  public Cart(List items) {this.items=Arrays.asList(items.toArray());}

  public List getItems() { return Collections.unmodifiableList(items);}

}
```

(Architectural) Façade

Make a complex system simpler by providing a unified or general interface, which is a higher layer to these subsystems. Or a simplified interface to a larger body of code, such as a class library.

- Want to reduce complexities of a system.
- Decouple subsystems, reduce its dependency, and improve portability.
- Minimize the communication and dependency between subsystems.
- Shield clients from subsystem components.

Example:

JDBC design is a good example of Façade pattern. A database design is complicated. JDBC is used to connect the database and manipulate data without exposing details to the clients.

java.util.Collections is another example.

Proxy

A proxy is a class functioning as an interface to something else.

It can use a simple object to represent a complex one or provide a placeholder for another object to control access to it.

Example: RMI

Behavioral Patterns

Chain of Responsibility

consisting of a source of command objects and a series of **processing objects**. Each processing object contains a set of logic that describes the types of command objects that it can handle, and how to pass off those that it cannot to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

Command

Streamlize objects by providing an interface to encapsulate a request and make the interface implemented by subclasses in order to parameterize the clients.

```
/*the Command interface*/

public interface Command {

    void execute();

}
```

Iterator

Provide a way to move through a list of collection or aggregated objects without knowing its internal representations.

- Use a standard interface to represent data objects.
- Similar to Enumeration class, but more effective.
- Need to filter out some info from an aggregated collection.

Observer (subset of the publish/subscribe patter)

An object (called the subject) maintains a list of its dependents (called observers) and notifies them automatically of any state changes.

It is mainly used to implement distributed event handling systems.

Example:

java.util.Observer and java.util.Observable.

State

This pattern is used in to represent the state of an object. This is a clean way for an object to partially change its type at runtime.

Tags: Design Pattern, Java
Posted in Java | 247 Comments »

Prepare Java Interview 8: Java Concurrency Programming

March 29th, 2010

It's used to be called multi-threads programming.

Java Thread Concepts

Thread and Process

The unit of execution. Process has its own memory space. Thread lives inside process, it shares the resources with other threads in the same process. Java application usually runs as a single process with multiple threads.

Synchronization

Synchronization mechanism is used to prevent two problems created in multithread programming: *thread interference* and *memory consistency errors*.

Reentrant Synchronization

A thread *can* acquire a lock that it already owns. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock.

Atomicity

Shared data are accessed by atomic operations – the operation cannot be interrupted by other threads. This is usually achieved by mutual exclusion.

A *race condition* occurs when getting the right answer relies on lucky timing.

Locking is used for threads to exclusively access a shared mutable object. Every java object has an intrinsic lock (or monitor). The intrinsic lock is auto acquired by executing thread before entering a synchronized black, and is released when exit the block.

A *deadlock* is a situation where threads are blocked indefinitely, waiting for each other to release the needed lock.

Visibility

Visibility determines when the effects of one thread can be seen by another. Visibility is ensured by some form of synchronization.

Volatile variables are a weaker form of synchronization. Volatile can only guarantee visibility. (Locking can guarantee both visibility and atomicity.)

Publishing – make object available to code outside of its current scope. An object that is published when it should not have been is said to escaped.

Thread confinement

When an object is confined to a single tread, is automatically thread-safe.

Stack confinement – local variables are intrinsically confined to the executing thread

Thread-local -variables are localized so that each thread has its own private copy.

ThradLocal class provides thread-local variables.

Immutability

Immutable objects are always thread-safe. An object is immutable if:

– its state cannot be modified after construction

– all its fields are finials

– it is properly constructed (that this reference does not escape during construction)

Examples are all Java primitive type wrapper classes include String are immutable.

Ordering

Ordering determines when actions in one thread can be seen to occur out of order with respect to another.

Thread-Safe

A class (or a piece of code) is thread-safe if it behaves correctly when accessed from multiple threads simultaneously with no additional synchronization or coordination on the part of the calling code.

Multithread programming is hard because the code first has to function correctly in single thread, then it also has to function correctly in multiple thread.

How to make your code thread-safe?

Stateless class, immutable Object, or thread confinement are some of the simple ways to make the code thread safe.

When dealing with shared mutable objects, use synchronized statements/methods on the critical section, use concurrency utilities, use Thread-safe wrappers.

Java Thread Management

Each Java thread is associated with an instance of the Thread class, created by either subclass Thread or implementation class of Runnable. In either case, you override run() method – a typical example of create and start a thread:

Create and Start

```
new Thread(new Runnable() {

    public void run() {  // do something... }

}).start();
```

Sleep and Join

Thread sleep is causes the current thread to suspend and yield to other thread: `Thread.sleep(1000);` The join method allows one thread to wait for the completion of another: `t.join(); // wait for Thread t`

Interrupt

Thread has interrupt() and interrupted() methods. But depending on thread interruption is not reliable, since there are many reason a thread can be interrupted, some times unintentionally.

Notice stop(), susupend() are deprecated, yieald() is rarely used. A thread is stopped once the run() ended, and it yield others once it's in sleep.

Guarded Blocks

There are different ways for threads to coordinate each other. One way is to periodically wake up the thread and test the condition (easy but not efficient).

```
    while (!conditionFlag) {  // a volatile variable

        try { Thread.sleep(1000);}

        catch(InterruptedException e){}

    }

    // do something
```

Another way is to use object wait and notify mechanism.

Object wait and notify

wait suspend the current thread until notify is called (by another thread). Note the thread has to hold the (object intrinsic) lock before it can release it (suspend the thread); and it'll re-gain the lock when someone else called notify (on the same object).

```
Thread A:

public void run() {
```

```
    synchronized (lockObject) {
        while (!conditionFlag) {
            try { lockObject.wait()}
            catch(InterruptedException e){}
        }
    }
}
In Thread B, call: lockObject.notifyAll() will trigger the event and wake up Thread A.
```

Wait and notify is an old mechanism but still a popular interview subject – avoid to use it in practice.

(Effective Java: Item 69. Prefer concurrency utilities to wait and notify)

Synchronization

```
//synchronized statement (block)
synchronized(lockObj){// lock can be set to any object include 'this'
    // do something
}


// synchronized method -
// the lock is 'this', the whole method body is synchronized block
public synchronized void foo() { ...}


public static synchronized void foo(){...}
```

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.

Java Atomic Access:

- Java reads and writes are atomic for all variables declared ***volatile***.
- Java reads and writes are atomic for reference variables and for primitive variables except ***long*** and ***double***.

Java 5 Concurrency Utilities

Executor Framework

```
// 1. Basic interface
public interface Executor {
            void execute(Runnable command);
}

// have your own implementation
class A implements Executor {
      @Override
      public void execute(Runnable command) {
            new Thread(command).start();
      }
}



