- Ähnliches gilt auch für den (seltenen) Fall, dass ein Attribut überschrieben wurde
  - bei überschriebenen Attributen wird nicht erst zur Laufzeit entschieden, auf welches zugegriffen wird
    - das wäre zu zeitaufwändig
  - bei ihnen findet stattdessen „static binding“ statt
  - entscheidend dabei ist der Datentyp der Referenz
    - man sagt, dass „die Referenz darüber entscheidet, auf welches Attribut zugegriffen wird“
    - nicht - wie bei Methoden - das Objekt!
- wir betrachten dazu noch einmal unser früheres Beispiel:


```
public class TestA                  // völlig untypische Klasse !
{
  public final int a = -1;
  public final double x = 3.14;
}
```

```
public class TestB extends TestA    // völlig untypische Klasse !
{
  public String a = "Hallo";                  // Namenskonflikt !
}
```

die Anweisungen

```
TestB testObjekt = new TestB();
TestB bReferenz = testObjekt;               // dasselbe Objekt!
TestA aReferenz = testObjekt;               // dasselbe Objekt!

System.out.println(bReferenz.a);
System.out.println(aReferenz.a);
```

führen zur Ausgabe:

```
Hallo
-1
```


- an Hand unserer Tierklassenhierarchie können wir noch ein weiteres wichtiges Konzept objektorientierter Programmierung erklären
  - die Basisklasse `Tier` der Hierarchie ist so allgemein, dass es uns schwer fiel, die Methode `sprich` sinnvoll zu implementieren
- dies ist typisch für die Wurzel(klasse) gut gestalteter Klassenhierarchien
  - es ist ja gerade ein Ziel objektorientierter Programmierung, das Allgemeine herauszuarbeiten, um es durch Vererbung wiederzuverwenden

- wir haben das Problem dadurch gelöst, dass wir den Rumpf von `sprich` leer gelassen haben

```
public void sprich()
{
}
```

- was aber würden wir machen, wenn eine Methode in ähnlicher Situation aus inhaltlichen Gründen einen Wert zurückgeben müsste, der erst in abgeleiteten Klassen sinnvoll ermittelt werden kann?
  - irgendeinen Wert als „Dummy“ zurückgeben?

- die Methode aus der Wurzelklasse entfernen und erst in den abgeleiteten Klassen implementieren?

- beide Alternativen sind schlecht
  - bei Rückgabe eines Dummies könnten Objekte der Klasse erzeugt werden, die sich nicht „vernünftig“ verhalten
  - würde die Methode aus der Wurzelklasse entfernt werden, würde der Vorteil des Polymorphismus entfallen
- für derartige Situationen gibt es in Java abstrakte Methoden und abstrakte Klassen

- wird eine Methode als **`abstract`** deklariert, braucht sie nicht implementiert zu werden
  - und zwar unabhängig davon, ob sie einen Rückgabewert hat oder nicht
  - statt des Methodenrumpfes ist lediglich ein Semikolon zu schreiben

```
public abstract void machWas();

public abstract double liefereWas(double input);
```

- dadurch wird die Klasse, die die Methode enthält, zu einer abstrakten Klasse
  - sie muss entsprechend gekennzeichnet werden

- zur Verdeutlichung machen wir unsere Klasse **`Tier`** zu einer abstrakten Klasse:

```
public abstract class Tier
{
  private String art;

  public Tier(String dieArt)
  {
    art = dieArt;
  }

  public abstract void sprich();

  public String liefereInfo()
  {
    return art + ". ";
  }
}
```

  - **`abstract`** ist also sowohl ein Methoden- als auch ein Klassenmodifizierer

- wenn eine Klasse **`abstract`** ist, hat das folgende Auswirkungen
  - es können keine Objekte von ihr erzeugt werden
    - sie kann nur als Basisklasse zur Vererbung herangezogen werden
  - jede Klasse, die von ihr abgeleitet wird, muss ihre als **`abstract`** deklarierten Methoden implementieren
    - ansonsten ist sie selbst **`abstract`**
- dadurch ist gewährleistet, dass Polymorphismus uneingeschränkt möglich ist

