```
Object[]: [Person: Name='Tom' City='Hamburg' Birthday='Sat Jul 04 13:31:17 CEST
      2009', Person: Name='Jerry' City='Kiel' Birthday='Sat Jul 04 13:31:17 CEST
      2009']
```

Die Vorstellung der Methode `toString()` ist damit zunächst abgeschlossen. Im Verlauf dieses Kapitels werde ich eine Verbesserung vornehmen.

4.1.2 Die Methode `equals()`

Dieser Abschnitt stellt die Implementierung der Methode `equals(Object)` ausführlich dar, weil diese Methode eine zentrale Rolle bei der Speicherung von Objekten in Containern spielt. Wie bereits erwähnt, ist jedes Objekt durch seinen ***Zustand*** (Belegung der Attribute), sein ***Verhalten*** (Methoden) und seine ***Identität*** (Referenz) definiert. Zum Vergleich von Objekten gibt es unter anderem folgende Möglichkeiten:

1. **Operator '==' –** Mit dem Operator '==' werden Objektreferenzen verglichen. Somit wird überprüft, ob es sich um *dieselben* Objekte handelt.
2. **Aufruf der Methode `equals()` –** Einen Vergleich mit `equals(Object)` nutzt man, wenn nicht die Identität, sondern der semantische Zustand von Interesse ist. Dazu muss die Methode `equals(Object)` in eigenen Klassen überschrieben werden, um den Objektzustand zu vergleichen. Die Defaultimplementierung der Klasse `Object` vergleicht lediglich Referenzen mit dem Operator '=='.

Die Methode `equals(Object)` dient demnach dazu, zwei Objekte ***semantisch*** zu vergleichen. Sie kann zwar explizit zum Vergleich aufgerufen werden, in der Regel geschieht dies aber implizit und automatisch bei verschiedensten Aktionen.

Die Rolle von `equals()` beim Suchen

Um erste Erkenntnisse zur Implementierung von `equals(Object)` zu gewinnen, betrachten wir ein einfaches Beispiel einer Klasse `Spielkarte`. Die Implementierung nutzt einen Aufzählungstyp `Farbe` und einen `int` als Wert der Karte:

```
public final class Spielkarte
{
    public enum Farbe
    {
        KARO, HERZ, PIK, KREUZ
    }

    private final Farbe farbe;
    private final int   wert;

    public Spielkarte(final Farbe farbe, final int wert)
    {
        this.farbe = farbe;
        this.wert = wert;
    }
}
```

Um ein Kartenspiel zu simulieren, müssen wir Objekte vom Typ `Spielkarte` in einer Datenstruktur speichern. Details zu Datenstrukturen beschreibt Kapitel 5 und geht dabei ausführlich auf das Collections-Framework ein.

Wir nutzen die intuitiv verständliche Datenstruktur `java.util.ArrayList<E>`. Das folgende Listing zeigt das Erzeugen einiger Objekte vom Typ `Spielkarte` und deren Speicherung. Anschließend prüfen wir mit der Methode `contains(Object)` aus dem `java.util.Collection<E>`-Interface, ob die Karte »Pik 8« in der Datenstruktur enthalten ist, und speichern den Rückgabewert in der Variablen `gefunden`:

```
public static void main(final String[] args)
{
    final Collection<Spielkarte> spielkarten = new ArrayList<Spielkarte>();

    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden=" + gefunden);
}
```

Listing 4.4 *Ausführbar als* **'SPIELKARTEEXAMPLE'**

Gefunden oder doch nicht? Auf den ersten Blick lautet die Antwort: Gefunden, natürlich! Denn es wurde eine »Pik 8« beim zweiten Aufruf der Methode `add(Spielkarte)` hinzugefügt. Starten wir das angegebene Ant-Target SPIELKARTEEXAMPLE und prüfen den Wert der Variablen `gefunden`. Wir erleben eine Überraschung, denn `gefunden` hat den Wert `false`. Die gesuchte »Pik 8« ist angeblich nicht in der Datenstruktur. Wie kann das sein?

