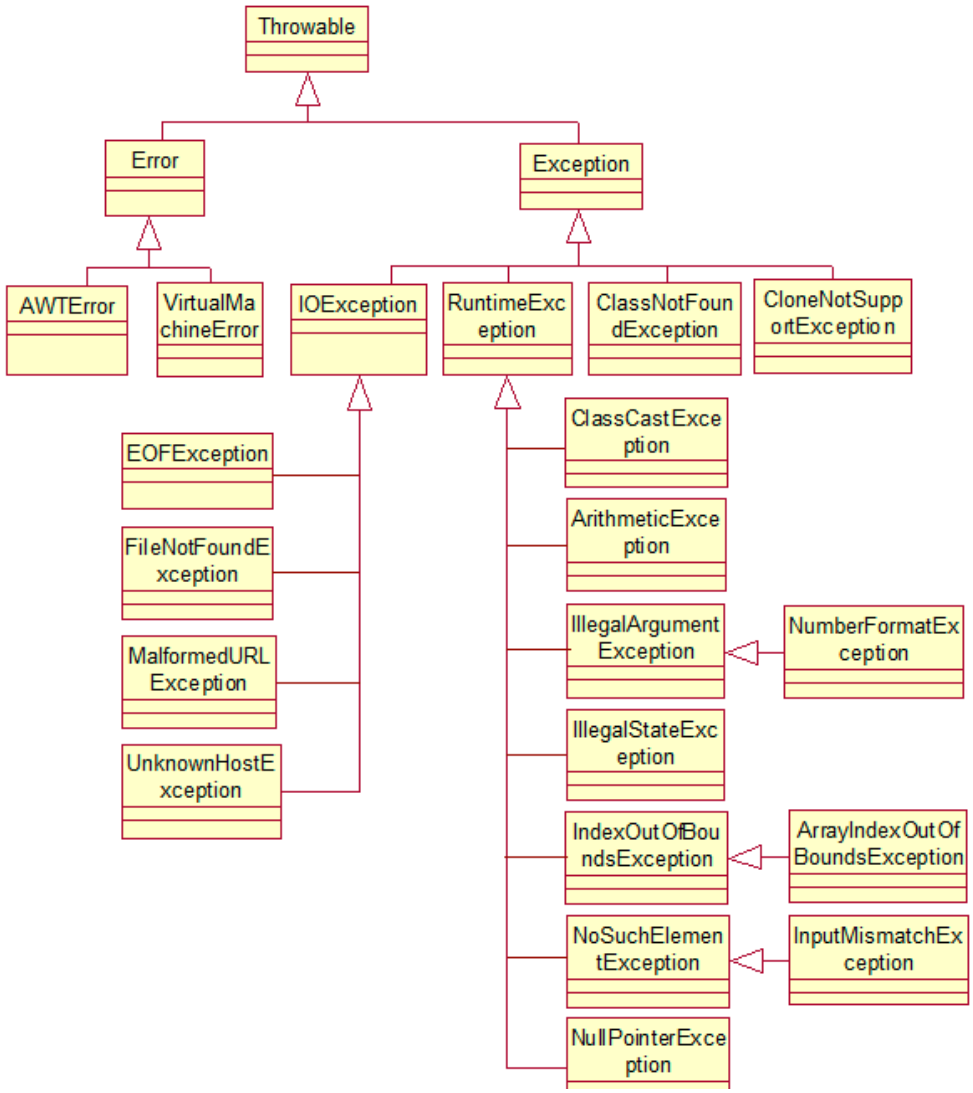
1、异常

1. 注意throw和throws的区别

   throws通常在函数的头后，声明该方法将要抛出该类型(子类型)的异常。throws不能单独使用，必须和throws[抛出异常，由caller处理]或catch[自己捕获异常，自己处理]

2. 捕获异常时,应该将子类异常的catch块放在父类catch块之前[具体放在之前]。因为，执行了父类异常，就不会执行子类异常

3. 

4. 异常分类:checked exception 、unchecked exception

   checked exception:程序编译时，必须要有try catch块或throws，例如io、连接

   unchecked exception:运行时异常，可以不用进行异常处理。一旦放生，程序直接结束运行，必须要修改代码。例如1/0，

5. 自定义异常
    1. 必须继承Throwable或其子类
    2. 如果要自定义异常类，则扩展Exception类即可，因此这样的自定义异常都属于**检查异常（checked exception）**。如果要自定义非检查异常，则扩展自RuntimeException。

       按照国际惯例，自定义的异常应该总是包含如下的构造函数：

       - 一个无参构造函数
       - 一个带有String参数的构造函数，并传递给父类的构造函数。
       - 一个带有String参数和Throwable参数，并都传递给父类构造函数
       - 一个带有Throwable 参数的构造函数，并传递给父类的构造函数。
6. 异常链

   当异常在catch块中被捕获，重新封装后在抛出，构成一条异常链

2、内部类

1. 内部类的优点

   1. 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

   2. 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

   3. 创建内部类对象的时刻并不依赖于外围类对象的创建。

   4. 内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

   5. 内部类提供了更好的封装，除了该外围类，其他类都不能访问。
2. 内部类的总体使用
    1. 内部类都隐式含有一个外部类的**this引用**，所以内部类可访问外部类的属性和方法(即使是私有)
    2. 内部类的创建