- weil wir unsere Klasse **Tier** zu einer abstrakten Klasse gemacht haben, müssen wir die Methode **sprich** in der Klasse **Ameise** implementieren

```
public class Ameise extends Tier
{
  public Ameise()
  {
    super("Ameise");
  }

  public void sprich()
  {
  }

  public String liefereInfo()
  {
    return "Ich bin eine " + super.liefereInfo();
  }
}
```


- die Klasse **LieblingsTier** deklarieren wir auch als **abstract**, da wir die Methode **sprich** für sie nicht sinnvoll implementieren können

```
public abstract class LieblingsTier extends Tier
{
  private String name;

  public LieblingsTier(String dieArt, String derName)
  {
    super(dieArt);
    name = derName;
  }

  public String liefereInfo()
  {
    return super.liefereInfo() + "Ich heisse " + name +". ";
  }
}
```

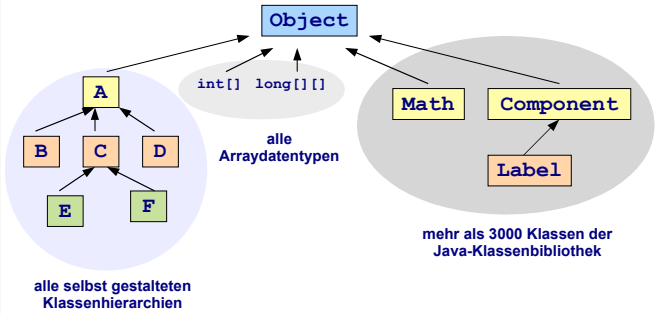

- von der Möglichkeit, Klassen als **abstract** zu deklarieren, wird häufig Gebrauch gemacht
  - abstrakte Klassen definieren einen Datentyp, der durch Vererbung polymorph ausgestaltet werden kann
  - die Wurzel gut gestalteter Klassenhierarchien ist in der Regel **abstract**
  - dementsprechend sind etliche Klassen der Java-Klassenbibliothek **abstract**
    - zum Beispiel **Number**, **Reader**, **Writer**, **Calendar**, **Component**

- beim Einsatz abstrakter Klassen sind noch einige Details zu beachten
  - eine Klasse kann nicht gleichzeitig **final** und **abstract** sein
  - eine Methode, die **private** ist, kann nicht **abstract** sein
  - eine Methode, die **abstract** ist, kann nicht gleichzeitig **static**, **final**, **synchronized**, **native** oder **strictfp** sein
  - Methoden, die **abstract** sind, können eine **throws**-Klausel haben

- alle Klassen in Java, die nicht explizit mittels **`extends`** von einer Klasse abgeleitet wurden, werden automatisch von der Klasse **`Object`** abgeleitet, die sich im Package **`java.lang`** befindet
  - in Java ist also jede von **`Object`** verschiedene Klasse eine abgeleitete Klasse!
  - es ist zulässig, aber überflüssig und unüblich, dies bei der Deklaration einer Klasse deutlich zu machen

```
public class A extends Object
{
  // Deklarationen
}
```


- **`Object`** ist die Wurzel der gesamten Java-Klassenhierarchie



- in anderen objektorientierten Sprachen gibt es keine gemeinsame Wurzel für alle Klassen
- weil in Java alle Klassen von **`Object`** erben, ist es wichtig, zu wissen, welche Eigenschaften **`Object`** hat
  - **`Object`** hat keine Attribute
  - **`Object`** hat (zur Zeit) 11 Methoden, die alle Java-Klassen erben
    - die Methoden **`notify`**, **`notifyAll`** und drei Varianten von **`wait`** sind erst für die Programmierung nebenläufiger Prozesse mit Hilfe von Threads wichtig
    - die sechs restlichen Methoden dienen unterschiedlichen Zwecken

