JavaBeans
The Component Model in Java

CUGS

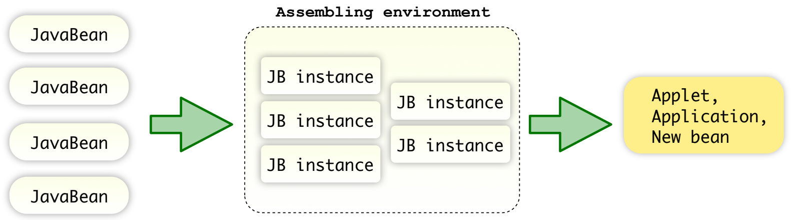
Mikhail Chalabine
mikch@ida.liu.se

JavaBeans

- **JavaBeans is the component model for Java**
  - Portable
  - Platform-independent
  - Written in Java
  - API introduced in Feb. 1997
- **A bean is a reusable software component**
- **JavaBeans != Enterprise JavaBeans**

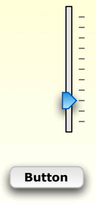
Reference: JavaBeans Tutorial @ java.sun.com

Programming model

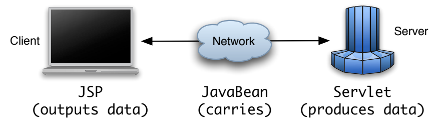


JavaBeans & Component Types

- **Visual components**
  - Used in Swing, AWT
  - Visual application builders (visual composition)
    - Work flow: load, customize (size, color), save (persist)
    - Eclipse VEP (Visual Editor Project)
    - NetBeans



- **Non-visual components**
  - In Java2EE, Hibernate
  - Capture business logic or state

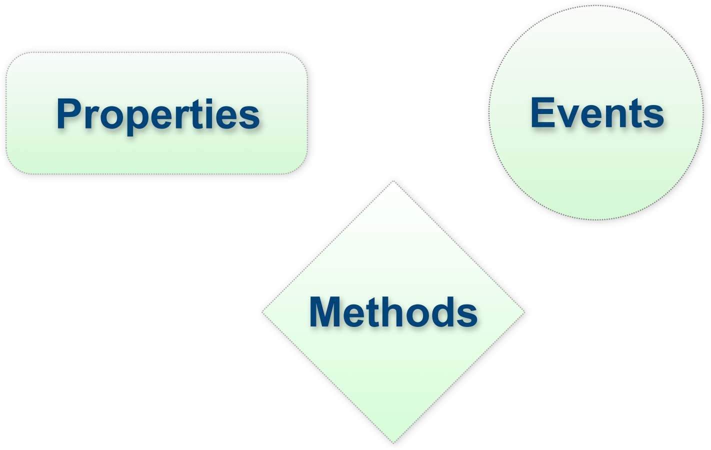


A reusable component in Java

```
Class
```

- Hides implementation, conforms to interfaces, encapsulates data
- Is written to a standard (component specification)
  - Implements the serializable interface (persistence)
  - No-argument constructor
    - E.g., instantiation through reflection
  - Design patterns or BeanInfo class (introspection)
  - Core features (methods, properties, events)

Beans' core features



Beans' features: Methods

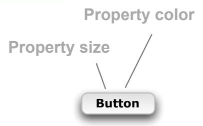
- Standard Java

Methods

Bean's features: Properties (1)

- Appearance and behavior characteristics

```
import java.io.Serializable;

public class MyJavaBean implements Serializable {

        private String first_name;
        private float income;
        ...
```



- Visual components
  - Builder tools can discover and expose
  - Customization - modifying appearance or behavior at design time by
    - Property editors (visual, programmable)
    - Bean customizers (visual, programmable)

Properties

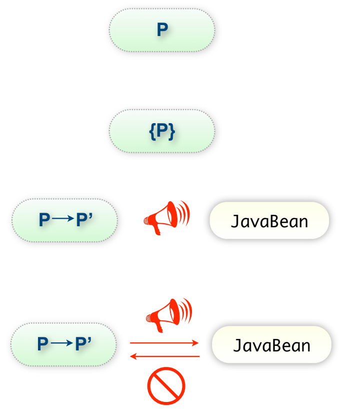
Bean’s features: Properties (2)

- Specification suggests to have ‘getters’ and ‘setters’

```
public String getFirst_name() {
        return first_name;
}

public void setFirst_name(String first_name) {
        this.first_name = first_name;
}

public float getIncome() {
        return income;
}

public void setIncome(float income) {
        this.income = income;
}
```

Bean’s features: Properties (3)

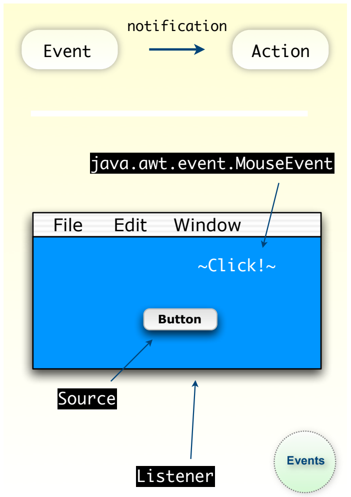
- **Simple** - A single-value bean property whose changes are independent of changes in any other property
- **Indexed** - A bean property that supports a range of values
- **Bound** - A bean property for which a change to the property results in a **notification** being sent to some other bean
- **Constraint** - A bean property for which a change to the property results in **validation** by another bean. The other bean may reject the change if it is not appropriate (veto).