// 2. use ExecutorService (sub-interface of Executor):
// it accepts Runnable and Callable objects.
ExecutorService exec = Executors.newFixedThreadPool(10);
exec.execute(new Runnable(){
    public void run(){
        try {
            for (int i=0;i<10;i++){
                // do something useful
            }
        } catch (InterruptedException e) {}
    }
});
// 3. Thread Pool
```

// Use factory class Executors:

Executors.newFixedThreadPool()

Executors.CachedThreadPool();

Executors.newSingleThreadExecutors();

Executors.newScheduedThreadPool();

Concurrent Collections

BlockingQueue/ArrayBlockingQueue/LinkedBlockingQueue/LinedBlockingDeque

put(e); take(); // blocking

offer(e,time,unit); poll(time,unit); // time out

ConcurrentMap/ConcurrentHashMap

The action is performed atomically – putIfAbsent(K key, V value)

Atomic Variables

```
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Lock

java.util.concurrent.locks.Lock

```
class Account {
      public Lock lock = new ReentrantLock();
      ...
```

```
}

fromAcct.lock.lock(); // wait until it gets the lock

toAcct.lock.lock();

try{

      fromAcct.debit(amount);

      toAcct.credit(amount);

      return true;

}

finally {

      fromAcct.lock.unlock();

      toAcct.lock.unlock();

}
```

Two More Items about Thread

- 8 Ways of Thread Communication
- Java Thread States

References:

The Java Tutorials – Concurrency

Java Concurrency in Practice

Tags: Concurrency, Java, Multithread
Posted in Java | 236 Comments »

Prepare Java Interview 7: Java Memory Management

March 27th, 2010

Related Concepts

- All java objects are created in the heap of the memory. So Java memory management is all about manage objects live in the heap.
- The goal of garbage collector is to identify and reclaim the unreachable objects – the objects have no strong references (regular java object reference).

- Weak references are the references created by using java.lang.ref.reference.WeakReference, which do not prevent their referents from being garbage collected.
- There are different garbage collection algorithms. Mark and Sweep algorithm is to mark the live objects in phase 1, and reclaim the unmarked objects in phase 2 (visit objects tree/graph twice); while Copying algorithm is to copy the live objects between the old and new memory spaces (faster but uses twice memory).
- Most java objects have a very short live time (die young), so generation collection algorithm split the heap into different generation spaces and handles them separately.
- In general, when people asking about garbage collector algorithm, they mean the generational collection algorithm which is used in Sun HotSpot JVM.
- Java VM runs differently in Server Mode and Client Mode, optimized for long running apps and quick startup and smaller memory footprint respectively. This involves compiler (JIT) behaving differently, but the garbage collector plays an important role as well.
- Further, JVM default garbage collection setting is sufficient in most cases especially in client mode; but it can be substantially different in multiprocessor server machine where tuning is needed.

How garbage collector works

A brief explanation:

Java object heap are divided into separate generation spaces. Objects are allocated in young generation space first, the survivor objects are moved to tenured generation space. When tenured space fills up, the major collection happens and clean up all spaces (collected all garbage).

The permanent space is where the JVM specific data are stored.

young	tenured	permanent

Details description as following:

1. Young space consists of Eden space, two survivor spaces, and a virtual space. Objects are initially allocated in Eden space.

||||||||| Eden	survivor	survivor	virtual	tenured	permanent

————————————- young generation ————————————

2. When Eden space is full, the surviving objects move up to one of survivor space while the rest are dead (so Eden space is ready to take new objects).

Eden	|||||	S2	virtual	tenured	permanent

————————————- young generation ————————————

3. Objects in survivor space 1 are copied to survivor space 2 in a minor collection and leave space 1 empty.

Eden	S1	|||||	virtual	tenured	permanent

———————————— young generation ————————————

4. Objects are copied back and forth between the two survivor spaces (copying collector algorithm), so most short lived objects gets left off (dead) during the copying (called a minor collection). The survivors move up to the Tenured space eventually when one of the survivor space fills up. Notice that at any time, there is always one survivor space is empty which allows other survivor space objects and Eden objects to be copied to there.

Eden	S1	S2	virtual	||||||	permanent

——————— young generation ———————

5. When tenured space fills up, a major collection occurs (use mark and sweep algorithm) where all unreachable objects from tenured space as well as Eden and survivor spaces are reclaimed.

The above steps repeats as long as program running.

(The algorithm is used in parallel collector in HotSpot JVM.)

Java memory leaks

Causes of Memory Leaks

The following are the causes of java memory leaks (some of them may related):

- Program is unintentionally holding references to objects that are no longer needed, especially in long lived applications.
- Program fails to release the references to objects after use.
- Caused by the bugs in the program or the bugs in the libraries that program called.
- Program make excessive use of finalizers, so the daemon finalizer thread cannot keep up with the increase rate of the finalization queue. (Do not override finalize())
- Program fail to clean up or free native system resources (for example in native methods calling C/C++ code which generate memory leak).
- Program fail to close the resources after use.

    An example is *JDBC* – when using connection pools, and after calling `close()` on the connection, `Statement` and `ResultSet` objects are not closed where it can hold large set of data.

Notice that OutOfMemoryError does not necessarily imply a memory leak. It could simply caused by improper configuration (heap size too small for instance).

Detect Memory Leaks

- Use the operating system process/memory monitor
- Use the totalMemory() and freeMemory() methods in the program
- Use JVM Heap Dump
- Use memory-profiling tools (may only feasible in development environment)

JVM Performance and Tuning

JVM Ergonomics is to provide good performance by selecting the right garbage collection, heap size, and runtime compiler. There are two primary measures:

Throughput goal:

the ratio of GC time vs. application (non-GC) time. If we consider the garbage collection time is “waste time” – since it’s not be used by your program, then this goal is looking for setting a percentage of the useful time vs. the total time (useful + wasted time). It can be set by: -XX:GCTimeRatio=<nnn>

Max pause time goal

the longest pause time to suspend applications (while garbage collector is working). It can be set by:

-XX:MaxGCPauseMills=<nnn>

You can prioritizing the goal based on your application type, either set the max pause time or maximize the throughput. The heap size tends to auto adjusted while JVM tries to meet the goal at runtime.

Basic Settings:

Besides the two goals mentioned above, there are other parameters people know more about:

-XX:MinHeapFreeRatio=40

-XX:MaxHeapFreeRatio=70

-Xms=3670k

-Xmx=64m

In the above setting sample: if the percent of free space falls below 40%, the heap size will be expanded to maintain 40% free space; if the free space exceeds 70%, the heap will be contracted so that only 70% of space is free. In both cases, it is of course limited by minimum and maximum size.

Finally, although it'd be nice if you familiar with some fancy java profilers, JDK does provide some handy tools to gather memory usage and GC collection information, such as command line option:

-verbose:gc

and option:

-XX:+PrintGCTimeStamps.

