Bean-independent Properties

A proposal by Jesse Wilson
With inspiration from Shai Almog, Richard Bair, Stephen Coulbourne, Remi Forax, Mikael Grev, and Shannon Hickey.

Draft 2 - October 1, 2007 - as suggested by Boris Bokowski, adding '##' marks to the declaration so the property keyword only needs to be used once
Draft 1 - Sept 30, 2007

This proposal introduces a Property interface and Java™ language changes to make working with properties more concise.

The core design decisions of this proposal:

- Properties are per-type, not per-instance.
- Anonymous classes are used for efficient property access.
- Properties are extensible using the decorator pattern.

---

1. The Property interface

```
public interface Property<B,V> {

  /**
   * Returns the value of this property on {@code bean}.
   * @throws PropertyException if the value cannot be resolved.
   */
  V get(B bean);

  /**
   * Returns this value of this property on {@code bean},
   * or {@code defaultValue} if the value cannot be resolved.
   */
  V get(B bean, V defaultValue);

  /**
   * Sets the value of this property on {@code bean}.
   * @throws PropertyException if the value cannot be applied.
   * @return the previous value
```

```
   */
  V set(B bean, V value);

  /**
   * Returns a short name for this property, such as
   * "city" or "shippingAddress".
   */
  String getName();
}
```

---

2. Syntax - getters & setters

JavaBeans:

```
  Customer customer = ...
  String city = customer.getCity();
  customer.setCity("San Francisco");
```

Bean-independent Properties, direct access:

```
  Customer customer = ...
  String city = customer#city;
  customer#city = "San Francisco";
```

Bean-independent Properties, access via a Property object:

```
  Customer customer = ...
  Property<Customer,String> customerCityProperty
      = Customer##city;
  String city = customerCityProperty.get(customer);
  customerCityProperty.set(customer, "San Francisco");
```

---

3. Syntax - declaring a non-observable property

We introduce the '`property`' keyword, which is used in every property declaration to create the property instance. We use 2 hashes '##' to designate that this is a property, which doesn't have the same behavior as a field.

JavaBeans:

```
public class Customer {
  private String city = "";
```

```
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
}
```

Bean-independent Properties:

```
public class Customer {
  public Property<Customer,String> ##city = property("");
}
```

Bean-independent Properties with getters and setters:

This is necessary for compatibility with legacy code and to implement interfaces.

```
public class Customer {
  public Property<Customer,String> ##city = property("");
  public String getCity() {
    return #city;
  }
  public void setCity(String city) {
    #city = city;
  }
}
```

---

4. Syntax - declaring an observable property

JavaBeans:

```
public class Customer {
  private final PropertyChangeSupport propertyChangeSupport
      = new PropertyChangeSupport(this);
  private String city = "";

  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    String oldCity = this.city;
    this.city = city;
```

```
    firePropertyChangeSupport(this, "city", oldCity, city);
  }
  public void addPropertyChangeListener(PropertyChangeListener
listener) {
    propertyChangeSupport.addPropertyChangeListener(listener);
  }
  public void
removePropertyChangeListener(PropertyChangeListener listener) {

propertyChangeSupport.removePropertyChangeListener(listener);
  }
}
```

Bean-independent Properties:

```
public class Customer {
  private Property<Customer,PropertyChangeSupport>
##propertyChangeSupport
      = property(new PropertyChangeSupport(this));
  private Property<Customer,String> ##city = new
BasicPropertyBuilder(property(""))
      .observable(##propertyChangeSupport)
      .build();

  public void addPropertyChangeListener(PropertyChangeListener
listener) {
    #propertyChangeSupport.addPropertyChangeListener(listener);
  }
  public void
removePropertyChangeListener(PropertyChangeListener listener) {

#propertyChangeSupport.removePropertyChangeListener(listener);
  }
}
```

---

5. A Complete Example

```
public class Customer {

  private Property<Customer,PropertyChangeSupport>
##propertyChangeSupport
      = property(new PropertyChangeSupport(this));

  public final Property<Customer,Long> ##id
      = new BasicPropertyBuilder(property())
```

```
            .build();

    public Property<Customer,String> ##city
        = new BasicPropertyBuilder(property())
            .observable(##propertyChangeSupport)
            .build();

    public Property<Customer,String> ##emailAddress
        = new BasicPropertyBuilder(property())
            .matchingConstraint(EmailAddresses.CONSTRAINT)
            .observable(##propertyChangeSupport)
            .build();

    public Property<Customer,List<Address>> ##addresses
        = new
BasicPropertyBuilder(property(Collections.emptyList()))

.defensiveCopyOnSet(PropertyFunctions.LIST_TO_IMMUTABLE_LIST)
            .nonNull()
            .observable(##propertyChangeSupport)
            .build();

    public Customer(long id) {
      this#id = id;
    }

    public void equals(Object other) {
      return other instanceof Customer
          && Properties.equals(this, other, ##id);
    }

    public void hashCode() {
      return Properties.hashCode(this, ##id);
    }

    public void toString() {
      return String.format("(Customer:%d)", #id);
    }

    public static void main(String[] args) {
      Customer jesse = new Customer(4);
      jesse#city = "Mountain View";

      Customer james = new Customer(5);
      james#city = "Portland";
```

```
    Customer kevin = new Customer(6);
    kevin#city = "Waterloo";

    Comparator<Customer> cityComparator
        = Properties.newComparator(Customer##city);

    List<Customer> customers = Arrays.asList(jesse, james,
kevin);
    Collections.sort(customers, cityComparator);

    System.out.println("The first city is " +
customers.get(0)#name);
  }
}
```