       OuterClass outerClass = new OuterClass ();

       OuterClass.InnerClass innerClass = outerClass.new InnerClass();
3. 内部类的分类
    1. 成员内部类
        1. 成员内部类中不能存在任何static的变量和方法
        2. 成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类

3. 推荐使用getxxx()来获取成员内部类，尤其是该内部类的构造函数无参数时

2. 局部内部类

   嵌套在方法，对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，到那时又不希望这个类是公共可用的，所以就产生了局部内部类，局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效

3. 匿名内部类

   1. 匿名内部类是没有访问修饰符的。
   2. new 匿名内部类，这个类[通常是**接口]**首先是要存在的。如果我们将那个InnerClass接口注释掉，就会出现编译出错。
   3. 当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为final|String。
   4. 匿名内部类是没有构造方法的。因为它连名字都没有何来构造方法。

4. 静态内部类

   使用static修饰的内部类我们称之为静态内部类，也称嵌套内部类。静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。没有这个引用就意味着：

   1、 它的创建是不需要依赖于外围类的。

   2、 它不能使用任何外围类的非static成员变量和方法。

3、对象拷贝问题

1.基本概念

1. 浅拷贝:对基本数据类型进行值传递,对引用类型数据进行引用传递的拷贝
2. 深拷贝:对基本数据类型进行值传递，对于引用数据类型则创建一个新的对象，并复制其内容

2.Obeject.clone()

在Java中所有的Class都继承自Object，而在Obejct中存在一个clone()方法，被声明为protected，所以我们可以在它的子类中使用clone()方法. 而无论深拷贝还是浅拷贝都需要实现clone()方法，

```
protected Object clone() throws CloneNotSupportedException {
    if (!(this instanceof Cloneable)) {
        throw new CloneNotSupportedException("Class " + getClass().getName() +
                                             " doesn't implement Cloneable");
    }

    return internalClone();
}

/*
 * Native helper method for cloning.
 */
private native Object internalClone();
```

可以看到，它的实现非常的简单，它限制所有调用 clone() 方法的对象，都必须实现 `Cloneable` 接口，否者将抛出 `CloneNotSupportedException` 这个异常。最终会调用 `internalClone()` 方法来完成具体的操作。而 `internalClone()` 方法，实则是一个 **native** 的方法。对此我们就没必要深究了，只需要知道它可以 `clone()` 一个对象得到一个新的对象实例即可。

```
/**
 * A class implements the <code>Cloneable</code> interface to
 * indicate to the {@link java.lang.Object#clone()} method that it
 * is legal for that method to make a
 * field-for-field copy of instances of that class.
 * <p>
 * Invoking Object's clone method on an instance that does not implement the
 * <code>Cloneable</code> interface results in the exception
 * <code>CloneNotSupportedException</code> being thrown.
 * <p>
 * By convention, classes that implement this interface should override
 * <tt>Object.clone</tt> (which is protected) with a public method.
 * See {@link java.lang.Object#clone()} for details on overriding this
 * method.
 * <p>
 * Note that this interface does <i>not</i> contain the <tt>clone</tt> method.
 * Therefore, it is not possible to clone an object merely by virtue of the
 * fact that it implements this interface.  Even if the clone method is invoked
 * reflectively, there is no guarantee that it will succeed.
 *
 * @author  unascribed
 * @see     java.lang.CloneNotSupportedException
 * @see     java.lang.Object#clone()
 * @since   JDK1.0
 */
public interface Cloneable {
}
```

而反观 **Cloneable** 接口，可以看到它其实什么方法都不需要实现。对他可以简单的理解只是一个标记，是开发者允许这个对象被拷贝。

3.实现深拷贝方法

1. 序列化
2. 对引用类型成员变量继续复写colne()
3. 实例

浅拷贝:

```
package copy;

public class School {
  public String schoolName;
  public String schoolAddress;
  public School(String schoolName, String schoolAddress) {
    super();
    this.schoolName = schoolName;
    this.schoolAddress = schoolAddress;
  }
  @Override
  public String  toString() {
    return "schoolName:"+schoolName+" " +"schoolAddress:
"+schoolAddress ;
  }
}


package copy;

public class Student implements Cloneable {
  public String studentName;
  public int age;
  public School school;

  public Student(String studentName, int age, School school){
    this.studentName = studentName;
    this.age = age;
    this.school = school;
```

```
  }
  @Override
  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return null;
  }
  public String  toString() {
    return "studentName:" + studentName+"  "+"age:" +age + " " + school;
  }
}


package copy;

public class Test {

  public static void main(String[] args) {
    Student student = new Student("xiaoming", 20, new School("HUST", "GuanShanKou"));
    Student  student1 = (Student)student.clone();
    System.out.println("student.hashCode: "+student.hashCode());
    System.out.println("student1.hashCode:"+student1.hashCode());

    student1.age = 21;
    student1.school.schoolName = "WU";
    student1.school.schoolAddress = "DongHu";

    System.out.println("student: "+student);
    System.out.println("student1: "+student1);
  }
}
/*
输出
student.hashCode: 1476011703
student1.hashCode:1603195447
student: studentName:xiaoming  age:20 schoolName:WU schoolAddress: DongHu
```

```
student1: studentName:xiaoming  age:21 schoolName:WU schoolAddress:DongHu
分析:
知识补充:由于不同对象的hashcode可能相同，所以不能依据hashcode相等判断两个对象相同。但是可以依据hashcode的不等判断两个对象不同，因为相同对象的hashcode一定相同。
综上输出结果，可以判断出:student.clone()确实创建了一个新的对象，但是student1.schoool和stdent.school指向同一实体对象。所以，只是基本数据数据类型成员变量得到了赋值，而域成员变量赋值的是只是引用，所以这只是浅拷贝
*/
```

