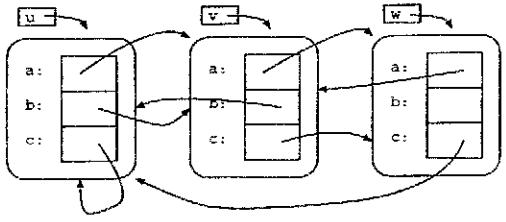
Lösungsvorschläge zur Klausur zum Kurs 1814
Sommersemester 2000

Aufgabe 1

a) Das Programm wird nicht ohne Beanstandungen übersetzt, denn eine überschreibende Methode kann in ihren Zugriffsrechten nicht weiter eingeschränkt sein als die Überschriebene. Subtypobjekte müssen an allen Stellen benutzt werden können, an denen auch Supertypobjekte benutzt werden dürfen. Bei strengeren Zugriffsrechten ist dies nicht immer möglich.

b) `System` ist eine Klasse. `err` ist ein Klassenattribut der Klasse `System`. `print` ist eine Methode, die im Typen des Klassenattributes `err` existieren muss. Insgesamt stellt die gegebene Java-Anweisung einen Aufruf der Methode `print` auf dem Objekt dar, das durch `System.err` referenziert wird, dar.

c) Die folgende Grafik zeigt das gewünschte Geflecht. Das Attribut `w.b` wird nicht auf einen bestimmten Wert gesetzt und enthält somit den Wert `null`.



d) Es wird natürlich „`Ich bin ein A Objekt`" und „`Ich bin ein B Objekt`" ausgegeben. Dazu wird zunächst ein B-Objekt erzeugt und die Referenz darauf in einer Variable vom Typ A gespeichert. Der Methodenaufruf `a.m()` wird dynamisch gebunden und die Methode m aus B wird ausgeführt. Beim Aufruf `super.m()` wird als impliziter Parameter das aktuelle this Objekt benutzt, also das B-Objekt und der super-Aufruf wird statisch gebunden. Da ein B-Objekt auch ein A-Objekt ist, gilt `this instanceof A` und `this instanceof B` und es wird zweimal etwas ausgegeben.

e) Es wird

```
K
L
M
```

ausgegeben, denn in jedem der Konstruktoren wird implizit als Erstes der entsprechende Konstruktor des Supertypen aufgerufen, d.h. hier der Konstruktor mit leerer Parameterliste.

f), g) An der Ausführung des folgenden Programms sieht man, dass der Attributzugriff statisch und der Methodenaufruf dynamisch gebunden wird.

```
class AA {
    String s = "s in AA";
    String m() {
        return "m in AA";
    }
    public static void main(String[] argv) {
        AA a = new BB();
        System.out.println(a.m());
        System.out.println(a.s);
    }
}

class BB extends AA {
    String s = "s in BB";
    String m() {
        return "m in BB";
    }
}
```

Ausgabe ist:

```
m in BB
s in AA
```

An diesem Beispiel kann man die geforderten Eigenschaften aus f) und g) ablesen: Der Methodenaufruf `a.m()` wird dynamisch gebunden. Die Variable `a` ist vom statischen Typ A, referenziert jedoch ein B-Objekt. Daher wird `m in BB` ausgegeben. Im Gegensatz dazu wird bei der zweiten Ausgabe `s in AA` ausgegeben, da bei der Bindung von Attributen nur der statische Typ (also AA) und nicht der dynamische Typ (also BB) beachtet wird. Attributzugriffe können so zur Übersetzungszeit gebunden werden.

Aufgabe 2

a) Das folgende Programm ersetzt die dynamisch gebundene Methode im gegebenen Programm nach dem angegebenen Muster:

```
class Dynamic {
    static String m(Dynamic THIS) {
        if(THIS instanceof S1)      return S1.m((S1)THIS);
        else if(THIS instanceof S2) return S2.m((S2)THIS);
        else return "m in Dynamic";
    }

    public static void main(String[] s) {
        Dynamic[] d = new Dynamic[4];
        d[0] = new Dynamic();
        d[1] = new S1();
        d[2] = new S2();
        d[3] = new S3();
        for(int i=0;i<d.length;i++) System.out.println(Dynamic.m(d[i]));
    }
}

class S1 extends Dynamic  {
    static String m(S1 THIS) {
       return "m in S1";
    }
}

class S2 extends Dynamic  {
    static String m(S2 THIS) {
       if(THIS instanceof S3) return S3.m((S3)THIS);
       else return "m in S2";
    }
}
```

```
class S3 extends S2 {
    static String m(S3 THIS) {
       return "m in S3";
    }
}
```