`public String toString()`

- liefert eine Darstellung des Objektes in Textform
- diese Darstellung wird von den Methoden **`print`** und **`println`** benutzt, wenn sie das Objekt schreiben
- bei der in **`Object`** implementierten Version der Methode hat die Darstellung eines Objektes folgende Form

  `Klassenbezeichner@Hexadezimalzahl`

- der Klassenbezeichner ist in der Regel der Name der Klasse
  - bei Arrays hat er jedoch eine spezielle Gestalt

- die Hexadezimalzahl ist der sogenannte Hashcode des Objektes

diese Anweisungen

```
int[] testArray = {1, 2, 3, 4};
System.out.println("int-Array :  " + testArray);
Hund testTier = new Hund("Lumpi", "Dackel");
System.out.println("Hund :       " + testTier);
Tier[] viecher = {testTier, new Ameise()};
System.out.println("Tier-Array : " + viecher);
```

führen zu dieser Ausgabe:

```
int-Array :  [I@19f953d
Hund :       Hund@e48e1b
Tier-Array : [LTier;@1ad086a
```

- **`toString`** sollte in allen selbst gestalteten Klassen sinnvoll überschrieben werden
  - dann können deren Objekte genauso einfach geschrieben werden wie Werte primitiver Datentypen


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  public String toString()
  {
    return tag + ". " + monat + ". " + jahr;
  }
}
```


diese Anweisungen

```
MeinDatum datum = new MeinDatum();
System.out.println("Heute ist der " + datum);
```

führen zu dieser Ausgabe:

```
Heute ist der 21. 3. 2008
```


`protected void finalize() throws Throwable`

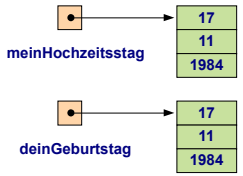
- diese Methode wird vom Garbage Collector aktiviert bevor er ein Objekt „zerstört“
- in **`Object`** enthält der Rumpf von **`finalize`** keine Anweisungen
- Klassen, deren Objekte auf externe Ressourcen zugreifen (z.B. Netzwerkverbindungen, Datenbanken, Drucker, etc.), sollten **`finalize`** so überschreiben, dass die belegten Ressourcen frei gegeben werden, bevor ihre Objekte zerstört werden
  - **Hinweis:** **`finalize`** kann benutzt werden, um die Aktivitäten des Garbage Collectors zu protokollieren

`public final Class getClass()`

- liefert das sogenannte Laufzeitobjekt der Klasse, zu der das Objekt gehört
  - wenn `x` ein Objekt der Klasse `A` ist, liefert `x.getClass()` das Laufzeitobjekt von `A`
- es handelt sich um ein Objekt vom Typ `Class`, das von der JVM erzeugt wird und zahlreiche Informationen über die jeweilige Klasse zur Verfügung stellt, die zur Laufzeit ausgewertet werden können
  - `Class` hat dazu mehr als 50 Methoden
- das Auswerten von Klasseninformationen zur Laufzeit wird als reflection bezeichnet
  - damit beschäftigen wir uns zu gegebener Zeit

`public boolean equals(Object einObjekt)`

- liefert `true`, wenn das aktuelle Objekt und `einObjekt` „gleich“ sind, ansonsten `false`
- falls der Input `null` ist, liefert `equals` ebenfalls `false`
- in `Object` liefert `equals` einfach den Wert des Ausdrucks `(this == einObjekt)`
- wenn dieser (sehr strenge) Begriff von Gleichheit (nämlich Identität) für eine Klasse nicht angemessen ist, sollte `equals` überschrieben werden
  - wir illustrieren dies anhand unserer Klasse `MeinDatum`
  - zwei Datumsobjekte sollten bereits dann gleich sein, wenn der Tag, der Monat und das Jahr übereinstimmen



zwei Objekte, die gleich sein sollten, obwohl sie nicht identisch sind

- daher überschreiben wir in unserer Klasse `MeinDatum` die Methode `equals`
- außerdem fügen wir ihr einen weiteren Konstruktor hinzu


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  public MeinDatum(int derTag, int derMonat, int dasJahr)
  {
    tag = derTag;
    monat = derMonat;
    jahr = dasJahr;
  }
```

```
public String toString()
{
  return tag + ". " + monat + ". " + jahr;
}

public boolean equals(Object einObjekt)
{
  if (this == einObjekt)                    // Objekte identisch?
    return true;
  else
    if (einObjekt == null)                  // Input kein Objekt?
      return false;
    else
      if (getClass() != einObjekt.getClass())     // Typ ungleich?
        return false;
      else
      {
        MeinDatum datum = (MeinDatum) einObjekt; // Input korrekt!
        return (tag == datum.tag) &&
               (monat == datum.monat) &&
               (jahr == datum.jahr);
      }
}
}
```