深拷贝

```
package copy;

public class School  implements Cloneable{
  public String schoolName;
  public String schoolAddress;
  public School(String schoolName, String schoolAddress) {
    super();
    this.schoolName = schoolName;
    this.schoolAddress = schoolAddress;
  }
  @Override
  public String  toString() {
    return "schoolName:"+schoolName+" " +"schoolAddress:"+schoolAddress ;
  }
  /*对域成员变量复写clone()实现深拷贝*/
  @Override
  public Object clone() {
    try {
      //调用Object.clone()方法实现对基本数据类型的成员变量的拷贝
      return super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return null;
  }
}

package copy;
//包含应引用成员变量的类
```

```
public class Student implements Cloneable {
  public String studentName;
  public int age;
  public School school;

  public Student(String studentName, int age, School school){
    this.studentName = studentName;
    this.age = age;
    this.school = school;
  }
  @Override
  public Object clone() {
    try {
      Student student = (Student)super.clone();//对自身基本数据类型成员变量拷贝
      student.school = (School)school.clone();//对引用类型成员变量对象中的基本成员变量拷贝
      return student;
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return null;
  }
  public String  toString() {
    return "studentName:" + studentName+"  "+"age:" +age + " " + school;
  }
}

package copy;

public class Test {

  public static void main(String[] args) {
    Student student = new Student("xiaoming", 20, new School("HUST", "GuanShanKou"));
    Student  student1 = (Student)student.clone();
    System.out.println("student.hashCode: "+student.hashCode());
    System.out.println("student1.hashCode:"+student1.hashCode());

    student1.studentName = "lihu";
    student1.age = 21;
```

```
    student1.school.schoolName = "WU";
    student1.school.schoolAddress = "DongHu";

    System.out.println("student: "+student);
    System.out.println("student1: "+student1);
  }
}
/*
输出:
student.hashCode: 1476011703
student1.hashCode:1603195447
student: studentName:xiaoming  age:20 schoolName:HUST schoolAddress:
GuanShanKou
student1: studentName:lihu  age:21 schoolName:WU schoolAddress:
DongHu
分析:student1.schoool和stdent.school指向不同实体对象，多以这是深拷贝
*/

```

序列化:

利用序列化来做深复制,把对象写到流里的过程是序列化（Serilization）过程，而把对象从流中读出来的过程则叫做反序列化（Deserialization）过程。应当指出的是，写在流里的是对象的一个拷贝，而原对象仍然存在于JVM里面。利用这个特性，可以做深拷贝 。

```
package copy;

import java.io.Serializable;

public class School  implements Serializable{
  //里面代码不变
}


package copy;

import java.io.Serializable;

public class Student  implements Serializable{
  //里面代码不变
}

```

```java

package copy;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
//深度复制对象的模板
 public class CloneUtils {
    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T obj){
        T cloneObj = null;
        try {
            //ByteArrayOutputStream:目的地：实际是一个byte[]
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            ObjectOutputStream obs = new ObjectOutputStream(out);
            obs.writeObject(obj);//写入byte[]
            obs.close();

            //从byte[]中读取出对象
            ByteArrayInputStream ios = new
ByteArrayInputStream(out.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(ios);
            //返回生成的新对象
            cloneObj = (T) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return cloneObj;
    }
}

public class Test {

  public static void main(String[] args) {
    Student student = new Student("xiaoming", 20, new School("HUST",
                                                    "GuanShanKou"));
    Student student1 = (Student)CloneUtils.clone(student);
    //...
  }
```

```
}
/*
输出:
student.hashCode: 466002798
student1.hashCode:93122545
student: studentName:xiaoming  age:20 schoolName:HUST schoolAddress:
GuanShanKou
student1: studentName:lihu  age:21 schoolName:WU schoolAddress:
DongHu

*/
```

4、关键字final

1. 修饰成员变量和局部变量。可以赋值一次，然后不能对其进行修改。通常在构造器和声明是赋值
2. 修饰形参,表明该参数在方法体内不能不被修改
3. 修饰成员函数,表明 该函数不能被重写
4. 修饰class，表明该类不能被继承

5、Comparator、Comparable

先上源码

```
public interface Comparable<T> {
    public int compareTo(T o);
}

public interface Comparator<T> {
    int compare(T o1, T o2);
    //...
}
```

区别:实现Comparable接口的类的对象实例可以直接比较;可以直接传入Collection.sort(List list)中;