b) Das Hauptproblem bei Implementierungen dieser Art ist, dass die Wiederverwendung des Programmcodes erschwert, wenn nicht gar unmöglich gemacht wird. Wenn man die dynamische Bindung explizit ausprogrammiert, muss man zu dem Zeitpunkt, an dem das Programm erstellt wird, alle Typen des Programms kennen. Bei einer Programmerweiterung müssen dann ggf. betroffene Programmstellen um entsprechende Verzweigungen und Aufrufe für die neu hinzugekommenen Typen erweitert werden. Weiterhin muessen alle Typinformationen zur Laufzeit des Programms zugänglich sein und somit explizit verwaltet werden..

c) Die Ersetzung einer Methode *M* kann wie folgt vorgenommen werden:

- Ersetze *M* und alle Methoden, die *M* z.B. in einer Klasse `K` überschreiben durch `static`-Methoden, die zusätzlich zu den Parametern explizit einen impliziten Parameter vom Typ `K` übergeben bekommen (Wie in Teilaufgabe a).
- Sei `a = x.m(p1,...,pn)` ein Methodenaufruf einer dynamisch gebundenen Methode. Ersetze `x.m(p1,...,pn)` durch einen Methodenaufruf, z.B. `dispatchm(x,p1,...,pn)`. `dispatchm` kann irgendwo z.B. als `static`-Methode implementiert werden und hat denselben Rückgabetyp wie `m`.
- Die Implementierung von `dispatchm` entscheidet anhand des Typen des ersten übergebenen Parameters mit if-then-else und instanceof, welche ersetzende Methode aufgerufen werden muss, ruft diese auf und liefert deren Ergebnis zurück. Dazu muss dann das als impliziter Parameter zu übergebende `x` entsprechend gecastet werden.

Der Hauptvorteil der gerade beschriebenen Technik liegt darin, dass die dynamische Bindung innerhalb einer einzigen Methode implementiert wird. Erweitert man ein Programm z.B. um neue Typen, so muss man in der Regel nur diese Methode anpassen und nicht den gesamten Code nach evtl. zu ändernden Stellen absuchen.

Eine Anwendung der zuletzt beschriebenen Technik auf das gegebene Beispiel führt zu folgendem Ergebnis:

```
class Dynamic {
    static String m(Dynamic THIS) {
        return "m in Dynamic";
    }

    public static void main(String[] s) {
        Dynamic[] d = new Dynamic[4];
        d[0] = new Dynamic();
        d[1] = new S1();
        d[2] = new S2();
        d[3] = new S3();
        for(int i=0;i<d.length;i++) System.out.println(mdispatch(d[i]));
    }

    static String mdispatch(Dynamic d) {
        if(d instanceof S3) return S3.m((S3)d);
        else if (d instanceof S1) return S1.m((S1)d);
        else if (d instanceof S2) return S2.m((S2)d);
        return Dynamic.m(d);
    }
```



```
}

class S1 extends Dynamic  {
    static String m(S1 THIS) {
       return "m in S1";
    }
}

class S2 extends Dynamic  {
    static String m(S2 THIS) {
       return "m in S2";
    }
}

class S3 extends S2 {
    static String m(S3 THIS) {
       return "m in S3";
    }
}
```

Aufgabe 3

a) Die `start`-Methode eines Threads hat eine besondere Semantik. Ruft ein Thread *T1* die `start`-Methode eines Threads *T2* auf, so kehrt *T1* sofort vom Aufruf der `start`-Methode zurück. Der Thread *T2* wird gestartet und läuft nebenläufig zum aufrufenden Thread. *T2* führt die `run`-Methode seines `Runnable`-Objektes aus. Die `run`-Methode ist eine gewöhnliche Java-Methode, die die Arbeit des Thread erledigt. Ist die Ausführung der `run`-Methode beendet, so ist auch die Ausführung von *T2* beendet.

