objective

• study different implementations of `equals()`
• identify and evaluate different techniques
comparison of reference variables

- can mean two things:
  - check for identity of referenced objects
  - check for equality of referenced objects

- identical objects:
  - occupy same memory location
  - are effectively the "same" object

- equal objects:
  - behave in the same way
  - often means: have same state or content

comparison of reference variables

- check for identity via operator ==
  - yields true if
    - operand values both null, or
    - both refer to same object or array

- check for equality via method
  \[\text{boolean equals(Object o)}\]
  - inherited from class Object
    - can be overridden to implement a check for equality
      - default provided by Object.equals(): check for identity!
    - semantics class-specific
      - depends on implementation
example: class-specific semantics of equals()

- class StringBuffer does not override equals()
- class String does

```java
String s = "Hello World!";
String s1 = new String(s);
String s2 = new String(s);
if ( s1.equals(s2) ) ...

StringBuffer sb1 = new StringBuffer(s);
StringBuffer sb2 = new StringBuffer(s);
if ( sb1.equals(sb2) ) ...
```

which classes must override equals()?

- fundamental distinction between
  - value types
  - non-value types (sometimes called entity types)

- value types
  - have content and behavior depends vitally on this content
  - examples: BigDecimal, String, Date, Point, etc.

- entity types
  - behavior not strictly determined by content
    - service-based types
    - handles to an underlying object
  - examples: Thread, FileOutputStream, entity beans in EJB
**which classes must override `equals()`?**

- **entity types**
  - do not override `Object.equals()`
  - comparing content is pointless, is not relevant anyway

- **value types**
  - override `Object.equals()`
  - equality means "equal content"
  - significant difference between identity and equality

**recommendation:**

Classes that represent value types must override `Object.equals()`.

---

**requirements**

- **obvious requirement** (to an implementation of `equals()` for a value type)
  - check for equality of objects

- **additional requirements imposed by JDK**
  - stem from `HashSet` and `HashMap`
  - have certain expectations; do not work properly otherwise

- **defined in the so-called `equals()` contract**
  - see documentation of Java platform libraries
the equals() contract

- **reflexivity**
  - for any reference value \( x \), \( x.equals(x) \) should return true

- **symmetry**
  - for any reference values \( x \) and \( y \), \( x.equals(y) \) should return true if and only if \( y.equals(x) \) returns true

- **transitivity**
  - for any reference values \( x \), \( y \), and \( z \), if \( x.equals(y) \) returns true and \( y.equals(z) \) returns true, then \( x.equals(z) \) should return true

- **consistency**
  - for any reference values \( x \) and \( y \), multiple invocations of \( x.equals(y) \) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified

- **non-nullity**
  - for any non-null reference value \( x \), \( x.equals(null) \) should return false

---

lack of compliance

- **failure to comply to the equals() contract**
  - leads to subtle bugs
    - difficult to track down because they are conceptual problems
    - conceptual problem are not detected by debugging source code
  - not only hash-based collections rely on reflexivity, symmetry, and transitivity
    - everybody who calls equals() does

- **recommendation:**

> Never provide an implementation of equals() that does not comply to the equals() contract.
implementing equals()  

- different techniques for implementing `equals()`  
  - study a number of sample implementations that have been published  

[1] "Program Development in Java" by Barbara Liskov  
[2] "Effective Java" by Joshua Bloch  
[3] "Practical Java" by Peter Haggar  
[4] JDK 1.3 source code  
  (authors: James Gosling, Arthur van Hoff, Alan Liu)
JDK 1.3, package java.util, class Date

```java
public class Date implements java.io.Serializable, Cloneable, Comparable {
    private transient Calendar cal;
    private transient long fastTime;

    private static Calendar staticCal = null;
    // ... lots of static fields ...

    public long getTime() {
        return getTimeImpl();
    }
    private final long getTimeImpl() {
        return (cal == null) ? fastTime : cal.getTimeInMillis();
    }
    // ... lots of methods ...

    public boolean equals(Object obj) {
        return obj instanceof Date && getTime() == ((Date) obj).getTime();
    }
}
```