References:

http://java.sun.com/docs/hotspot/gc5.0/ergo5.html

http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html

Tags: Garbage Collector, Java, Java Memory Management
Posted in Java | 252 Comments »

Prepare Java Interview 6: Java Basic I/O

March 26th, 2010

The bottom line is to know that there are two different types of data: byte streams and character streams, and Java uses different APIs to handle different streams.

Further, serialization is a process of converting Java object into a sequence of bits to store in file system or transmit through network.

Byte Streams

Abstract class *InputStream* and *OutputStream* and their subclasses are used to perform input and output of 8-bit bytes.

```
in = new FileInputStream("xanadu.txt");
```

```
out = new FileOutputStream("outagain.txt");
int c;
while ((c = in.read()) != -1) {
    out.write(c);
}
```

Notice `read()` returns an `int` value, although it reads a byte – it holds a byte value in its last 8 bits., thus allows -1 to indicate the end.

Character Streams

Abstract class *Reader* and *Writer* and the subclasses are used for Character Streams operations, it automatically translates fixed width 16-bit Unicode stream to local character set.

The read() usage is the same as byte stream, now the `int` variable holds a character value in its last 16 bits. Remember int is 4-byte long.

Buffered Streams

Since it is not efficient to read byte or character one at a time, there are buffered version of read and write:

```
in = new BufferedReader(new FileReader("xanadu.txt"));
String l;
while ((l = out.readLine()) != null) {
    out.println(l);
}
```

Data Streams

It just makes your life easy by providing APIs to read/write the primitive types data (include String) into byte streams. Defined by two interfaces: DataInput and DataOutput, APIs like:

readInt(), readDouble() and writeInit(), writeDouble(), …

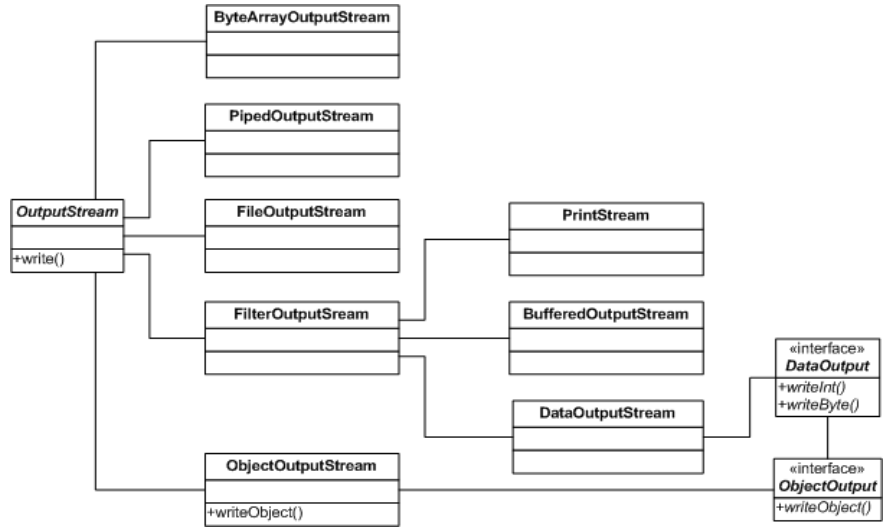
Object Streams

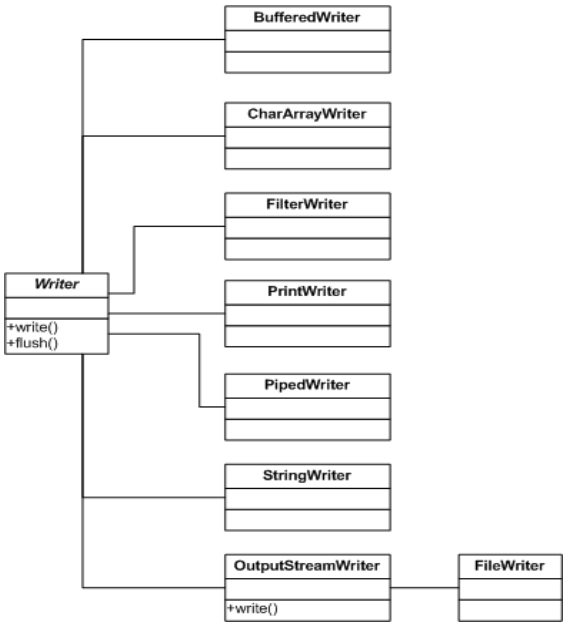
Object Streams are part of Data Streams, dealing with Object input and output. Defined by two interfaces: ObjectInput and ObjectOutput (sub-interface of DataInput and DataOutput). Methods include:

Object readObject(); void writeObject(Object obj);

I/O Classes

The diagram only contains output interfaces, abstract classes (italic), and implementation classes. The input classes are similar.





Notice that `FileOutputStream` is meant for writing streams of raw bytes such as image data. For writing characters stream, use `FileWriter`.

There are two general-purpose byte-to-character “bridge” streams: InputStreamReader and OutputStreamWriter, converting characters and bytes.

Serialization

Serializable

Serialization can be done as simple as adding implements Serializable. (Serializable is a marker interface which you don't really need to implement anything). Example:

```
class A implements Serializable {
      private int x;
      private String s;
      // other stuff
}
A a = new A();
a.x=3; a.s="hi"; // ya, can do it in the same class
// serializing - write to a file
ObjectOutputStream oos =
    new ObjectOutputStream(new FileOutputStream("test.ser"));
oos.writeObject(a);
oos.close();
// deserializing - get it out
ObjectInputStream ins =
    new ObjectInputStream(new FileInputStream("test.ser"));
A a2 = (A) ins.readObject();
System.out.println(a2.s + ", "+a2.x);
```

ins.close();

Notice that trancient and static files are not serialized.

All non-primitive type fields class need to implements serializable as well, or declared as transient if they are not part of the serialization. Otherwise NotSerializableException will be thrown.

Declare an explicit serial version UID to ensure the compatibility (and slightly perform gain):

```
private static final long serialVersionUID = 1L; // any number will do
```

Using a customize serialized form

You can customize your class serialization process. The default serialized form is provided for convenience but has many disadvantages (expose internal structures, consume excessive space and time, can cause stack overflows).

Customized serialized is done by providing providing writeObject and readObject in the serialized class:

```
private void writeObject(ObjectOutputStream s) throws IOException{
```

```
        s.defaultWriteObject();   // always call default first
        s.writeObject(someState); // add your thing
        s.writeInt(size);         // more your thing
    }
    private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
        s.defaultReadObject();   // always read default first
        someState d = (SomeState)s.readObject(); // get your stuff
        size = s.readInt();                      // yours
    }
```

Notice the order for the variables in write and read have to be exactly the same.

Notice writeObject/readObject is private methods, but you may want to document it (with @serialData) since it is a public API in serialized form. (Same is true for private fields.)

Concerns about Serializable

It descreases the flexibility to change the class implementation, since the class's serialized form (byte-stream) becomes part of exported API. (Any changes would break the older version.)

Increases the chance of bug – the deserialization is a hidden constructor, object construct this way my not be properly initialized.