Die Suche mit `contains(Object)` durchläuft alle gespeicherten Elemente und prüft diese auf Gleichheit mit dem übergebenen Objekt durch Aufruf von `equals(Object)`. Die Klasse `Spielkarte` überschreibt diese Methode jedoch nicht. Es wird daher die Methode der Basisklasse `Object` genommen, die wie folgt implementiert ist:

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

Diese `equals(Object)`-Methode prüft auf Referenzgleichheit. Somit erklärt sich das Ergebnis der gezeigten Suche. Dazu wird die Methode `contains(Object)` mit einem neu erzeugten Objekt aufgerufen. Dieses ist (natürlich) nicht in der Datenstruktur vorhanden.

In vielen Fällen benötigt man aber eine Prüfung auf inhaltliche Gleichheit. Dazu muss die Methode `equals(Object)` überschrieben und deren Kontrakt eingehalten werden.

Der equals()-Kontrakt

In der JLS [28] ist die Methode equals(Object) durch folgende Signatur definiert:

```
public boolean equals(Object obj)
```

Sie muss eine Äquivalenzrelation mit folgenden Eigenschaften realisieren:

- **Null-Akzeptanz** – Für jede Referenz x ungleich null liefert x.equals(null) den Wert false.
- **Reflexivität** – Für jede Referenz x, die nicht null ist, muss x.equals(x) den Wert true liefern.
- **Symmetrie** – Für alle Referenzen x und y darf x.equals(y) nur den Wert true geben, wenn y.equals(x) dies auch tut.
- **Transitivität** – Für alle Referenzen x, y und z gilt: Wenn sowohl x.equals(y) und y.equals(z) den Wert true ergeben, dann muss x.equals(z) auch true liefern.
- **Konsistenz** – Für alle Referenzen x und y, die nicht null sind, müssen mehrmalige Aufrufe von x.equals(y) konsistent immer wieder den Wert true bzw. false liefern.

Mögen diese Forderungen ein wenig kompliziert klingen, so ist eine Umsetzung in der Praxis doch relativ einfach nach folgendem zweistufigen Vorgehen möglich:

1. **Prüfungen** – Zunächst stellen wir mit drei Prüfungen sicher, dass
   a) keine null-Referenzen,
   b) keine identischen Objekte und
   c) nur Objekte des gewünschten Typs verglichen werden.
2. **Objektvergleich** – Anschließend werden die korrespondierenden Attributwerte verglichen.

Prüfungen Die Methode equals(Object) beginnt immer mit einer Prüfung des Übergabeparameters auf null (Punkt a). Für eine bessere Performance wird danach auf Identität (Punkt b) geprüft, weil man sich bei Identität alle weiteren, gegebenenfalls aufwendigen Prüfungen sparen kann. Um nicht Äpfel mit Birnen zu vergleichen, erfolgt eine Typprüfung (Punkt c) vor dem eigentlichen Vergleich der Attribute.

Zu dieser Typprüfung findet man kontroverse Meinungen, ob diese mit getClass() oder instanceof erfolgen sollte. Eine Prüfung mit instanceof ist zwar in vielen Fällen ausreichend, beim Einsatz von Vererbung kann es jedoch zu Fehlern kommen. Wenn man sich unsicher ist, sollte man bevorzugt getClass() verwenden, da ansonsten schnell die geforderte Symmetrie und Transitivität verletzt wird. Darauf gehe ich später genauer ein.

Für die finale, nicht abgeleitete Klasse `Spielkarte` fällt die Wahl leicht: Wir verwenden hier `getClass()`. Aus den Punkten a) bis c) ergeben sich folgende, erste Zeilen der Realisierung von `equals()`:

```
@Override
public boolean equals(Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;

    if (this == other)                          // Reflexivität
        return true;

    if (this.getClass() != other.getClass())    // Typgleichheit sicherstellen
        return false;
```

Tipp: `equals()` mit `instanceof` vs. `getClass()`

Wie erwähnt, findet man zur Typprüfung in `equals(Object)` kontroverse Meinungen. Eine Diskussion mit Joshua Bloch finden Sie unter `http://www.artima.com/intv/bloch17.html`.

Wirkung von `instanceof` bzw. `getClass()` Ein Vergleich mit `getClass()` prüft, ob zwei Klassen exakt den gleichen Typ besitzen. Über `instanceof` werden auch alle Subklassen beim Vergleich auf eine Basisklasse akzeptiert. Es wird also eine Subtypbeziehung überprüft: Man testet damit, ob von der angegebenen Klasse abgeleitet bzw. das Interface implementiert wurde.