Josh Bloch, "Effective Java", item 7 and item 8

```java
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;
    // ... lots of methods ...

    public boolean equals(Object o) {
        if (o==this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extensions == extension &&
              pn.exchange == exchange &&
              pn.areaCode == areaCode;
    }
    // ... lots of methods ...
```
class Golfball {
    private String brand;
    private String make;
    private int compression;
    ...
    public String brand() {
        return brand;
    }
    ...
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj!=null && getClass() == obj.getClass()){
            Golfball gb = (Golfball)obj; // Classes are equal, downcast.
            if (brand.equals(gb.brand())) && (make.equals(gb.make())) &&
                compression == gb.compression())
                return true;
        }
        return false;
    }
}

class MyGolfball extends Golfball {
    public final static byte TwoPiece = 0;
    public final static byte ThreePiece = 1;
    private byte ballConstruction;
    ...
    public byte constuction() {
        return ballConstruction;
    }
    ...
    public boolean equals(Object obj) {
        if (super.equals(obj))
        {
            MyGolfball bg = (MyGolfball)obj; // Classes equal, downcast.
            if (ballConstruction == gb.ballConstruction())
                return true;
        }
        return false;
    }
}
different techniques

• signature
  – disagreement:
    • overriding in combination with overloading
      – several versions of `equals()` method (Listing 1)
    • just one signature
      – namely `Object.equals(Object o)`

• comparing fields
  – agreement:
    • `equals()` must compare fields defined in the class
      – if fields contribute to state of object
      – transient and static fields are not considered (see Listing 2)

• delegation to super
  – agreement:
    • `super.equals()` should be invoked
    • if class has a superclass other than `Object`

• type check and downcast
  – agreement:
    • check that other object is of a type
to which this object can be compared
  – disagreement regarding the actual check:
    • `instanceof` operator
    • `getClass()` method (see Listing 4)
agenda

• check for type match
• delegation vs. inheritance
• signature of `equals()`
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners

listing 1 - an incorrect implementation

```java
public class Point3 extends Point2 {
    private int z;
    
    public boolean equals(Object p) { // overriding definition
        if (p instanceof Point3) return equals((Point3)p);
        return super.equals();
    }

    public boolean equals(Point2 p) { // overriding definition
        if (p instanceof Point3) return equals((Point3)p);
        return super.equals();
    }

    public boolean equals(Point3 p) { // extra definition
        if (p==null || z!=p.z) return false;
        return super.equals();
    }
    
    ...
}
```

• not transitive
mixed-type comparison

```java
Point2 origin(0,0);
Point3 p1(0,0,-1);
Point3 p2(0,0,1);

System.out.println(p1.equals(origin));  // calls Point3.equals(Point2)
System.out.println(origin.equals(p2));   // calls Point2.equals(Point2)
System.out.println(p1.equals(p2));      // calls Point3.equals(Point3)
```

should print: true true true

instead prints: true true false

lack of transitivity - reason

- `equals()` methods in `Point2` and `Point3` perform semantically different comparisons
- mixed-type comparison
  - involving a superclass and a subclass object
  - slice comparison: comparing only superclass part
- same-type comparison
  - involving two subclass objects
  - whole-object comparison
conceptual problem in class hierarchies

- `equals()` must be transitive
- `equals()` must compare for equality of objects

- objects in a class hierarchy have a different structure:
  - subclass objects have more fields than superclass objects
  - subclass must implement `equals()`
    - take subclass-specific fields into account
  - subclass `equals()` semantically different from superclass version

- if `equals()` permits mixed-type comparison
  - it is non-transitive (and incorrect)

