**第一种（懒汉，线程不安全）：**

public class Singleton {  
 private static Singleton *instance* = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

 这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。

**第二种（懒汉，线程安全）：**

public class Singleton {  
 private static Singleton *instance* = null;  
  
 private Singleton() {}  
  
 public static synchronized Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

 这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

**第三种（饿汉）：**

public class Singleton {  
 private static Singleton *instance* = new Singleton();  
  
 private Singleton(){}  
  
 public Singleton getInstance() {  
 return *instance*;  
 }  
}

 这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

**第四种（饿汉，变种）：**

public class Singleton {  
 private static Singleton *instance* = null;  
  
 static {  
 *instance* = new Singleton();  
 }  
  
 private Singleton(){}  
  
 public static Singleton getInstance() {  
 return *instance*;  
 }  
}

 表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

**第五种（静态内部类）：**

public class Singleton {  
 private static class SingletonHolder {  
 private static final Singleton *INSTANCE* = new Singleton();  
 }  
  
 private Singleton() {}  
  
 public static final Singleton getInstance() {  
 return SingletonHolder.*INSTANCE*;  
 }  
}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。

**第六种（枚举）：**

public enum Singleton {  
 *INSTANCE*;  
}

 这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

**第七种（双重校验锁）：**

public class Singleton {  
 private static volatile Singleton *instance* = null;  
  
 private Singleton(){}  
  
 public static Singleton getInstance() {  
 Singleton tInstance = *instance*;  
 if (tInstance == null) {  
 synchronized (Singleton.class) {  
 tInstance = *instance*;  
 if (tInstance == null) {  
 tInstance = new Singleton();  
 *instance* = tInstance;  
 }  
 }  
 }  
 return tInstance;  
 }  
}

 这个是第二种方式的升级版，俗称双重检查锁定，在JDK1.5之后，双重检查锁定才能够正常达到单例效果。