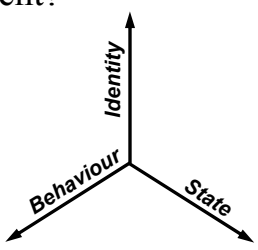
Value-Based Programming in Java

Kevlin Henney
kevlin@curbralan.com

Identity, State and Behaviour

- Objects can be characterised in terms of their identity, state and behaviour
- These aspects are rarely equal in importance
  - Is identity significant or transparent?
  - Is an object stateful or stateless?
  - Does an object have significant behaviour independent of its state?



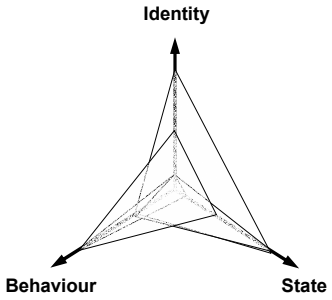
Stereotyping Objects

Entity based
Represent information in a system. Identity and state are significant, whereas behaviour is secondary and directly related to these.

Service based
Represent activities in a system. Behaviour is significant. Typically stateless and identity transparent.

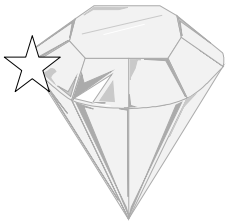
Value based
Represent transient or trivial information in a system. State is significant, behaviour directly related to it, and identity transparent.

Task based
Represent control based activities in a system. Behaviour is most significant, state and identity less so.



Values in Java

- Value-based programming in Java must be supported through a set of practices
  - Java's object model does not support value concepts directly
- Referentially-transparent objects in a reference-based language
  - Built-ins are the (unfortunate) exception



Value Objects

- Fine-grained objects for which content over identity must be emphasised
  - Need to consider creation and sharing issues
  - Override and provide methods based on comparison of content
- Value objects can improve the expressiveness of code at many levels

Whole Values

```
public final class Year ...
{
    ...
}
public final class Month ...
{
    ...
}
public final class Date ...
{
    public Date(
        Year  year,
        Month month,
        int   dayInMonth)
    ...
}
```

```
today =
  new Date(
    Year.valueOf(2002),
    Month.april,
    6);
```

```
today =
  new Date(
    6,
    Month.april,
    2002);
```

Error

```
today =
  new Date(
    Month.april,
    6,
    2002);
```

Error

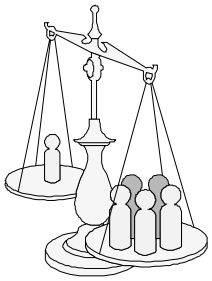
Error

Inheritance and Conversions

- Values do not typically find themselves in class hierarchies
  - However, they may often implement interfaces to mix-in properties
- Type conversions and views are normally supported by explicit *toX* and *asX* methods
  - Rather than superclass substitutability

Value Comparison

- Values support comparison based on content not just on identity
  - *equals* should be overridden whenever two objects can exist with the same state
  - *hashCode* must be overridden whenever *equals* is overridden
  - *compareTo* (from *Comparable*) may be overridden whenever there is a total ordering



Equality Comparison

- *Object*'s *equals* contract effectively forbids handling equality across different types
  - Values should be non-subclassing and concrete

```
public final class Date ...
{
    public boolean equals(Date other)
    {
        return other != null && year == other.year &&
               month == other.month && day == other.day;
    }
    public boolean equals(Object other)
    {
        return other instanceof Date && equals((Date) other);
    }
    ...
}
```

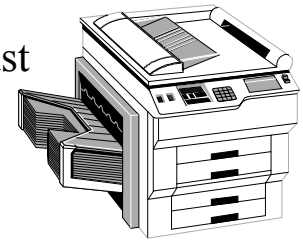

Relational Comparison

- *Comparable* imposes a total ordering over the objects of each class that implements it
  - But not across all objects, therefore only objects of the implementing class need to be considered