而Comparator可以看成是一种算法,实现对数据和算法的分离。显然可以不用修改原始类代码而达到可以排序的效果，更加灵活。

6、String StringBuilder StringBuffer

1、String VS StringBuilder

源码

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    /** The value is used for character storage. */
    private final char value[];    / /
    /** Cache the hash code for the string */
    private int hash; // Default to 0
}

abstract class AbstractStringBuilder implements Appendable,
CharSequence {
     /**
     * The value is used for character storage.
     */
    char[] value;
    /**
     * The count is the number of characters used.
     */
    int count;
    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }
}
```

String:

```
    1. 不可变对象:重建对象
    2. 由final修饰:不能被继承
    3. 属性value[]由final修饰: 引用不能被修改
```

StringBuilder

1. 可变性:不用创建新对象
2. 属性value[]没有被final修饰
3. append("string")返回自身对象，没有创建新对象;

重点：String 每加一次就在对创建一个新对象

2、StringBuilder StringBuffer

源码

```
/**
     * @since      1.5
     */
    @Override
    public synchronized void trimToSize() {
        super.trimToSize();
    }
    /**
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @see        #length()
     */
    @Override
    public synchronized void setLength(int newLength) {
        toStringCache = null;
        super.setLength(newLength);
    }
```

StringBuffer是线程安全的

StringBuilder不是线程安全的

所以StringBuilder的性能是高于StringBuffer的

3、三者的使用

1. 少量数据用String；单线程操作大量数据，用StringBuilder;多线程操作大量数据，用StringBuffer
2. 不能用String反复做+操作，严重影响性能
3. 由于StringBuffer是线程安全的，所以放在实例的变量上；而StringBuilder不是线程安全的，所以可以放在方法内作局部变量

7、集合

1、集合的由来

数组发展到集合（当然数组也是集合），不知道存储对象的容量

2、集合是什么

集合类放在java.uti包中，用来存放对象的容器

1. 集合只能存放对象。当存放的是基本数据类型的变量，将自动对其拆装箱
2. 集合存放的是对象的引用，而不是对象本身（依旧在堆里）
3. 集合支持泛型，所以可以存放不同类型的对象
4. 容量可变

3、Iterator

迭代器，Java集合的顶层接口（不包括map系列集合，Map接口是map系列集合的顶层接口）

Object next();返回迭代器刚越过的元素的引用，返回值类型为Obejct，所以要强转

Boolean hasNext();判断是否还有元素可遍历

void remove();删除迭代器刚越过的元素

除了 map 系列的集合，我们都能通过迭代器来对集合中的元素进行遍历。那是因为

Collection 接口继承的是类接口Iterable，而Iterable接口中封装了Iterator接口，

所以**只要是Collection的子类都可用Iterator来遍历**

4、List

有序，元素可以重复的集合

List接口直接继承Collectio接口，三个典型实现类：ArrayList、Vector、LinkedList

ArrayList 动态数组

LinkedList 双向链表

5、Set

无序，不可重复的集合

HashSet 基于 HashMap 实现的，元素保存在Keys中，value=PRESENT

hashCode-->equals()

linkedHashSet 哈希表和链接列表，保持顺序

TreeSet 底层是TreeMap

如何排序和唯一性:元素必须实现Comparable接口，重写compareTo()

6、Map

HashMap 数组+链表/数组+红黑树

视图:keySet() values() entrySet()

允许一组key-value=(null,null)

可以使用Collections.synchronizedMap(HashMap map)使HashMap变为同步;

HashTable

线程同步,不允许(null,null)

linkedHashMap

保证entry插入的顺序

TreeMap

默认规则按照：key的字典顺序来排序

自定义:实现Comparator接口，传入TreeMap的构造方法中

7.Map的遍历

Map->entrySet->Set->Iterator

```
        Map<String,Object> map = new HashMap<>();
        map.put("Name","Wang");
        map.put("Age",20);
        Set set = map.entrySet();
        Iterator it = set.iterator();
        while(it.hasNext()){
            Map.Entry entry = (Map.Entry) it.next();
            System.out.println(entry.getKey() + ""+ entry.getValue());
        }
```

8、I/O

1、File

```
File•(String pathname)
boolean delete()
String  getName()
String  getPath()
 //遍历一个文件夹下所有的文件
  void traverseDiretory(File file,int level){
  if(file.isDerectory()){
    for(File f:file.listFiles()){
      traverseDirectory(f,level+1);
    }
  }
  else{
    for(int i=0;i<level;i++)
      sout("/t");
    sout(f.getName);
  }
}
```

2、FileInputSream

```
//构造器
FileInputStream•(File file)
FileInputStream•(String name)

 //每次读一个字节。读出下一个字节，遇到文件末尾返回-1
  public int read();
 //一次读取b.lenegth的长度的字节到b[]中。遇到文件末尾返回-1
  public int read(byte[] b);
 //从偏移off开始读len长度的字节到b[]中，遇到文件末尾返回-1
 public int read(byte[] b, int off, int len)

```

3、FileOutputStream

```
//写入将b[]中b.len的字节写到文件中
void  write(byte[] b)
void  write(byte[] b, int off, int len)
```

4、BufferedInputStream BufferedOutputStream

BufferedInputStream**继承** FilterInputStream**继承** InputStream

BufferedOutputStream**继承** FilterInputStream**继承** OutputStream