b) Die folgende ThreadPool-Implementierung leistet das Gewünschte:

```
import java.util.Vector;

class ThreadPool {
    Vector jobs = new Vector();
    private int working_threads=0;
    private int max_threads;
    private int started_threads = 0;

    public  ThreadPool(int m) {
       max_threads = m;
    }

    synchronized public void doJob(Runnable r) {
       if(started_threads<max_threads) {
          Worker t = new Worker();
          t.setDaemon(true);
          t.start();
          started_threads++;
       }
```

```
        jobs.add(r);
        notifyAll();
    }

    synchronized public void  waitUntilComplete() {
        try {
            while(jobs.size() > 0 || working_threads>0) wait();
        } catch (InterruptedException ie) {
            // spielt hier keine Rolle
        }
    }

    private class Worker extends Thread {
        public void run() {
            while(true) {
                Runnable job;
                synchronized(ThreadPool.this) {
                    try {
                        while(jobs.size() == 0) ThreadPool.this.wait();
                    } catch (InterruptedException e) {
                    // spielt hier keine Rolle
                    }
                    job = (Runnable)jobs.firstElement();
                    jobs.remove(job);
                    working_threads++;
                }
                job.run();
                synchronized(ThreadPool.this) {
                    working_threads--;
                    ThreadPool.this.notifyAll();
                }
            }
        }
    }
}
```

Jeder Zugriff auf Attribute eines `ThreadPool`-Objektes geschieht nur durch `synchronized` Methoden oder aus `synchronized` Blöcken heraus, wobei auf dem ThreadPool-Objekt synchronisiert wird, um Inkonsistenzen durch den gemeinsamen Gebrauch von Ressourcen zu verhindern. Die `Worker`-Klasse ist daher als innere Klasse vereinbart, da so alle `Worker`-Objekte auf das zugehörige ThreadPool-Objekt (mit `ThreadPool.this`) und auf dessen Attribute zugreifen können. Die arbeitenden Threads werden im Attribut `working_threads` gezählt, da bei der `waitUntilComplete`-Methode nicht nur überprüft werden muss, ob keine Jobs mehr auf Bearbeitung warten, sondern ob auch alle bereits begonnenen Jobs zu Ende berechnet wurden.

c) Die unten angegebene Quicksort-Implementierung benutzt einen ThreadPool, um Quicksort zu parallelisieren. Folgende Änderungen wurden dabei vorgenommen:

- Jedes `Quicksort`-Objekt hat eine Referenz auf ein `ThreadPool`-Objekt, das dem Konstruktor der Klasse Quicksort zusätzlich übergeben wird.
- Die Klasse Quicksort implementiert das Runnable Interface, damit die run-Methode von einem Thread ausgeführt werden kann.
- Anstatt der run-Methode ruft der Konstruktor zum Start eines Sortierdurchganges die `doJob`-Methode auf und übergibt damit die Sortieraufgabe dem ThreadPool.
- In der `main`-Methode muss nach dem ersten Sortieraufruf so lange gewartet werden,

bis alle evtl. gestarteten Sortieraufträge erledigt sind. Dies geschieht mit der Methode `waitUntilComplete`.

```
class Quicksort implements Runnable {
    int a[],l,r;
    ThreadPool tp;

    public static void main(String[] sv) {
        ThreadPool tp = new ThreadPool(5);
        int f[] = { 1,5,3,4,5,6,5,5,4,3,6,5,3,6,7,7,5,4,3 };
        for(int i=0;i<f.length;i++) System.out.println(f[i]);
        new Quicksort(f,0,f.length-1,tp); System.out.println("");
        tp.waitUntilComplete();

        for(int i=0;i<f.length;i++) System.out.println(f[i]);
    }

    Quicksort(int a[], int l, int r, ThreadPool tp) {
        this.a=a;
        this.l=l;
        this.r=r;
        this.tp = tp;
        tp.doJob(this);
    }

    public void run() {
        int i,j,v;
        if(r>l) {
            v = a[r];
            i = l-1;
            j = r;
            while(true) {
                while(a[++i] < v);
                while(--j>=0 && a[j] > v);
                if(i>=j) break;
                int h=a[i];a[i]=a[j];a[j]=h;
            }
            int h=a[i];a[i]=a[r];a[r]=h;
            new Quicksort(a,l,i-1,tp);
            new Quicksort(a,i+1,r,tp);
        }
    }
}
```