- alternatives:
  - do not allow slice comparison of superclass and subclass objects
  - `equals()` method is final in the superclass

listing 2 - a debatable implementation

```java
public class Date implements java.io.Serializable, Cloneable, Comparable {
    private transient Calendar cal;
    private transient long fastTime;
    
    // ... lots of static fields ...
    ... 
    public long getTime() {
        return getTimeImpl();
    }
    private final long getTimeImpl() {
        return (cal == null) ? fastTime : cal.getTimeInMillis();
    }
    ... 
    public boolean equals(Object obj) {
        return obj instanceof Date && getTime() == ((Date) obj).getTime();
    }
}
```

- subclasses are in trouble
### a subclass - example

- implementation of subclass and its equals():

```java
public class NamedDate extends Date {
    private String name;
    public boolean equals(Object other) {
        if (other instanceof NamedDate &&
            !name.equals(((NamedDate)other).name))
            return false;
        return super.equals(other));
    }
}
```

- implementation of superclass equals():

```java
public class Date implements ... {
    public boolean equals(Object obj) {
        return obj instanceof Date &&
            getTime() == ((Date) obj).getTime();
    }
}
```

### mixed-type comparison

```
NamedDate EndOfMillenium = new NamedDate(99, 11, 31, "end of 2nd millenium AC");
NamedDate TheEnd         = new NamedDate(99, 11, 31, "end of the world");
Date NewYearsEve        = new Date(99, 11, 31);

EndOfMillenium.equals(NewYearsEve)   // slice comparison: true
NewYearsEve.equals(TheEnd)           // slice comparison: true
EndOfMillenium.equals(TheEnd)        // whole-object comparison: false
```

Not transitive
evaluation

• same transitivity problem as before
  - `Date.equals()` allows for slice comparison
  - overriding versions in a subclass performs semantically different comparison
  - incorrect non-transitive behavior

• class `Date` can avoid pitfall
  - by declaring `equals()` method final
  - by declaring entire class final

another common mistake

• another implementation of a subclass:

```java
public class NamedDate extends Date {
    private String name;
    public boolean equals(Object other) {
        if ( !(other instanceof NamedDate) )
            return false;
        else if ( !name.equals(((NamedDate)other).name) )
            return false;
        else if ( !super.equals(other) )
            return false;
        else
            return true;
    }
}
```
asymmetry

• mixed-type comparison
  – compare `Date` to `NamedDate` via `Date.equals()`
    • `NamedDate` passes `instanceof` test
    • comparison is performed; result can be false or true
  – swap the two objects:
    • compare `NamedDate` to `Date` via `NamedDate.equals()`
    • `Date` object does not pass `instanceof` test
    • these two objects will never compare equal

• violates the symmetry requirement
  – fairly common in Java platform library classes before JDK 1.3

listing 3 - a correct implementation

```java
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;
    ...

    public boolean equals(Object o) {
        if (o==this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extensions == extension &&
                pn.exchange   == exchange  &&
                pn.areaCode   == areaCode;
    }
    ...
}
```
evaluation

- `equals()` uses `instanceof` test

- class `PhoneNumber` is final
  - no subclasses can ever exist
  - transitivity problem can never come up

- flawless solution, but restricted to final classes

---

listing 4 - another correct implementation

class Golfball {
    private String brand;
    private String make;
    private int compression;
    ...
    public String brand() {
        return brand;
    }
    ...
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Golfball gb = (Golfball) obj; // Classes are equal, downcast.
        if (brand.equals(gb.brand())) // Compare attributes.
            if (make.equals(gb.make()) 
                && compression == gb.compression())
                return true;
        return false;
    }
}
listing 4 (cont.)

```java
class MyGolfball extends Golfball {
    public final static byte TwoPiece = 0;
    public final static byte ThreePiece = 1;
    private byte ballConstruction;
    ...
    public byte construction() {
        return ballConstruction;
    }
    ...
    public boolean equals(Object obj) {
        if (super.equals(obj)) {
            MyGolfball bg = (MyGolfball)obj;  // Classes equal, downcast.
            if (ballConstruction == bg.construction())
                return true;
        }
        return false;
    }
}
```