In general, value classes like Date should implement Serializable. Interface should rarely extend Serializable. Notice Throwable implements Serializable, so all exception classes have serialized form.

readResolve and writeReplace are used for instance control – a new instance is created in every deserializing which can be problem in multiple jvm, or a singleton will no longer be singleton once it implements Serializable.

Externalizable

If you want completely control over the serialization, use Externalizable instead of Serializable. Exeternalizable is a sub-interface of Serializable, has two abstract methods: writeExternal and readExternal, which supercede customized implementations of writeObject and readObject methods. Example:

```
class A  implements Externalizable {
        private static final long serialVersionUID = 1L;
        private int x;
        private String s;
        public void readExternal(ObjectInput in) throws IOException,
                    ClassNotFoundException {
              x = in.readInt();
              s = (String)in.readObject();
```

```
        }
        public void writeExternal(ObjectOutput out) throws IOException {
                out.writeInt(x);
                out.writeObject(s);
        }
}


A a = new A();
...
a.writeExternal(oos); // serializing a
...
A a2 = new A();
a2.readExternal(ins);  // deserializing
```

Refereces:

The Java Tutorials – Basic I/O

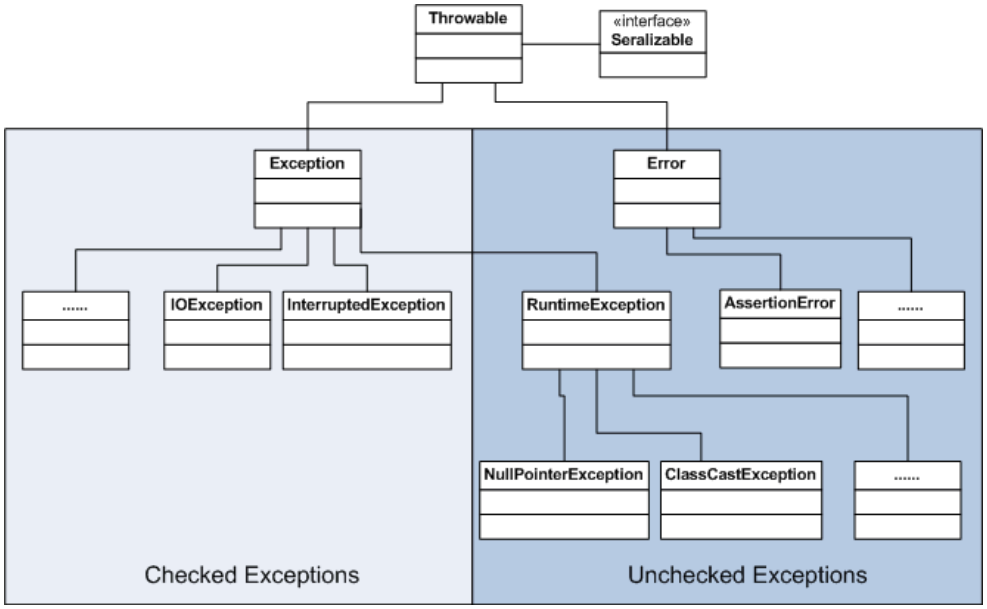
Effective Java 2nd Edition

Tags: I/O, Java
Posted in Java | 263 Comments »

Prepare Java Interview 5: Java Exceptions And Assertions

March 25th, 2010

Exceptions



1. Throwable class is the super-class of all errors and exceptions. It is not an interface.

Only Throwable or its subclasses can be thrown by the throw statement or can be the argument type in a catch clause.

2. Checked exceptions *are subject* to the Catch. All exceptions are checked exceptions, except for those indicated by `Error`, `RuntimeException`, and their subclasses. These are exceptional conditions that a well-written application should anticipate and recover from. Example: `FileNotFoundException, InterruptedException`.

3. Unchecked Exceptions includes Errors and Runtime Exceptions.

4. Runtime exceptions are `RuntimeException` and its subclasses. These are exceptional conditions internal to the application and usually cannot anticipate or recover from. Example: `NullPointerException, ArrayIndexOutOfBoundsException , IllegalArgumentException`

5. Errors indicates serious problems that the application should not try to catch because they cannot be anticipated or recovered from. Examples: `AssertionError, ThreadDeath, VirtualMachineError.`

6. Checked or Unchecked Exception?

- Use checked exceptions for conditions from which the caller can reasonably be expected to recover.
- Use runtime exceptions to indicate programming errors. The majority of runtime exceptions indicate precondition violations – simply by the client not to adhere to the API specification. Example: `ArrayIndexOutOfBoundsException`.
- Avoid unnecessary use of checked exceptions

```
catch(TheCheckedException e) {
throw new AssertionError(); //  or System.exit(1);
}
```

- Favor the use of standard exceptions. Examples: IllegalArgumentException, UnsupportedOperationException
- Exception Translation: Higher layers should catch lower-level exceptions, and throw exceptions that can be explained in higher-level abstraction.
- Document all exceptions thrown by each method

7. Finally

One question people like to ask is that can you have try block without catch? The answer is yes, and you use finally:

```
try {
   //doSomething();
} finally {
    //releaseResource();
}
```

In this case, you want the caller to handle the exception but want to make sure the release resource code is executed no matter what happens.

Assertions

Assertion is basically for testing the assumptions in the code. Widely used in unit test. Assertion can be turned on with command line argument "-ea". Assertion code are normally disabled (but not removed) in production environment.

- Usage:

  assert x == 0;
  assert x == 0 : "x (should be 0) is:" + x;
  // provide detail message for AssertionError

  It usually used to assert an invariant (Conditions, Control-Flow, Class invariant, etc). Example:

  ```
  switch(x){
    case 0: ...; break;
    case 1: ...; break;
    case 2: ...; break;
   default: assert false; // we don't expect x has other values
  }
  ```

- Do not use assertions for argument checking in public methods.

  ```
  public boolean parse (String s){
    assert s != null;
           // bad - it can cause trouble when assertion turns off
   …
  }
  ```

  public void parse (String s){
  if (s == null)
  throw new IllegalArgumentException("s is null");
  // good – now we know it is caller's fault
  …
  }

  However it is ok to use assertion in private methods – that's something internal to your class.

- Do *not* use assertions to do any real work that the rest of your program rely on:

  ```
  // broken - action is contained in assertion
  assert list.remove(obj);
  ```

  Remember that all assertion code will be disabled in production.

Assertion vs. Exception

- Assertion does not have stack traces, try/catch/finally, catch clause while exception does;
- Assertion handles the error right in the place, while exception can propagate errors to upper level classes;
- Assertion code is disabled in production while exception doesn't;
- Assertion always checks the condition at runtime, while exception can catch the errors at compile time (checked exception);
- Assertion handles error in a fail-fast fashion – simply exit program, while exception is design to catch the error and try to recover from it;
- Assertion is a quick easy way of assure the programming condition, used more often in unit testing and internal development.