```
BufferedInputStream(InputStream in)
BufferedOutputStream(OutputStream out)
```

5、FileReader

```
FileReader•(String fileName)
FileReader•(File file)
//Reader中的方法
//将reder中的字节存到cbuf[]中，返回读取的长度，遇到文件末尾返回-1
int read(char[] cbuf)
```

6、FIleWriter

```
FileWriter•(File file)
FileWriter•(File file, boolean append)
FileWriter•(String fileName)

//Writer中的方法
void  write•(char[] cbuf)
void  write•(String str)
```

7、BufferedReader BufferedWriter

BufferedReader 继承Reader

BufferedWriter继承Writer

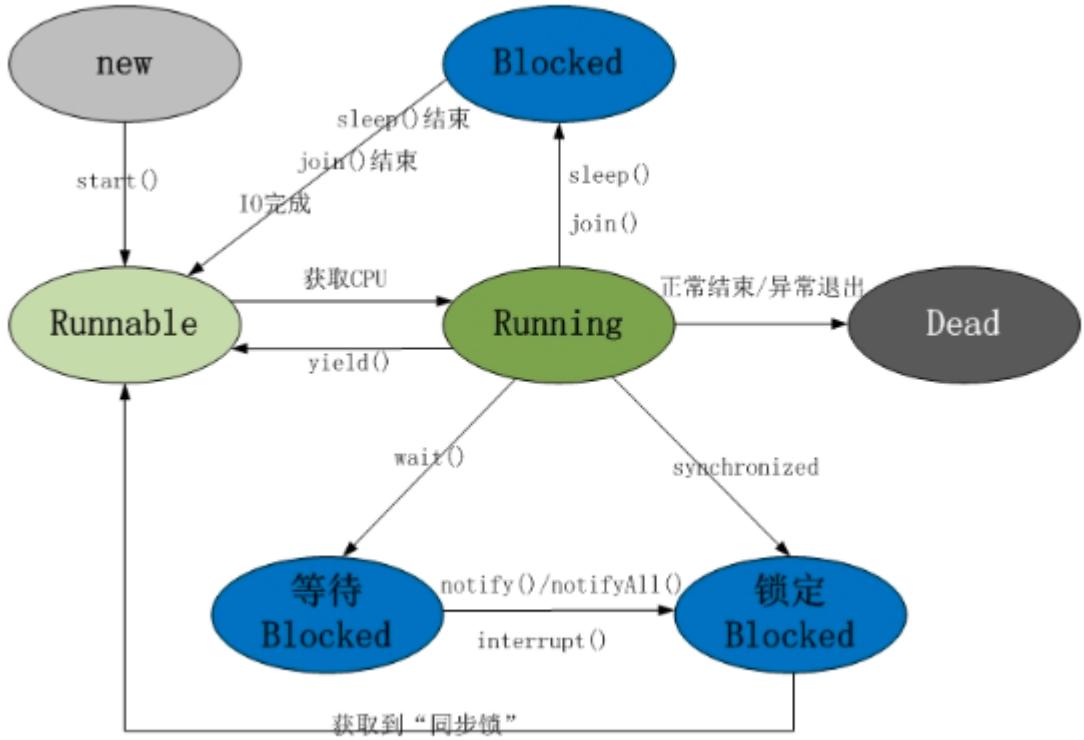
```
BufferedReader•(Reader in)
BufferedWriter•(Writer out)
```

```
public final class MyUtil {

    // 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)
    private MyUtil() {
        throw new AssertionError();
    }