diese Anweisungen

```
MeinDatum meinHochzeitstag = new MeinDatum(17, 11, 1984);
MeinDatum deinGeburtstag = new MeinDatum(17, 11, 1984);
System.out.print("Objekte identisch?  ---->  ");
System.out.println(meinHochzeitstag == deinGeburtstag);
System.out.print("Objekte gleich?     ---->  ");
System.out.println(meinHochzeitstag.equals(deinGeburtstag));
```

führen zu dieser Ausgabe:

```
Objekte identisch?  ---->  false
Objekte gleich?     ---->  true
```

- **Achtung:** das sinnvolle Überschreiben von `equals` ist gelegentlich schwieriger, als es zunächst scheint!

`public int hashCode()`

- liefert den Hashcode des Objektes, d.h. eine ganze Zahl, die folgenden Bedingungen genügt
  - wenn `x.equals(y)` den Wert `true` hat, dann muss `x.hashCode() == y.hashCode()` sein
  - solange sich im Programmverlauf nichts am Zustand von `x` ändert, was einen signifikanten Einfluss auf die Methode `equals` hat, ist `x.hashCode()` immer dieselbe Zahl
  - wenn `x` und `y` verschieden sind, ist es sehr wahrscheinlich, dass auch `x.hashCode()` und `y.hashCode()` verschieden sind
- der Hashcode kann in sogenannten Hashtabellen zum schnellen Auffinden von Daten benutzt werden

- in `Object` wird der von `hashCode` gelieferte Wert aus der Adresse des Objektes auf dem Heap berechnet
  - er ist daher bei jedem Programmablauf anders

- **Achtung:** durch die oben genannten Bedingungen sind die Methoden `equals` und `hashCode` miteinander verknüpft
  - wenn `equals` überschrieben wird, muss stets auch `hashCode` so überschrieben werden, dass die Bedingungen erfüllt sind
    - wir müssen daher in unserer Klasse `MeinDatum` auch die Methode `hashCode` überschreiben

```
public int hashCode()
{
  return (new Long(tag + 1000 * monat + 100000 * jahr)).hashCode();
}
```

diese Anweisungen

```
MeinDatum meinHochzeitstag = new MeinDatum(17, 11, 1984);
MeinDatum deinGeburtstag = new MeinDatum(17, 11, 1984);
MeinDatum heute = new MeinDatum();
System.out.print("Hashcode von 'meinHochzeitstag'   --->  ");
System.out.println(meinHochzeitstag.hashCode());
System.out.print("Hashcode von 'deinGeburtstag'     --->  ");
System.out.println(deinGeburtstag.hashCode());
System.out.print("Hashcode von 'heute'              --->  ");
System.out.println(heute.hashCode());
```

führen zu dieser Ausgabe:

```
Hashcode von 'meinHochzeitstag'   --->  198411017
Hashcode von 'deinGeburtstag'     --->  198411017
Hashcode von 'heute'              --->  200803022
```