• slice comparison is not permitted

mixed-type comparison

```
Golfball original = new Golfball("xyz", "abc", 100);
MyGolfball gb1 = new MyGolfball("xyz", "abc", 100, MyGolfball.TwoPiece);
MyGolfball gb2 = new MyGolfball("xyz", "abc", 100, MyGolfball.ThreePiece);

gb1.equals(original)  // mixed-type comparison: yields false
original.equals(gb2)  // mixed-type comparison: yields false
gb1.equals(gb2)       // same-type comparison:  yields false
```
evaluation

- slice comparison is not permitted in the sense that:
  - mixed-type comparison is allowed,
  - but a Golfball does never compare equal to MyGolfball

- type check via getClass() yields false for all attempts of mixed-type comparison
  - only correct approach in class hierarchies of value types
  - where equals() must be overridden

revisiting listing 2

- can we solve problem with subclass of Date?
  - using getClass() instead of instanceof

- NO: leads to asymmetry
  - superclass Date permits mixed-type comparison
  - asymmetric behavior if subclass treats mixed-type comparison differently
    - for symmetry any subclass must also allow mixed-type comparison
example - incorrect asymmetric `equals()`

```java
public class Date {
    // as before
    public boolean equals(Object obj) {
        return obj instanceof Date
                && getTime() == ((Date) obj).getTime();
    }
}
public class NamedDate extends Date {
    private String name;
    public boolean equals(Object other) {
        if (other != null && getClass() == other.getClass()) {
            if (!super.equals(other)) return false;
            return name.equals(((NamedDate) other).name);
        }
    }
}
```

```java
NamedDate EndOfMillenium = new NamedDate(99, 11, 31, "end of 2nd millenium AC");
Date NewYearsEve = new Date(99, 11, 31);
EndOfMillenium.equals(NewYearsEve)   // slice comparison: false
NewYearsEve.equals(EndOfMillenium)   // slice comparison: true
```

cannot mix strategies

- if superclass uses `instanceof` test
  - subclass cannot switch to `getClass()` test
  - violates symmetry requirement
- if superclass uses `getClass()` test
  - subclass cannot allow mixed-type comparison via `instanceof`
  - symmetry requirement violated

**Conclusion:**

The strategy chosen for a superclass's `equals()` method determines the implementation of `equals()` for the entire class hierarchy. All classes in the hierarchy either allow slice comparison and use `instanceof` or they disallow it and use `getClass()`. 
wrap-up

- two substantially different checks for type match
  - allow mixed-type comparison between super- and subclass objects via `instanceof`
  - treat objects of different type as non-equal via `getClass()`
- implementations using `getClass()` are more robust than those using `instanceof`

wrap-up

- `instanceof` test correct only for final classes
  - or if at least method `equals()` is final in a superclass
- a subclass must not extend the superclass's state
  - but can only add functionality or
  - fields that are irrelevant for the object's state and behavior
    - such as transient or static fields
wrap-up

• implementations using `getClass()` test always comply to the `equals()` contract
  – correct and robust

• semantically very different from `instanceof` test
  – prohibits comparison of sub- with superclass objects
  – even for "trivial" class extension
    • when subclass does not add any fields and would not override `equals()`

agenda

• check for type match
• delegation vs. inheritance
• signature of `equals()`
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners
avoiding inheritance

- why not simply avoid inheritance at all?
  - problems with non-transitive `equals()` only come up in class hierarchies
- habitually declare every new class a final class
  - unless it is supposed to be a superclass
  - effort of providing a robust and correct implementation of a superclass is substantially higher

- recommendation:
  
  Keep it simple; try out a final class before you consider implementing a non-final class.