Question: If a class directly extends Thowable, is it a checked or unchecked exception?

(checked. Only Error and RuntimeException and their subclasses are unchecked.)

References:

- Effective Java 2nd Edition
- Programming With Assertions

Tags: Assertion, Exception, Java
Posted in Java | 286 Comments »

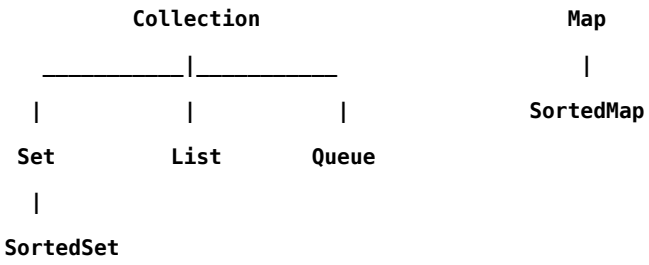
Prepare Java Interview 4: Java Collections Framework

March 24th, 2010

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

What is Java Collections Framework? Three things: interfaces, implementations, and algorithms.

Interfaces



- `Collection` — A collection represents a group of objects known as its *elements*.
- `Set` — a collection that cannot contain duplicate elements.
- `List` — an ordered collection than can contain duplicate elements.

- `Queue` — a collection used to hold multiple elements prior to processing.
- `Map` — an object that maps keys to values. Cannot contain duplicate keys.

SortedSet and SortedMap are merely sorted versions of `Set` and `Map`.

Collection interface is the super interface in all Java Collections Framework's interfaces, and it defines some common operations, like:

```
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);
boolean remove(Object element);
Iterator<E> iterator();
```

Set interface inherits the same operations as Collection (no extra). Set contains no duplicate items and no order. **SortedSet** interface is an sorted version (has order) of Set.

List is An ordered collection and typically allow duplicate elements. It defines additional operations specific for list.

- List keeps the order in which the elements are added.
- Element can be added in a specific location (index of the list).
- List has a special ListIterator which can travels the list reversely – so besides `hasNext(); next()` methods, it also has `hasPrevious(), previous()`.

Queue is designed for holding elements prior to processing. Notice there are two methods are provided for each of the operations:

	Throws exception	Returns special value	Comment (for typical case)
Insert	add(e) (true or exception)	offer(e) (true or false)	Add element as the last one
Remove	remove() (e or exception)	poll() (e or null)	Remove and return the first element
Examine	element() (e or exception)	peek() (e or null)	Return but not remove the first element

Queues typically order elements in a FIFO (first-in-first-out). An exception is PriorityQueue which are ordered according to their natural ordering.

Deque is a sub interface of Queue. It's a double ended queue that supports insertion and removal at both ends. (pronounced *deck*)

Map is an object that maps keys to values. A map cannot contain duplicate keys. And **SortedMap** is a sorted version of Map.

- Map does not extend Collection.
- Map stores key and value pair – a key maps to a value (no duplicate)
- Map has two operations: put and get

```
V put(K key, V value); // return previous value or null
V get(Object key);
```

- Map has its internal views of the stored key/value data:
  - All key-value pairs — entrySet() , so you can exam each pair:
  - All keys — keySet()
  - All values — values()

Example:

```
for (Map.Entry<String,Integer> entry : m.entrySet()) {
   String k = entry.getKey();
    Integer v = entry.getValue();
    System.out.printf("key=%s, value=%d%n", k, v);
}
```

Implementations

This table is a good summary of commonly used collection implementation classes.

General-purpose Implementations:

Interfaces	**Implementations**				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Notes about collection classes:

- HashSet and HashMap are fast, always use them if order is not required.
- TreeSet and TreeMap, implements SortedSet and SortedMap, offer the ordered Set and Map, but they are generally slow in comparison with the Hash version.
- Stack is a class while it should be an interface. It extends Vector so you know you should not use it any more (performance impact). Interestly, when you need stack, don't use Stack, use Deque instead:

Deque<Integer> stack = new ArrayDeque<Integer>();

- ArrayDeque is an implementation of Queue. Notice the java doc: This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.

Algorithms

The following algorithms are offered by java.util.Collections class, part of Java Collection Framework.

- `sort -- merge sort`

- `shuffle` — randomly permutes the elements in a `List`.
- `binarySearch`
- `reverse/fill/copy/swap/addAll (data manipulation)`
- `Composition` — reverses the order of the elements in a `List`.
- `min, max`

Notice that:

- Most of the algorithms apply to list based collections
- All public methods in Collections class take the form of static methods, like:

```
Collections.sort(list);
```

- Sometimes Comparator interface is used in sorting:

```
Comparator c = new Comparator<My>() {
    public int compare(My o1, My o2) {
      return o2.length() - o1.length();
    }
};
```

Collections.sort(list, c);

More on Collections Class

Do not confused with Collection interface (Collections ends with s).

Collection and Array conversion

List to Array:

```
Collection<String> c = new ArrayList<String>;
c.add("x");
c.add("y");
String[] a = c.toArray(new String[0]);

//a good way to print out array
System.out.println(Arrays.toString(a));
```