`protected Object clone() throws CloneNotSupportedException`

- liefert eine „Kopie“ des Objektes, d.h. eine Referenz auf ein zweites Objekt, das zum Zeitpunkt des Kopierens exakt denselben Zustand hat, dessen Zustand sich aber im Verlauf der Zeit unterscheiden kann vom Zustand des kopierten Objektes
- die Kopie sollte folgende Eigenschaften haben:
  - die Referenzen `x` und `x.clone()` sollen stets verschieden sein, d.h. es soll `x != x.clone()` sein
  - die Kopie soll denselben Datentyp haben wie das Original, d.h. es soll stets `x.getClass() == x.clone().getClass()` sein

  - das Original und die Kopie sollen gleich sein, d.h. unmittelbar nach dem Erzeugen der Kopie soll `x.clone().equals(x)` den Wert `true` haben
- in `Object` ist `clone` so implementiert, dass einfach ein neues Objekt erzeugt wird, dessen Attribute Kopien der Attribute des Originals sind
  - die Methode ist nicht in Java geschrieben, sondern `native`
- es wird also eine flache Kopie des Objektes erzeugt
- wie wir bereits bei Arrays gesehen haben, ist dieses Verhalten unproblematisch, wenn die Attribute des Objektes einen primitiven Datentyp haben oder ihr Datentyp eine nicht veränderbare Klasse ist

- bei komplexen Klassen, die Attribute haben, deren Datentyp eine veränderbare Klasse ist, sollte `clone` besser eine tiefe Kopie des Objektes liefern, d.h. `clone` sollte überschrieben werden
- `clone` sinnvoll zu überschreiben, erfordert eine Reihe von Überlegungen
  - daher wird man das nur für Klassen tun, von denen man annimmt, dass tatsächlich tiefe Kopien von Objekten gebraucht werden
  - die meisten Klassen der Java-Klassenbibliothek überschreiben `clone` nicht!
- wenn `clone` nicht überschrieben wird, steht die Methode auch nicht zur Verfügung
  - `clone` ist `protected`, nicht `public` !

diese Anweisungen

```
MeinDatum deinGeburtstag = new MeinDatum(17, 11, 1984);
MeinDatum meinHochzeitstag = (MeinDatum) deinGeburtstag.clone();
```

führen zu dieser Fehlermeldung:

```
clone() has protected access in java.lang.Object

MeinDatum meinHochzeitstag = (MeinDatum) deinGeburtstag.clone();
                                                        ^
1 error
```

- **Achtung:** bei allen Arraydatentypen ist `clone` aber automatisch überschrieben, und zwar so, dass
  - `clone public` ist
  - `clone` eine flache Kopie des Arrays erzeugt

- für Arrays, bei denen das Erstellen einer flachen Kopie adäquat ist, kann daher statt `arraycopy` auch `clone` benutzt werden

diese Anweisungen

```
int[] x = new int[10];
initialisiereArray(x, 1, 8, "<=");
int[] y = x.clone();
System.out.println("Arrays identisch?    ----->  " + (x == y));
System.out.print("Arrays gleich?       ----->  ");
System.out.println(x.equals(y));
System.out.print("Array 'x'  --->  ");
schreibeArray(x, " ");
System.out.println();
System.out.print("Array 'y'  --->  ");
schreibeArray(y, " ");
```

führen zu dieser Ausgabe:

```
Arrays identisch?    ----->  false
Arrays gleich?       ----->  false
Array 'x'  --->  0 1 2 3 4 5 6 7 8 0
Array 'y'  --->  0 1 2 3 4 5 6 7 8 0
```


- für unsere Klasse `MeinDatum` reicht das Standard-verhalten von `clone` aus
  - daher überschreiben wir `clone` wie folgt

```
public Object clone()
{
  try
  {
    return super.clone();
  }
  catch (CloneNotSupportedException cnse)
  {
    return null;
  }
}
```