Wissenswertes zu `instanceof` Der Einsatz von `instanceof` ist nur erlaubt, wenn die geprüften Klassen eine gemeinsame Typhierarchie, d. h. eine gemeinsame Basisklasse ungleich `Object`, besitzen. Würde man eine beliebige Referenzvariable vom Typ `JButton` auf Typkonformität mit der zuvor vorgestellten Klasse `Person` prüfen wollen, so führt die Anweisung `okButton instanceof Person` zu einem Kompilierfehler. ***Unterschiedliche Vererbungshierarchien lassen sich mit `instanceof` nicht vergleichen.***

Objektvergleich Wurden die zuvor beschriebenen Prüfungen bestanden, so kann nun das übergebene Objekt sicher auf den entsprechenden Typ gecastet werden. Die einzelnen Attribute des Objekts werden jetzt (vorzugsweise in der Reihenfolge ihrer Definition) auf Gleichheit geprüft. Dabei gelten folgende Regeln:

1. Vergleiche alle primitiven Ganzzahltypen per Operator '==’. Für Gleitkommazahlen sind derartige Vergleiche fehleranfällig und fragil. Aufgrund der systemimmanenten Ungenauigkeit bei Berechnungen mit Gleitkommazahlen müssen wir bei deren Vergleich besondere Vorsicht walten lassen bzw. sie in `equals(Object)` möglichst vermeiden. Nur in Ausnahmefällen kann man eine spezielle Gleichheitsprüfung vornehmen. Beachten Sie dazu bitte die Ausführungen in dem folgenden Hinweis.

2. Vergleiche alle Objekttypen mit deren `equals(Object)`-Methode. Für optionale Attribute, bei denen `null` ein gültiger Wert ist, muss ein spezielle Behandlung erfolgen. In den folgenden Abschnitten wird dies genauer betrachtet und eine Hilfsmethode `nullSafeEquals(Object, Object)` entwickelt.

Die gewonnenen Erkenntnisse nutzen wir, um die `equals()`-Methode für die Klasse `Spielkarte` zu vervollständigen:

```
@Override
public boolean equals(Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;

    if (this == other)                          // Reflexivität
        return true;

    if (this.getClass() != other.getClass())  // Typgleichheit sicherstellen
        return false;

    // Enum mit equals, int mit Wertevergleich
    final Spielkarte karte = (Spielkarte) other;
    return this.farbe.equals(karte.farbe) && this.wert == karte.wert;
}
```

Hinweis: Ungenauigkeiten der Gleitkommatypen `float` und `double`

Selbst einfache Berechnungen mit den Gleitkommatypen `float` und `double` können bereits zu Ungenauigkeiten führen. Folgendes Beispiel einer `for`-Schleife, die zehnmal den Wert `0.1` addiert und anschließend die Summe mit dem Wert `1` vergleicht, macht mögliche Probleme eindrucksvoll deutlich:

```
public static void main(final String[] args)
{
    float sum = 0.0f;
    for (int i = 0; i < 10; i++)
        sum += 0.1;

    System.out.println("1 != " + sum); // 1 != 1.0000001
}
```

Listing 4.5 *Ausführbar als* **'FLOATUNGENAUIGKEIT'**

Zum Vergleich von Gleitkommazahlen kann man sich eine Hilfsmethode `isEqualWithinPrecision()` wie folgt definieren:

```
public static boolean isEqualWithinPrecision(final double value,
                      final double expected, final double epsilon)
{
    return (value > expected - epsilon && value < expected + epsilon);
}
```

Auf diese Weise werden alle Werte für den erwarteten und den zu prüfenden Wert als »gleich« angesehen, falls deren Differenz kleiner als »Epsilon« ist. ***Für `equals()` ist dann jedoch häufig keine sinnvolle Aussage mehr möglich!***