    /**
     * 统计给定文件中给定字符串的出现次数
     *
```

```
     * @param filename  文件名
     * @param word 字符串
     * @return 字符串在文件中出现的次数
     */
    public static int countWordInFile(String filename, String word) {
        int counter = 0;
        try (FileReader fr = new FileReader(filename)) {
            try (BufferedReader br = new BufferedReader(fr)) {
                String line = null;
                while ((line = br.readLine()) != null) {
                    int index = -1;
                    while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {
                        counter++;
                        line = line.substring(index + word.length());
                    }
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return counter;
    }
}
```

9、线程

1、状态转换图



new: 当线程创建后，i即进入了新建状态，Thread t = new MyThread();

Runnable: 当调用线程对象的star()方法，线程即进入了就绪状态，等待CPU调度

Running: 线程被CPU执行。PS:就绪状态是到运行状态的唯一入口

Blocked：线程进入阻塞状态，失去CPU执行权

等待阻塞:wait()

同步阻塞:synchronized

其他阻塞:sleep/join/I/O

pS:调用yield()，线程直接进入Runnable状态

Dead:线程run()方法执行完毕

2、创建线程的三种方法

1. 继承Thread类，重写该类的run()方法
2. 实现Runnable接口，重写该接口的run()方法，将该实现类的实例传入Thread类的target来创建线程

3. 使用Callable和Future接口。先实现**Callable**接口的，重写call()方法，并使用FutureTask类来包赚Calllable实现类的对象，然后以此FutureTask对象作为Thead对象的target来创建线程.

   注意:call()方法由返回值其类型为T

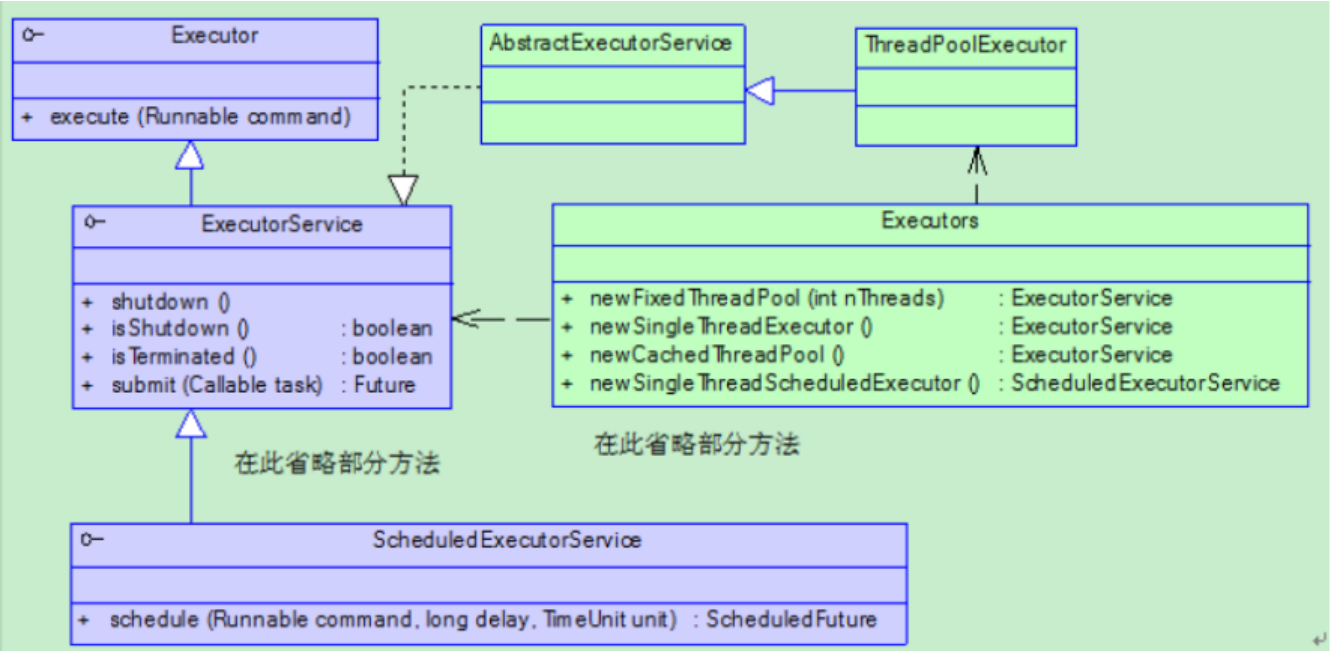
```
FutureTask<V> implements RunnableFuture<V>
RunnableFuture<V> extends Runnable, Future<V>
```

3、wait()和sleep()区别

wait(): Object类中定义的**实例方法。**wait需要组合synchronized使用，wait时会释放掉拿到synchronized 锁

sleep(): Thread类中的静态方法，作用是让当前线程进入休眠状态，以便让其他线程有机会执行。进入休眠状态的线程**不会释放它所持有的锁**

4、Executor结构图



Future、Callable

```java
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
TimeoutException;
}
```

5、Lock

```java
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws
InterruptedException;
    void unlock();
    Condition newCondition();
}
//使用模式
Lock lock = ...;//定义为实例变量