- weil der Zugriffsschutz von `protected` zu `public` verringert wurde, steht die Methode jetzt allgemein zur Verfügung

- sie leistet aber nicht das Gewünschte

diese Anweisungen

```
MeinDatum deinGeburtstag = new MeinDatum(17, 11, 1984);
MeinDatum meinHochzeitstag = (MeinDatum) deinGeburtstag.clone();
System.out.println("Dein Geburtstag:    " + deinGeburtstag);
System.out.println("Mein Hochzeitstag:  " + meinHochzeitstag);
```

führen zu dieser Ausgabe:

```
Dein Geburtstag:    17. 11. 1984
Mein Hochzeitstag:  null
```

- die Ursache für das Fehlverhalten ist die Implementierung von `clone` in `Object`
- `clone` überprüft immer zuerst, ob die Klasse, zu der das aktuelle Objekt gehört, das Interface `Cloneable` implementiert hat

- falls das nicht der Fall ist, erzeugt **clone** eine **CloneNotSupportedException**, die von unserer Methode gefangen wird und zu obiger Ausgabe führt
- wir müssen also in unserer Klasse **MeinDatum** noch das Interface **Cloneable** implementieren
  - weil es sich um ein leeres „Markerinterface“ handelt, ist das sehr einfach
  - damit signalisieren wir allen Anwendern, das unsere Klasse **MeinDatum** das Kopieren mittels **clone** unterstützt
- da wir **MeinDatum** noch einige weitere Methoden (zum Testen) hinzufügen, hat die Klasse jetzt folgende Gestalt


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum implements Cloneable      // Interface nötig!
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  public MeinDatum(int derTag, int derMonat, int dasJahr)
  {
    tag = derTag;
    monat = derMonat;
    jahr = dasJahr;
  }
```



```
public void setzeTag(int derTag)
{
  tag = derTag;
}

public void setzeMonat(int derMonat)
{
  monat = derMonat;
}

public void setzeJahr(int dasJahr)
{
  jahr = dasJahr;
}

public String toString()
{
  return tag + ". " + monat + ". " + jahr;
}

public int hashCode()
{
  return (new Long(tag + 1000 * monat + 100000 * jahr)).hashCode();
}
```



```
  public boolean equals(Object einObjekt)
  {
    if (this == einObjekt)                        // Objekte
identisch?
      return true;
    else
      if (einObjekt == null)                      // Input kein
Objekt?
        return false;
      else
        if (getClass() != einObjekt.getClass())     // Typ
ungleich?
          return false;
        else
        {
          MeinDatum datum = (MeinDatum) einObjekt; // Input
korrekt!
          return (tag == datum.tag) &&
                 (monat == datum.monat) &&
                 (jahr == datum.jahr);
        }
  }
```

```
  public Object clone()
  {
    try
    {
      return super.clone();
    }
    catch (CloneNotSupportedException cnse)
    {
      return null;
    }
  }
}
```

- jetzt verhalten sich die Methoden vernünftig:

diese Anweisungen

```
MeinDatum deinGeburtstag = new MeinDatum(17, 11, 1984);
MeinDatum meinHochzeitstag = (MeinDatum) deinGeburtstag.clone();
System.out.println("Dein Geburtstag:   " + deinGeburtstag);
System.out.println("Mein Hochzeitstag:  " + meinHochzeitstag);
System.out.println("\nMein Hochzeitstag wird korrigiert.");
meinHochzeitstag.setzeJahr(2000);
System.out.println("Mein Hochzeitstag:  " + meinHochzeitstag);
System.out.println("\nDas hat keine Auswirkung auf Dein
                    Geburtstag.");
System.out.println("Dein Geburtstag:   " + deinGeburtstag);
```

führen zu dieser Ausgabe:

```
Dein Geburtstag:    17. 11. 1984
Mein Hochzeitstag:  17. 11. 1984

Mein Hochzeitstag wird korrigiert.
Mein Hochzeitstag:  17. 11. 2000

Das hat keine Auswirkung auf Dein Geburtstag.
Dein Geburtstag:    17. 11. 1984
```


- eine tiefe Kopie von Objekten kann leicht erzeugt werden, wenn ihre Attribute sinnvoll kopiert werden können
  - gegebenenfalls ist zunächst für tiefe Kopien der Attribute zu sorgen
- wir illustrieren das Erzeugen einer tiefen Kopie für die Klasse **`HaltbarkeitsInfo`**, die ein Attribut vom Typ **`MeinDatum`** hat


```
public class HaltbarkeitsInfo implements Cloneable          // nötig!
{
  private int losNummer;
  private MeinDatum verfallsDatum;

  public HaltbarkeitsInfo(int dieLosNummer,
                          MeinDatum dasVerfallsDatum)
  {
    losNummer = dieLosNummer;
    verfallsDatum = dasVerfallsDatum;
  }

  public void setzeTag(int tag)
  {
    verfallsDatum.setzeTag(tag);
  }

  public void setzeMonat(int monat)
  {
    verfallsDatum.setzeMonat(monat);
  }
```

```
public String toString()
{
  return "Los Nummer " + losNummer +
         ":  haltbar bis  --->  "  +
         verfallsDatum;
}

public Object clone()
{
  try
  {
    HaltbarkeitsInfo info = (HaltbarkeitsInfo) super.clone();
    info.verfallsDatum = (MeinDatum) verfallsDatum.clone();
    return info;
  }
  catch (CloneNotSupportedException cnse)
  {
    return null;
  }
}
}
```

zuerst flache Kopie

dann „richtige“ Kopie des Attributs

diese Anweisungen erzeugen 4 Kopien:

```
HaltbarkeitsInfo[] testInfo = new HaltbarkeitsInfo[4];
HaltbarkeitsInfo initialInfo = new HaltbarkeitsInfo(123456,
                                                    new MeinDatum());
for (int i = 0; i < testInfo.length; i++)
  testInfo[i] = (HaltbarkeitsInfo) initialInfo.clone();
for (HaltbarkeitsInfo info : testInfo)
  System.out.println(info);
testInfo[2].setzeTag(1);              // ändert eine der Kopien
testInfo[2].setzeMonat(11);           // ändert eine der Kopien
System.out.println();
for (HaltbarkeitsInfo info : testInfo)
  System.out.println(info);
```

```
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008

Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
Los Nummer 123456:  haltbar bis  --->  1. 11. 2008
Los Nummer 123456:  haltbar bis  --->  22. 3. 2008
```

die Kopien sind voneinander unabhängig!

- weil in Java alle Klassen von `Object` abgeleitet sind, können Arrays mit Komponentendatentyp `Object` alle Arten von Objekten aufnehmen
- dadurch ergibt sich eine erste Möglichkeit, allgemein gültige Methoden zu schreiben, d. h. Methoden, die für möglichst viele Datentypen anwendbar sind
  - das Schreiben allgemeingültigen Codes wird generische Programmierung genannt
  - seit der Version J2SE 5.0 unterstützt Java die generische Programmierung noch auf andere Weise
    - dazu später mehr

- als Beispiel betrachten wir folgende Methode, die in einem beliebigen Array nach beliebigen Objekten sucht
  - wenn das gesuchte Objekt im Array vorhanden ist, liefert die Methode den Index der ersten Komponente (von links), die das Objekt enthält, ansonsten -1

```
public int sucheInArray(Object[] array, Object einObjekt)
{
  int fundStelle = -1;
  boolean gefunden = false;
  for (int i = 0; !gefunden && i < array.length; i++)
    if (array[i].equals(einObjekt))
    {
      fundStelle = i;
      gefunden = true;
    }
  return fundStelle;
}
```