inheritance vs. delegation

- inheritance can be replaced by delegation
  - instead of implementing new class as subclass of existing class
  - new class is implemented as unrelated class that holds reference to object of existing class
  - implements same methods as existing class by delegation
delegation - example

- re-implement class `NamedDate`

```java
public class NamedDate {
    private String name;
    private Date date;
    ...
    public boolean equals(Object other) {
        if (other!=null && getClass()==other.getClass()) {
            if(date.equals(((NamedDate)other).date)
                && name.equals(((NamedDate)other).name))
                return true;
        }
        return false;
    }
    ...
    public boolean before(Date when) {
        return date.before(when);
    }
    ...
}
```

does not extend `Date`

Date field

delegation

evaluation

- tedious typing required
  - public interface must be repeated
- sensible and robust alternative to inheritance

- delegation is similar to `getClass()` approach
  - `NamedDate` not considered compatible with `Date`
limitations

• delegation cannot always replace inheritance

• example:
  – protected parts of superclass must be accessed by subclass
  – Template Method pattern from GOF book

• example:
  – frameworks in which superclass also serves as marker interface
  – class hierarchy of standard exceptions (Throwable, Error, Exception, and RuntimeException)
    • user-defined domain-specific exception must derive
    • delegation is not an option here

agenda

• check for type match
• delegation vs. inheritance
  • signature of equals()
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners
signature of `equals()`

- all implementations of `equals()` in a class hierarchy have the same signature
  - namely the one defined in class `Object`

  ```java
  boolean equals(Object other)
  ```

- reason:
  - avoid ambiguities

ambiguous `equals()` - example

```java
class Super {
    public boolean equals(Super o) { ... }
}

class Sub extends Super {
    public boolean equals(Object o) { ... }
}

class Test {
    public static void main(String args[]) {
        Sub obj = new Sub();
        if (obj.equals(obj))
            ... 
    }
}```

error: ambiguous
**Ambiguity Analysis**

<table>
<thead>
<tr>
<th>Object equals(Object)</th>
<th>distance</th>
<th>conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super equals(Super)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sub equals(Object)</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Barbara Liskov, "Program Development in Java", page 182

```java
public class Point3 extends Point2 {
    private int z;
    ...
    public boolean equals(Object o) { // overriding definition
        if (o instanceof Point3) return equals((Point3)o);
        return super.equals();
    }
    ...
    public boolean equals(Point2 p) { // overriding definition
        if (p instanceof Point3) return equals((Point3)p);
        return super.equals();
    }
    public boolean equals(Point3 p) { // extra definition
        if (p==null || z!=p.z) return false;
        return super.equals();
    }
    ...
}
```
overriding and overloading

• upside
  – can omit check for type match in one of the versions
  – all other versions uniformly delegate to `super.equals()`

• downside
  – all subclasses must override all overloaded versions in order to avoid ambiguities

agenda

• check for type match
• delegation vs. inheritance
• signature of `equals()`
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners
slice comparison

- slice comparison is a problem in class hierarchies
  - leads to non-transitive implementations of equals()
- when is it needed? when can it be avoided?

is slice comparison sensible?

- Fruit is an abstraction
  - should be an abstract class
- comparing Fruit part of apples and pears is meaningless
is slice comparison sensible?

Person

Employee

Student

- Employee and Student are roles
  - do not determine characteristics of a Person
- comparing Person part of employees and students makes sense

is slice comparison sensible?

- comparison of Rectangle slices
  - design violates LSP (Liskov Substitution Principle)
    - a Square cannot be used in all places where a Rectangle can be used
    - example: stretch by doubling one of the edges
- comparison of Square slices
  - abuse of inheritance
    - inheritance of structure only
    - instead of inheritance of behavior
  - highly questionable design
    - a Rectangle is not a Square
conclusion

• whether or not slice comparison makes sense
  – depends on semantics of class

• if we want to provide slice comparison in hierarchies of value types
  – how do we do it?

agenda

• check for type match
• delegation vs. inheritance
• signature of equals()
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners
final slice comparison

- implement `equals()` as slice comparison and make `final`.
  - `Point.equals()` compares `Point` slices.
  - Uses `instanceof` check to make sure that other instance has a `Point` part.

- `ColoredPoint` cannot override `equals()`.
  - Transitive: comparison is always slice comparison.