lock.lock();
try{
    //处理任务
}catch(Exception ex){

}finally{
    lock.unlock();//释放锁
}
```

ReentrantLock

唯一一个实现Lock的类

ReadWriteLock

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

ReentrantReadWriteLock实现了ReadWriteLock接口

10、JDBC

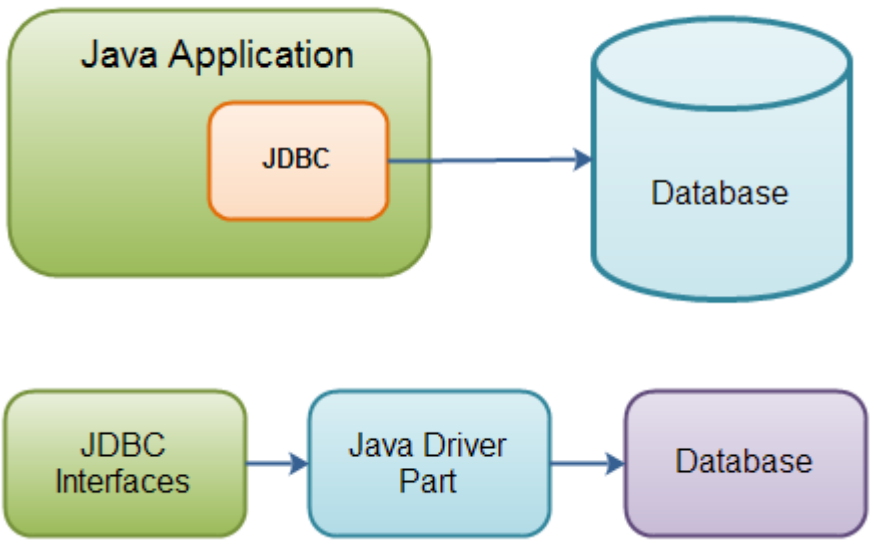
1、整体图

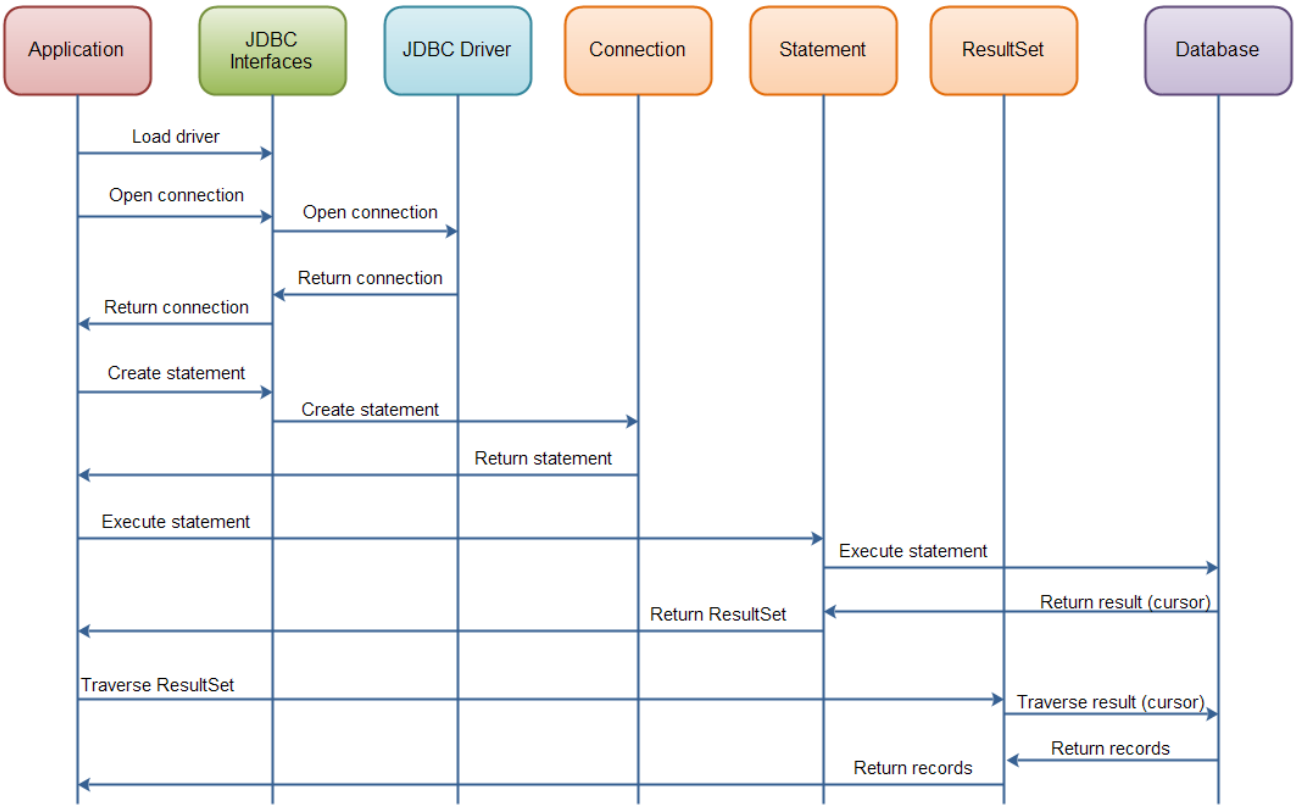
JDBC Driver: Java类的集合，它实现了JDBC接口。不同数据库提供商都需要提自己的JDBC Driver。它可以让你连接到特定的数据库

Connection:一旦JDBC Driver载入和初始化后，可以通过JDBC API 和 JDBC Driver来获取一个Connection,它可以然你连接到数据库。对数据库的所有操作都是经过Connection来完成的。当然，一个数据库可以有多个Connection实例

Statement:操作数据库的语句

ResultSet:数据库查询结果集





2、Connection

1. 载入JDBC Driver

```
1  Class.forName("driverClassName");
2  java6可以不用手动载入JDBC Driver;
3 没有必要在每一次获取Connection之前载入driver，只需在第一次获取之前加载一次即可
```

2. 获取连接

调用java.sql.DriverManager中getConnection()来获取

```java
//总共有三种方法
//第一种...
//第二种 常用
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";
Connection connection = DriverManager.getConnection(url, user,
password);
//第三种 不推荐，因为属性名应数据库不同而可能不同
String url      = "jdbc:h2:~/test";   //database specific url.
Properties properties = new Properties( );
properties.put( "user", "sa" );
properties.put( "password", "" );
Connection connection = DriverManager.getConnection(url, properties);
```

3. 关闭Connection

   由于Connection在打开状态会占用系统资源，所以要在它不需要的时候将其关闭

   1. 直接关闭

      ```java
      connection.close();
      ```

   2. try(){...}

      JDK7以后自动关闭机制

      ```java
      try(Connection connection =
          DriverManager.getConnection(url, user, password)) {
          //use the JDBC Connection inhere
      }
      ```

4. setAutoCommit()

   ```java
   //自动提交 每条更新语句执行完后立即提交，而无需等待事务结束commit
   //如果不处于自动提交模式，则必须通过调用Connection commit()方法显式提交每个
   数据库事务。
   connection.setAutoCommit(true);//connection默认状态
   //后面transation细谈
   connection.setAutoCommit(false);
   ```

3、查询数据库

statement.executeQuery(sql);