Typische Fehler bei der Implementierung von `equals()`

Bis jetzt scheinen die Umsetzung des Kontrakts und die Realisierung der Methode `equals(Object)` relativ einfach zu sein. Bei der unbedachten Implementierung kann man jedoch schnell Fehler machen. Betrachten wir dies anhand einer ersten, naiven Implementierung für die Klasse `Person`. Es wird hier momentan bewusst auf die bereits kennengelernte Annotation `@Override` verzichtet, um auf ein Problem aufmerksam machen zu können. Eine erste Realisierung könnte wie folgt aussehen:

```
public boolean equals(final Person other)
{
    return this.getName().equals(other.getName());
}
```

Zunächst scheint so weit alles in Ordnung zu sein, doch tatsächlich enthält diese Lösung zwei Fehler. Prüfen wir die Signatur und den Kontrakt:

- **Signatur** ✗ – Es ist wichtig, bei einer eigenen Implementierung die korrekte Signatur mit dem Eingabetyp `Object` zu verwenden. Die gezeigte Methode verwendet stattdessen folgende Signatur: `equals(Person)`.
- **Null-Akzeptanz** ✗ – Für jede Referenz `x` ungleich `null` sollte der Ausdruck `x.equals(null)` den booleschen Wert `false` ergeben. Die gezeigte Realisierung löst bei Übergabe von `null` aber eine `NullPointerException` durch den Aufruf von `getName()` auf einer `null`-Referenz aus.
- **Reflexivität** ✓
- **Symmetrie** ✓
- **Transitivität** ✓
- **Konsistenz** ✓

Auswertung Wir erkennen, dass die Methode zum einen eine falsche Signatur besitzt und zum anderen den Aspekt der Null-Akzeptanz des Kontrakts verletzt. Beide Fehler fallen nicht auf, solange man nur Objekte gleichen Typs explizit über `equals(Object)` vergleicht. Allerdings betreibt man hier ungewollt Overloading statt Overriding. Zu Problemen kommt es erst, wenn Vergleiche mit `null` erfolgen oder `Person`-Objekte in Containerklassen des JDKs gespeichert werden. Die in der Klasse `Person` definierte Methode `equals(Person)` besitzt *nicht* die erwartete Signatur und wird daher von den Containerklassen auch nicht aufgerufen. Stattdessen wird die Methode `equals(Object)` aus der Klasse `Object` genutzt, die einen Referenzvergleich statt des Vergleichs von Attributwerten durchführt.

Neben diesen eher formalen Fehlern enthält diese Implementierung einen inhaltlichen Fehler. Zwei `Person`-Objekte werden bereits bei gleichem Namen als gleich angesehen. Herr Müller aus Kiel ist aber sicherlich nicht Herr Müller aus Hamburg.

Korrekturen Zur Korrektur folgen wir den Forderungen des Kontrakts und definieren die Methode mit der Signatur `equals(Object)`. Eine Realisierung sichert zunächst die Forderungen Null-Akzeptanz, Reflexivität und Typgleichheit ab. Anschließend werden alle diejenigen Attribute verglichen, die notwendig sind, um `Person`-Objekte eindeutig zu unterscheiden. Dadurch ergibt sich folgende den Kontrakt erfüllende Methode:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;

    if (this == other)                          // Reflexivität
        return true;

    if (this.getClass() != other.getClass())    // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return  this.getName().equals(otherPerson.getName()) &&
            this.getCity().equals(otherPerson.getCity()) &&
            this.getBirthday().equals(otherPerson.getBirthday());
}
```

Das ist bereits ein guter Schritt in die richtige Richtung. Wir können eine Verbesserung erzielen, indem wir den Sourcecode zum Prüfen der Attribute in eine typsichere Hilfsmethode `equalsImpl(Person)` auslagern:

```
private boolean equalsImpl(final Person otherPerson)
{
    return  this.getName().equals(otherPerson.getName()) &&
            this.getCity().equals(otherPerson.getCity()) &&
            this.getBirthday().equals(otherPerson.getBirthday());
}
```

Die Auslagerung der Attributprüfungen in die gezeigte Hilfsmethode erlaubt es, die Methode `equals(Object)` in allen eigenen Klassen uniform zu erstellen. Dort werden zunächst die formalen Kriterien abgesichert, um dann jeweils die private, typsichere Hilfsmethode `equalsImpl()` aufzurufen:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;

    if (this == other)                          // Reflexivität
        return true;

    if (this.getClass() != other.getClass())    // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return equalsImpl(otherPerson);
}
```