```java
class Point {
    final boolean equals(Object other) {
        ...
        if (!(other instanceof Point))
            return false;
        ...
    }
}
class ColoredPoint extends Point {
    ...
}
Point origin = new Point(0,0);
ColoredPoint p1 = new ColoredPoint(0,0,0xFFFF);
ColoredPoint p2 = new ColoredPoint(0,0,0x0000);
p1.equals(origin);
origin.equals(p2);
p1.equals(p2);
true
```

Point.equals()
evaluation

- no subclass can ever define `equals()` again
  - semantically correct?
  - attributes of subclasses will never be considered for comparison

- in our example:
  - color is irrelevant for comparison of (colored) points

named slice-comparison method

- different names for slice comparison and full comparison
  - `Point.equals()`
    - compares only `Point` instances
    - uses check for exact type match
  - `ColoredPoint.equals()`
    - compares only `ColoredPoint` instances
    - no slice comparison
    - uses check for exact type match
  - `MyPoint.isSameLocationAs()`
    - compares `Point` part of `ColoredPoints`
    - uses `instanceof` check
    - is transitive, because it considers only the `Point` part in all cases
named slice-comparison - example

class Point {
    boolean equals(Object other) {
        ...
        if (!(other.getClass() == getClass()))
            return false;
        ...
        final boolean isSameLocationAs(Object other) {
            ...
            if (!(other instanceof Point))
                return false;
            }
        }
    }
    class ColoredPoint extends Point {
        boolean equals(Object other) {
            ...
            if (!super.equals(other))
                return false;
            }
        }
    }

Point origin = new Point(0,0);
ColoredPoint p1 = new ColoredPoint(0,0,0xFFFF);
ColoredPoint p2 = new ColoredPoint(0,0,0x0000);
p1.equals(origin);
origin.equals(p2);
p1.equals(p2);
p1.isSameLocationAs(origin);
origin.isSameLocationAs(p2);
p1.isSameLocationAs(p2);

false: type mismatch
false: different content
true: same point part
evaluation

- collections use `equals()` method of element type for finding elements
  - example:
    - cannot find Point origin in collection of ColoredPoints
    - search would use `Point.equals()`
    - yields false because of type mismatch

```
Set fractal;
...
Point origin = new Point(0,0);
if (fractal.contains(origin))
  ...
```

note

- The next 13 slides are not contained in your handouts.
full-blown tree traversal

- key idea to solve the conceptual transitivity problem
  - mixed-type comparison only succeeds if
    - superclass slice is equals and
    - subclass fields have “default” values

- example:
  - \texttt{Point2(1,2)} is equal to \texttt{Point3(1,2,0)}
  - but is not equal to \texttt{Point3(1,2,3)}

- implementation is a non-trivial task
  - lots of issues to consider
    - how do we compare objects from different branches?
    - how do we avoid recursive calls to superclass equals?
    - how much of a burden do we impose on future subclasses?

principles

4 possible cases:

[a] other is an instance of my class

[b] other is an instance of a superclass of mine

[c] other is an instance of a subclass of mine

[d] other is none of the above (a side branch)
[a] other is of same type

- just compare my fields and
- call my super to compare the superclass fields

[b] other is an instance of a superclass

- check whether my fields have default values
- call my super to compare the superclass fields
[c] other is an instance of a subclass

- same as [b] if roles are switched
- reverses the order of the operands
  - return \texttt{other.equals(this)}

[d] other is from a side branch

- check for default values in both \texttt{this} and \texttt{other}
  - traverse both branches
- identify superclass that they have in common
  - straight up the hierarchy from then on
public class Point {

private boolean _compareFields(Object other) {
    if (other instanceof Point) {
        Point myType = (Point)other;
        if (x != myType.x || y != myType.y)
            return false;
    } else {
        if (x != 0 || y != 0)
            return false;
    }
    return true;
}

implementation - equals()

public class Point {

    ...

    public boolean equals(Object other) {
        if (other == null) return false;
        if (other == this) return true;
        if (!(other instanceof Point)) return false;
        return _navigate(other, false);
    }

    ...

}
implementation - tree traversal