```
Statement statement = null;

try{
    //用Connection来创建SQL语句
    statement = connection.createStatement();
    ResultSet result = null;
    try{
        //SQL语句
        String sql = "select * from people";
        //用Statemente执行SQL语句，并将结果返回给ResultSet
        ResultSet result = statement.executeQuery(sql);

        //遍历ResultSet
        while(result.next()) {//如果还有下一行，result.next()返回true
            //result<==>row
            //result.getXXX("columnName);获取当前行列名为colnmnName的值
            String name = result.getString("name");
            long   age  = result.getLong("age");
            System.out.println(name);
            System.out.println(age);
        }
    } finally {
        if(result != null) result.close();//注意关闭result
    }
} finally {
    if(statement != null) statement.close();//注意关闭statement
}
```

4、更新数据库

包括：update、delete

statement.executeUpdate(sql);

```
//update操作
Statement statement = connection.createStatement();
String    sql       = "update people set name='John' where id=123";
int rowsAffected    = statement.executeUpdate(sql);
//delete操作
Statement statement = connection.createStatement();
String    sql       = "delete from people where id=123";
int rowsAffected    = statement.executeUpdate(sql);
```

5、PreparedStatement

1. 创建PreparedStatement **connection.prepareStatement(sql);**

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
```

2. 插入参数到PreparedStatement **preparedStatement.setXXX(index, value);**

```
String sql = "select * from people where firstname=? and lastname=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "John");
preparedStatement.setString(2, "Smith");
```

3. 执行PreparedStatement **preparedStatement.executeQuery()/executeUpdate()**
4. 重用PreparedStatement

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
//传递参数
preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);
//执行更新，放回受影响的记录条数
int rowsAffected = preparedStatement.executeUpdate();
```

```
//重新给preparedStatement传递参数
preparedStatement.setString(1, "Stan");
preparedStatement.setString(2, "Lee");
preparedStatement.setLong  (3, 456);
//再次执行preparedStatement
int rowsAffected = preparedStatement.executeUpdate();
```

不要使用Statement,而要使用性能更高的PreparedStatement

6、批量更新

preparedStatement.addBatch();

preparedStatement.executeBatch();

```
String sql = "update people set firstname=? , lastname=? where id=?";


PreparedStatement preparedStatement = null;
try{
    preparedStatement =
            connection.prepareStatement(sql);

    preparedStatement.setString(1, "Gary");
    preparedStatement.setString(2, "Larson");
    preparedStatement.setLong  (3, 123);

    preparedStatement.addBatch();

    preparedStatement.setString(1, "Stan");
    preparedStatement.setString(2, "Lee");
    preparedStatement.setLong  (3, 456);
    //添加
    preparedStatement.addBatch();

    //批量执行
    int[] affectedRecords = preparedStatement.executeBatch();

}finally {
    if(preparedStatement != null) {
        preparedStatement.close();
    }
}
```

注意各个语句是**分开执行**的，所以可能会执行成功，有的可能执行失败，这就破坏了原子性。可以将他们放在一个事务中执行，从而保证原子性

7、Transaction

简单介绍一下事务：一系列数据库操作的集合，他们要么全做，要么全不做，即满足**原子性。**

1. 开启一个事务

```
connection.setAutoCommit(false);
```

2. 回滚

   开启事务后，后面的查询和更新就都属于一个事务中，一旦有一个操作执行失败，就要回滚事务

```
connection.rollback();
```

3. 提交

事务中所有操作都执行成功了，就可提交事务了。一旦提交事务，所有所有操纵执行的结果都保存到数据库中，永久不会改变，即满足**永久性**

```
connection.commit();
```

模板

```
Connection connection = ...
try{
    connection.setAutoCommit(false);

    // create and execute statements etc.

    connection.commit();
} catch(Exception e) {
    connection.rollback();//发生异常就回滚
} finally {
    if(connection != null) {
        connection.close();
    }
}
```

11、类的加载过程

静态内容是随着类加载而加载的，jvm的代码编译运行过程是:class文件编译->类的加载->执行

1. 静态代码块只加载一次
2. 构造方法每创建一个实例就加载一次
3. 静态方法在调用时才会执行

如果类没有被加载

1. 先执行父类的静态代码块he静态变量的初始化
2. 执行子类的静态代码块和静态变量的初始化
3. 执行父类实例变量的初始化
4. 执行父类的构造器(有构造代码块就先执行构造代码块)
5. 执行子类实例变量的初始化
6. 执行子类的构造器

如果类被加载过

静态代码块和静态变量不会再次被加载。再创建对象时，只会执行实例变量初始化和构造方法。

构造代码块

给对象进行初始化，并且先于构造方法执行。而构造代码块与构造器的区别在于：构造代码块是给所有对象统一初始化，而构造方法是给特定对象初始化。