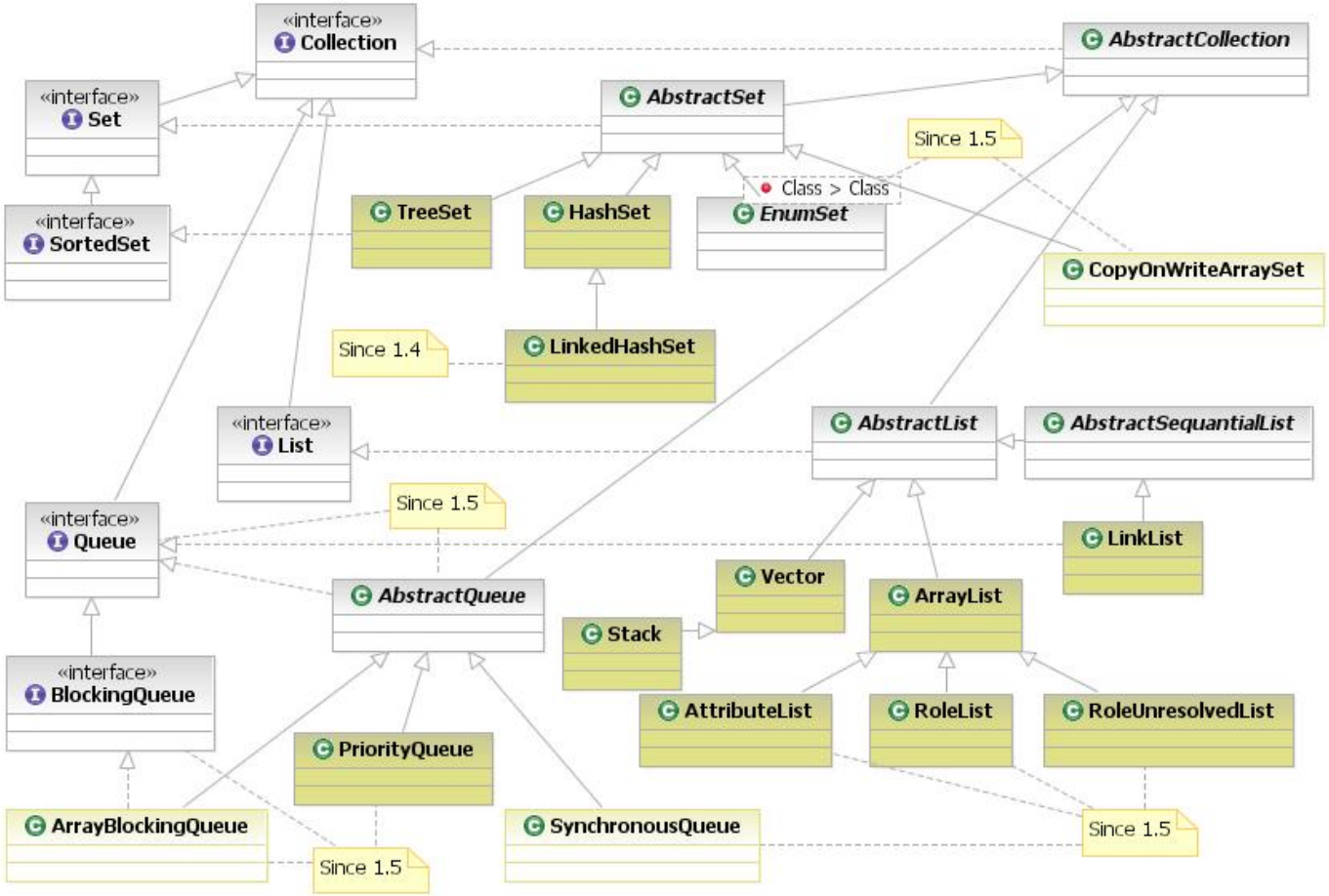
Array to List:

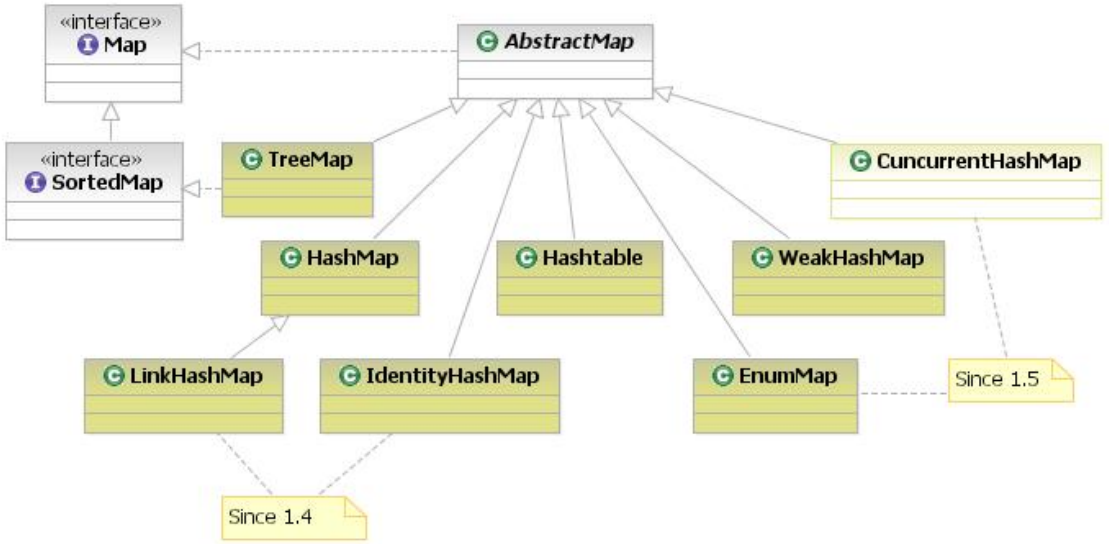
```
// a nice way to init list
List<String> l = Arrays.asList(new String[]{"a", "b", "c"});

// a good way to print out collection
System.out.println(l.toString());
```

```
Empty Collection
```

Collections.emptyList(), Collections.emptyMap(), Collections.emptySet() are handy to use (instead of using null value to represent an empty collection).

Class Diagram (UML)





References

The Java Tutorials – Collections

Tags: Collections, Java
Posted in Java | 242 Comments »

Prepare Java Interview 3: Java Object Oriented Programming

March 23rd, 2010

A Few Concepts

- ***Object-Oriented Programming*** (OOP) is a programming paradigm that users objects – data structures consisting of data fields and methods together with their interactions – to design computer programs.
- ***Object*** is a software bundle of related state and behavior.
- ***Class*** is a blueprint or prototype from which objects are created.
- ***Instance*** is the actual object created at runtime.
- ***Interface*** is a contract between a class and the outside world.
- ***Package*** is a namespace for organizing classes and interfaces in a logical manner.
- ***Abstraction*** is simplifying complex reality by modeling classes appropriate to the problem.
- ***Inheritance*** provides a powerful and natural mechanism for organizing and structuring your software.
- ***Encapsulation*** conceals the functional details of a class from objects that send messages to it.
- ***Polymorphism*** allows the programmer to treat derived class members just like their parent class's members.

- ***Decoupling*** allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction.
- ***Modularity*** is to design technique to make a software composed from separate parts, called modules.

Java Method

A ***method*** is a subroutine that is exclusively associated either with a class or with an object.

Instance Method

If a method is associated with a (class) instance, it is called Instance Method.

```
public void foo() {...}     // instance method
```

Class Method

When a method is associated with a class, it is a Class Method, or I call it Static Method because it has “static” keyword as its modifier.

```
public static void goo() {...}  // static method
```

Method Overriding

Overriding is the ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is “close enough” and then to modify behavior as needed.

For *instance method*, the method in a subclass with the same signature as an instance method in the superclass *overrides* the superclass’s method.

Notice that *the same signatures* means: it is the same in

- method name
- the number of parameters
- the types of the parameters
- return type
- exceptions in throw clause

Notice that the sub-type is allowed in return type (called *covariant return type*)

Notice that and exception type can also be a sup-type

Notice that the modifier can be less restricted.

Example:

```
protected List getList() throws IOException {...} // defined in super class

public ArrayList getList() throws FileNotFoundException {...} // overriding in subclass
```

However, the parameters type **cannot** be a sub-type. For example, it is a mistake to override equals like this:

```
public boolean equals(MyClass o){...} // Not overriding!
```

Tip: it is a good habit to always add @Override annotation in overriding method.

Overriding Behavior

Overriding is a useful technique that widely used, and the reason behind is the dynamic lookup (binding) capability.