```java
public class ColoredPoint extends Point {
    ...
    protected boolean _navigate(Object other, boolean reversed) {
        if (other instanceof ColoredPoint && !reversed) {
            return ((ColoredPoint)other)._navigate(this, true);
        }
        else {
            if (!_compareFields(other)) return false;
            return super._navigate(other, reversed);
        }
    }
    ...
}
```

implementation - avoid infinite loop

- process first branch
  - start at `MySubClass_1` and ascend to `MyClass` (common superclass)
- process second branch bottom up
  - switch roles of `this` and `other`; start at `MySubSubClass`; ascend to `MyClass`
- do NOT process the first branch again

```
        MySuperClass
          ↑
        MyClass

        MySubClass_1
          ↓
        MySubSubClass

        MySubClass_2
          ↑
```
implementation - the boolean flag

```java
public class ColoredPoint extends Point {
    protected boolean _navigate(Object other, boolean reversed) {
        if (other instanceof ColoredPoint && !reversed) {
            return ((ColoredPoint)other)._navigate(this, true);
        } else {
            if (!_compareFields(other)) return false;
            return super._navigate(other, reversed);
        }
    }
}
```

boilerplate code

- **all** subclasses must use this mechanism
- boilerplate code
  - trivial modification for root class
  - trivial modification for new subclass

```java
public class Point {
    protected boolean _navigate(Object other, boolean reversed) {
        if (other instanceof Point && !reversed) {
            return ((Point)other)._navigate(this, true);
        } else {
            if (!_compareFields(other)) return false;
            return true;
        }
    }
}
```
evaluation

• collections use `equals()` method of element type for adding elements
  
  – example:

  • cannot add Point origin to collection of ColoredPoints
  – if a “origin with default color” is contained in the collection
  – no duplicates allowed in a Set
  • add would use Point.equals()

  → yields true because of slice comparison

```java
Set fractal;
...
Point origin = new Point(0,0);
fractal.add(origin);
...```

agenda

• check for type match
• delegation vs. inheritance
• signature of `equals()`
• slice comparison yes/no
• correct slice comparison methods
• guidelines for practitioners
check list

- entity or value type?
- value type
- superclass or final class?
- final class
- do not override equals()
- override equals() implementation does not matter
- superclass or final class?
- slice comparison?
- mixed-type comparison
- instanceof test
- same-type comparison
- getClass() test

object equality - case studies (83)

references

- Joshua Bloch
  Effective Java
  Addison Wesley, 2001

- Peter Haggar
  Practical Java
  Addison Wesley, 2000

- Barbara Liskov with John Guttag
  Program Development in Java
  Addison Wesley 2001

- Klaus Kreft & Angelika Langer
  Secrets of equals
  Java Solutions, April 2002
  www.cuj.com/java/articles/a19.htm?topic=java

- Mark Davis
  Liberté, Egalité, Fraternité
  Java Report, 1999/2000
  www.macchiato.com/columns/Durable5.html

object equality - case studies (84)
contact info

Angelika Langer
Training & Mentoring
Object-Oriented Software Development in C++ & Java
Munich, Germany
http://www.AngelikaLanger.com

Klaus Kreft
Siemens Business Services, Munich, Germany
Email: klaus.kreft@siemens.com