---

6. What the compiler does behind-the-scenes: Declaring a Property

(Colours are used to show the mapping between symbols)

```
public class Customer {
  public Property<Customer,String> ##city = property("")
}
```

is equivalent to:

```
public class Customer {
  private String $v_city = "";
  public static final Property<Customer,String> $p_city = new
AbstractProperty<Customer,String>() {
    public String get(Customer c) {
      return c.$v_city;
    }
    public void set(Customer c, String value) {
      c.$v_city = value;
    }
    public void getName() {
      return "city";
    }
  };
```

---

7. What the compiler does behind-the-scenes: Accessing a Property's

value

```
  String city = customer#city;
  customer#city = "San Francisco";
```

is equivalent to:

```
  String city = Customer.$p_city.get(customer);
  Customer.$p_city.set(customer, "San Francisco");
```

---

8. What the compiler does behind-the-scenes: Accessing the Property object

```
  Property<Customer,String> customerCityProperty =
Customer##city;
  String city = customerCityProperty.get(customer);
```

is equivalent to:

```
  Property<Customer,String> customerCityProperty =
Customer.$p_city;
  String city = customerCityProperty.get(customer);
```

---

9. The BasicPropertyBuilder class

This is a regular Java Builder. It has the following API:

```
public class BasicPropertyBuilder<B,V> {
  public BasicPropertyBuilder(Property<B,V> delegate);

  /** make the returned property observable */
  public BasicPropertyBuilder observable(
       Property<B,PropertyChangeSupport> propertyChangeSupport);

  /** make the returned property enforce {@code constraint} */
  public BasicPropertyBuilder matchingConstraint(Constraint<V>
constraint);

  /** copy mutable set() parameters using the specified copying
converter */
```

```
  public BasicPropertyBuilder defensiveCopyOnSet(Converter<V,V>
copier);

  /** shorthand for {@code
#matchingConstraint(Constraints.NON_NULL)}. */
  public BasicPropertyBuilder nonNull();

  /** create the property */
  public Property<B,V> build();
}
```

Since this is implemented with simple Java, it is user-replaceable. For example, you could create custom builders that build Property objects that support Localizable names, logging, etc.

This code shows how the builder is not special - the compiler just converts the property() command in place as an argument to the builder's constructor:

```
  public Property<Customer,String> ##city
      = new BasicPropertyBuilder(property())
          .observable(##propertyChangeSupport)
          .build();
```

is equivalent to:

```
  private String $v_city;
  public static final Property<String> $p_city
      = new BasicPropertyBuilder(
            new AbstractProperty<Customer,String>() {
            public String get(Customer c) {
              return c.$v_city;
            }
            public void set(Customer c, String value) {
              c.$v_city = value;
            }
            public void getName() {
              return "city";
            })
        .observable($p_propertyChangeSupport)
        .build();
```

9. Outstanding Design Decision: Exposing bean and value types

It might be worthwhile to include additional methods on the Property interface to expose the types of the bean and value object.

```
public interface Property {
  ...
  Type<B> getDeclaringType();
  Type<V> getValueType();
}
```

10. Outstanding Design Decision: modifier keywords

Behaviour for modifier keywords '`static`', '`final`', '`transient`', etc. still requires formalization.

11. Outstanding Design Decision: Interfaces

Can interfaces expose Properties?

12. Outstanding Design Decision: Symbols

The hash '#' and double hash '##' symbols are chosen for convenience. We may want to choose a different symbol depending on user taste, such as '->'.

13. Known limitations

Properties don't play nicely with inheritance, regardless of how they are implemented.

A large number of anonymous inner classes will be generated using this solution. Each has a cost on the filesystem (which is reduced when .jars

are used) and at runtime. One alternative is reflection, which has a high runtime cost. A third alternative is a combination of reflection and runtime code-generation.

This proposal requires a '`property`' keyword. That's a fairly heavily used name in today's Java code, so another keyword may need to be used?