Example:

```
class A { String foo(){return "A";} }

class B extends A { String foo(){return "B";} }

class C extends B { String foo(){return "C";} }

public static void main(String[] args) {

      List<A> list = new ArrayList<A>();

      list.add(new A());

      list.add(new B());

      list.add(new C());

      for (A a: list)

            System.out.print(a.foo());

}
```

It prints out: ABC

By the way, it becomes a convention to use an interface in such cases:

```
interface I { String foo();}
```

and implements I in A, B, and C.

```
      List<I> list = new ArrayList<I>();

      list.add(new A());

      list.add(new B());

      list.add(new C());

      for (I a: list)

            System.out.print(a.foo());
```

Method Hiding

For *class methods*, if a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass ***hides*** the one in the superclass.

The version of the overridden method that gets invoked is the one in the subclass. The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Example:

```
class A {
	public int foo(){return 1;}
	public static int goo(){return 1;}
}
class B extends A {
	public int foo(){return 2;}
	public static int goo(){return 2;}
}
public static void main(String[] args) {
	A a = new B();
	System.out.println(a.foo());
	System.out.println(a.goo());
}
```

It printed out:

```
2
1
```

The instance method foo() invoked is the one in the subclass B (so A's foo is overriden); the class method goo() gets invoked is the one in the superclass A (so B's goo is hidden).

Hiding is mostly unwanted behavior and should be avoided.

Tip: Always call static method in the recommended way, like: `B.foo()`

Overriding vs. Hiding

The following table summarizes what happens when you define a method with the same signature as a method in a superclass:

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	compile error
Subclass Static Method	compile error	Hides

Notice that, overloading method neither hide nor override – they are new methods, unique to the subclass.

Method Overloading

When more than one methods have the same name but different parameter list, it is called overloading.

Example:

```
public StringBuilder append(boolean b)

public StringBuilder append(int i)

public StringBuilder append(char c)
```

Overloading methods usually are defined in the same class. Define in a subclass is ok too, like this:

```
class A {  public void foo(){...} }

class B extends A { public int foo(int x) {...} }
```

Notice that two factors determine the overloading: same name and different parameters; the return type and throws clause does not matters – you can have the same or different ones.

Field Shadowing

Like static methods hiding, the fields can be shadowed in subclasses.

```
class A { int x = 1;}

class B extends A { int x = 2;}

public static void main(String[] args) {

      A a = new B();            // a points to an instance of B

      System.out.println(a.x);  // which x invoked?

}
```

Yes, it prints: 1

although “a” is now reference to an instance of class B.

You can cast it to force it invoke B’s x:

```
System.out.println(((B)a).x);
```

Now it prints 2.

So field shadowing like static method hiding, is more annoying than useful. It should be avoided.

Tags: Java, OOP
Posted in Java | 251 Comments »

Prepare Java Interview 2: Java Object

March 22nd, 2010

Object class is the root of the class hierarchy. All classes, include arrays, are the subclass of Object.

Object Creation

There are basically 4 ways that the object is created:

1. new operator

```
MyObject obj = new MyObject();
```

2. clone method

```
MyObject obj = oldObj.clone();
```

3. serialization

```
ObjectInputStream inputStream = ...
MyObject obj = inputStream.readObject();
```

4. reflection

```
Constructor c = ...
MyObject obj = c.newInstance();
```

In practice, sometimes use alternative is desired – For instance, the following are two examples that use Static Factory Method pattern (will discuss more in Java Design Pattern topic):

// return an immutable instance

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

```
// hide how the object is created
public static Calendar getInstance() {
  ...
  return cal;
}
```

(Effective Java: Item 1. Consider static factory methods instead of constructors).

Object Methods

There are a few methods defined in Object class that are important to know (and sometimes need to override):

equals

The default implementation only tests the identity equality – the reference is the same or not:

```
public boolean equals(Object obj) { return (this == obj); }
```

Sometimes, you need to override it. Example:

```
@Override
public boolean equals(Object obj){
    // 0. if (obj == null) return false; // not necessary
    // 1. check the reference - good optimization
    if (this == obj) return true;
    // 2. check the type (subtypes are ok)
    if (! (obj instanceof MyObject)) return false;
    // 3. cast the class - guaranteed to succeed
    MyObject o = (MyObject)obj;
    // check each 'significant' field
    if (this.x == o.x &&  // x is an int; s is String
        (this.s == o.s ||(this.s != null && this.s.equals(o.s))))
        return true;
    else
      return false;
}
```

4 Restrictions

- *reflexive*: for any reference value x, x.equals(x)
- *symmetric*: for any reference value x and y, x.equals(y) if and only if y.equals(x)
- *transitive*: for any reference values x, y, and z, if x.equals(y) and y.equals(z), then x.equals(z)
- *consistent*: for any reference values x and y, multiple invocations of x.equals(y) consistently return true (or false)

For any non-null reference value x, x.equals(null) should return false.

(Effective Java: Item 8. Obey the general contract when overriding equals)

hashCode

Returns a hash code value for the object. It's for used in hash-based collections. The default is converting the internal address.

The general contract of hashCode is:

- return the same consistent value in multiple invocation
- a and b must have the same hash code if a.equals(b)
- if a.equals(b) == false, a and b are not required to have different hash code. (but this is not recommended because of performance)
- if a field is not used in `equals`, then it must not be used in `hashCode`
- if a class overrides equals, it must override `hashCode`

Override Example:

```
public int hashCode() {
    return 31 + someNonNullField.hashCode();
                + (someOtherField == null ? 0 : someOtherField.hashCode());
}
```

(Effective Java: Item 9. Always override hashCode when you override equals)

Clone

You can copy a Java object by calling clone() methods, very handy:

```
MyObject copy = obj.clone();
```

This is done by:

- implement Cloneable interface
- override clone(), often change the modifier from protected to public

You can even do it without override clone(), just add implements Cloneable in your class.

However, you should know the negative side of it:

- It makes shallow-copy by default – object reference are copied but the objects that being referenced are not copied. (This likely causes problem – solution is to use deep-copy.)

It'd ok to use it if your class variables are all primitive type.

(Effective Java: Item 11. Override clone judiciously)

Finalize

Two things:

- There is no guarantee of when (or if) the garbage collector will call.

- Don’t count on it.

This method is supposed to be called by the garbage collector once determined the object is not reachable. So it’s the last place to do the clean up.

(Effective Java: Item 7. Avoid finalizers)

toString

Should be overrided. It’s pretty easy to do so with IDE (Eclipse, IntelliJ, etc) auto code generation feature.

(Effective Java: Item 10. Always override toString)

CompareTo

CompareTo is not a method defined in Object class. It is declared in ***java.lang.Comparable*** interface. It is something you may consider to implement when you write your own class. (Do you ever need to sort your objects?)

```
public interface Comparable<T> {

    public int compareTo(T o);

}
```

Return value: `==0 equal; >0 larger; <0 smaller`

Couple of things:

- Consistent with equals: a.equals(b) ==> a.compareTo(b) == 0;
- Its parameter is generic type
- Related Interface: ***java.util.Comparator***, compare two objects:

```
Interface Comparator<T> {

  int compare(T o1, T o2);

}
```

(Effective Java: Item 12. Consider implementing Comparable)

Tags: Java, Object
Posted in Java | 257 Comments »

Prepare Java Interview 1: Java Basics

March 19th, 2010

Here is a quick overview of Java language basic concepts.