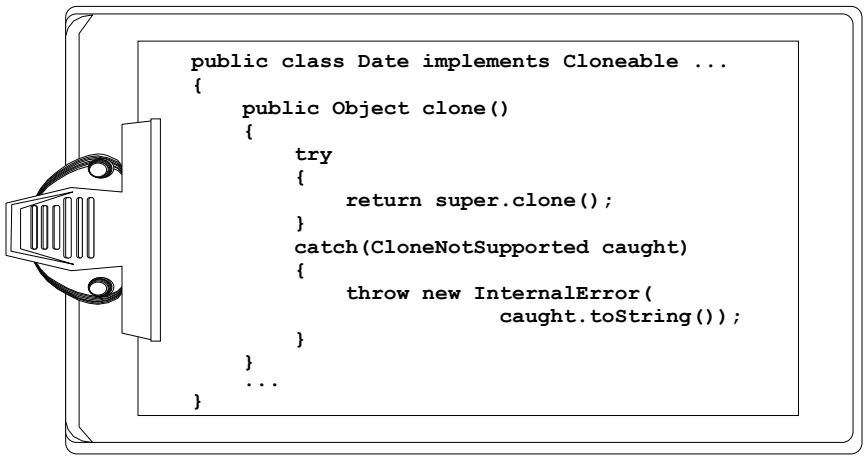
```
public final class Date implements Comparable ...
{
    public int compareTo(Date other) ...
    public int compareTo(Object other)
    {
        return compareTo((Date) other);
    }
    ...
}
```

Copying Values

- Value objects with modifiable state...
  - Must be manually copied to emulate pass by value and avoid aliasing side effects
  - Must ensure they are synchronised if shared between threads
- This means that copying must occur on the way into and out of every method



Cloning



```
public class Date implements Cloneable ...
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupported caught)
        {
            throw new InternalError(
                        caught.toString());
        }
    }
    ...
}
```

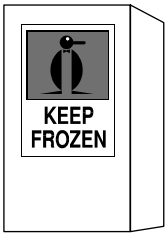
Copy Constructors

```
public final class Date ...
{
    public Date(Date that)
    {
        year  = that.year;
        month = that.month;
        day   = that.day;
    }
    ...
    private int year, month, day;
}
```

Copying with a copy constructor is based on compile-time type rather than runtime type, which only makes sense for objects not in a class hierarchy

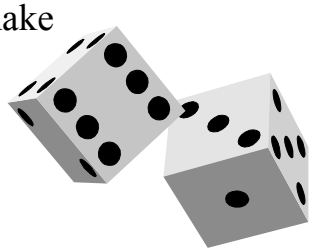
Immutable Values

- Copying objects to preserve modifiable state is tedious and error prone...
- Therefore, don't have modifiable state!
  - Inherently thread safe and shareable
  - Sorted collections can't behave "strangely": keys can't change
  - Ensure that classes are final to guarantee the contract



Value Factories

- Plain *new* expressions are not always the most effective option for value creation
- As well as copying idioms, factory objects and methods can play a role
  - A *static* factory method can make creation expressions clearer and more controllable
  - Mutable companions often partner with immutables



Mutable Companions

- A mutable companion is a factory object for immutable value objects
  - Addresses efficiency and usability issues

Immutable Value
Mutable Companion

```
public final class Date ...
{
  public int getYear() …
  public int getMonth() …
  public int getDayInMonth() …
  ...
}
```

```
public class DateManipulator ...
{
  public synchronized void set(int year, int month, int dayInMonth) …
  public synchronized Date toDate() …
  ...
}
```

Summary

- In contrast to aspirations in our own culture, objects do not live in a free society
  - They are created to serve a purpose
  - They are not all created equal
  - They shouldn't aspire to being equal
- However, respect them and their differences
  - Some objects offer service whereas others just have value