Java Primitive Data Types

Data Type	Contents	Byte	bit	Range	Default Value (for fields)
byte	signed integer	1	8	-128 to 127 (inclusive)	0
short	signed integer	2	16	-32,768 to 32767 (inclusive)	0
int	signed integer	4	32	-2,147,483,648 to 2,147,483,647 (inclusive)	0
long	signed integer	8	64	9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	single-precision floating point	8	64	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	0.0f
double	double-precision floating point	8	64	0.0d	0.0d
char	single 16-bit Unicode character	2	16	'\u0000' – '\uffff' (0 – 65,535 inclusive)	'\u0000'
boolean	two possible values		1	true, false	fales

- local variables are not assigned default values (force you to init before use).
- float and double should never be used for precise values like currency (use BigDecimal instead).
- Relying field default values is generally considered bad programming style.
- byte and short type can be useful for saving memory in large arrays, where the memory savings actually matters. (don't use them otherwise)

Literals

```
byte b = 100;

short s = 10000;

int i = 100000;

int decVal = 26;     // The number 26, in decimal (default)

int octVal = 032;    // The number 26, in octal (starts with 0)

int hexVal = 0x1a;   // The number 26, in hexadecimal (starts with 0x)

double d1 = 123.4;   // ends with D or d, can be ommited (default)

double d2 = 1.234e2; // same value as d1 in scientific notation

float f1  = 123.4f;  // ends with F or f

Object o = null;     // null literal

char capitalC = 'C'; // single quote, can be Unicode (UTF-16) character
```

```
String str = "hell"; //double quote, can be Unicode (UTF-16) character

Class<String> cls = String.class; // class literal
```

Java Language Keywords (version 1.5)

abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while

- The keywords `const` and `goto` are reserved, but are not currently used.
- `true`, `false`, and `null` might seem like keywords, but they are actually literals; (cannot be used as identifiers)

- strictfp used to restrict the precision and rounding of floating point calculations to ensure portability

Controlling Access to Members of a Class

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier*	Y	Y	N	N
private	Y	N	N	N

*Also called package private

Java Interface

An Interface define new reference type whose members are classes, interfaces, constants, and abstract methods. This type has no implementations but other classes can implement.

- Interface can have sub interface
- A class can be declared to directly implement one or more interfaces.
- Interface modifier: public, abstract (implicitly), and none (package private). Don’t confuse with interface methods which are ALL implicitly public and abstract.
- All fields are implicitly public, static, and final.

- Method cannot be declared as static – java abstract method cannot be static.
- Method cannot be declared final, native, synchronization (implementation can).
- Method can be overload and override.

Interface overriding is tricky: since no implementation is allowed, the sub-interface can only differ in throwing a sub-set exception.

Abstract Class

A ***Class*** define new reference types and describe how they are implemented.

An *abstract class* is a class that contains abstract methods that are declared but not implemented.

- Abstract class cannot be instantiated, its sole purpose is to be extended by subclasses.
- A class may extend only one abstract class.

Interface vs. Abstract Class

Benefit of using interface: simulate multiple interface, separated from its implementation which can be change at any time. Public exposed API should use interface.

Abstract class have common code can be shared by all its subclasses. So the subclasses reuse the code.

Example, in Java Collection Framework, AbstractCollection class, provides a skeletal implementation of the Collection interface, such as size(), isEmpty(). And the implementation classes of Collection extend AbstratCollection and implement only those necessary methods.

Java Nested Class

Java has 5 types of classes including one top level class and 4 nested classes – a nested class is a class defined within another class.

Type	Scope	Inner	Note	Modifier
static nested class	Defined by its modifier	no	Can be treated the same as the top level class	public, protected, private, abstract, final
inner class	Defined by its modifier	yes	A nested class without static keyword.	public, proteced, private, abstract, final
local class	Local (inside method)	yes	Like local variables, defined inside a methods	abstract, final
anonymous class	where it is defined	yes	It is declared in the body of a method without naming it.	no

Note: Both static nested class and non-static inner class can be accessed as their modifier defined. You cannot pass parameters in anonymous class constructor.

Examples:

TopClass.StaticInnerClass snc = **new** TopClass.StaticInnerClass();

TopClass.InnerClass inc = aTopClassInstance.**new** TopClass.InnerClass();

public void methodFoo() { class LocalClass {…} …}

new Thread(new Runnable(){public void run(){…}}).start();

Static Method

There are two types of methods.

Instance methods are associated with an object and use the instance variables of that object.

Static methods use no instance variables of any object of the class they are defined in. This is typical of utility API. A good example is `Math`.

- Simplier and Easier to Understand
- Cannot overwriting – it's called *hidding*
- Reduce the Memory Footprint of an Object? No, the instance methods simply pass one more variable (the instance object reference).

Pass by value

Java pass parameters to a method by value.

- For primitive type parameters, Java copy the parameter's value (so the original values will not be changed).
- For Object type parameters, Java copy the reference value, so the original reference will not be changed.

Notice that *the original reference* will not be changed, but *the object that the original reference pointed to* could be changed.

Immutable object

Immutable object is an object whose state cannot be modified after it is created. Immutable objects are often useful in multiple thread programming.

Example

All of the primitive wrapper class in Java are immutable.

An instance of this class is *not* immutable, the item in the list can be modified.

```
class Cart {
  private final List items;
  public Cart(List items) {this.items=items;}
  public List getItems() { return items; }
}
```

The solution:

```
class ImmutableCart {
  private final List items;
  public ImmutableCart(List items) {
    this.items = Arrays.asList(items.toArray());
  }
  public List getItems() {
    return Collections.unmodifiableList(items);
  }
}
```

Instance Initializer and Static initializer

```
static {
    sInitFlag = 1;
}
```

- can have any number of static initialization blocks,
- can appear anywhere in the class body.
- called in the order that they appear in the source code.

Instance initializer blocks (without static keyword):

```
{
    x = 5; s = "hello";
}
```

The Java compiler copies initializer blocks into every constructor. So it can be used to share a block of code in multiple constructors.

Tags: Basics, Interview, Java
Posted in Java | 262 Comments »

« Older Entries

- Search for: Search

- **Pages**
  - About
- **Archives**
  - June 2010
  - March 2010
  - February 2010
- **Categories**
  - Java (12)
  - Uncategorized (1)
- **Java References**
  - Java 6 API
  - The Java Language Specification
  - The Java Tutorials
- **Meta**
  - Register
  - Log in
  - Valid XHTML
  - XFN
  - WordPress