Properties

Bean’s features: Event model (1)

- **Fire** (send) / **handle** (receive)
  Components broadcast events and the underlying framework delivers the events to the components to be notified
- **Sources**
  - Define and fire events
  - Define methods for registering listeners
- **Listeners**
  - Get notified of events
  - Register using methods defined by sources



Events

Bean’s features: Event model (2)

- Write event class
  - Create your own custom event class named `<NAME>Event` or use an existing event class, e.g. `ActionEvent`

```
ActionEvent e;
```

- Write event listener (handler, receiver)
  - write `<NAME>Listener` interface and provide implementation of it or reuse existing listener interfaces, e.g., `ActionListener` or complete handlers, so-called, **adapters**, e.g. `MouseAdapter()`

```
public class ButtonHandler implements ActionListener() {…}
```

- Write event source bean (Event generator)
  - `JButton button = new JButton(“Fire”);`
  - In your custom bean implement `add<NAME>Listener()` and `remove<NAME>Listener()` methods. Implemented in `JButton`.
- Register event listener

```
button.addActionListener( new ButtonHandler());
```

Events

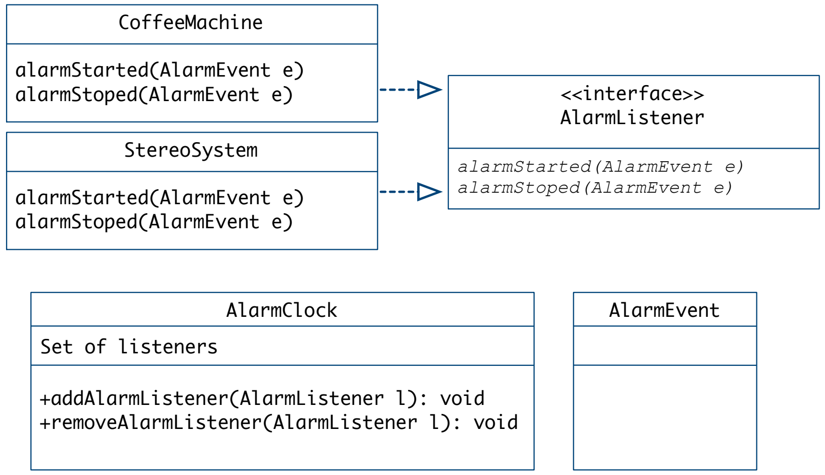
Example: Alarm Clock

- Properties
  - Current time
  - Alarm time
  - Alarm status (set/not set)
- Events
  - Alarm (source)

Example

```
class AlarmClock implements Serializable {

    public AlarmClock() {…}

    public boolean getAlarmStatus() {…}

    public void setAlarmStatus(boolean value) {
      …
    }

     …
}
```

Example cont.


CoffeeMachine
alarmStarted(AlarmEvent e)
alarmStoped(AlarmEvent e)
StereoSystem
alarmStarted(AlarmEvent e)
alarmStoped(AlarmEvent e)
<<interface>>
AlarmListener
alarmStarted(AlarmEvent e)
alarmStoped(AlarmEvent e)
AlarmClock
Set of listeners
+addAlarmListener(AlarmListener l): void
+removeAlarmListener(AlarmListener l): void
AlarmEvent

Advanced features

- Optional class (see below): `AlarmClockInfo`
- Optional class (study): `AlarmClockCustomizer`

Discovering features through introspection

- We concluded that
  - For a bean to be the source of an event, it must implement methods to add and remove listener objects for that type of event

```
add<EventName>Listener(<EventName>Listener listener)
remove<EventName>Listener(<EventName>Listener listener)
```

  - For a bean to be the listener of an event, it must implement the `<EventName>Listener` interface
- We see that
  - 'add', 'remove', 'Listener', <BeanClass>Info, <BeanClass>Customizer form syntactic patterns
- We also said that
  - component specification suggests that bean properties should have **setters** and **getters**

Reminder (lecture on Java Reflection)

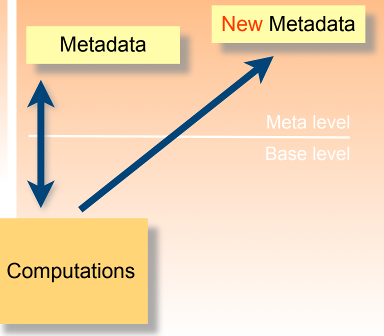
Representing metalevel concepts at the base level is called ***reification***.

```
/* Instantiate a metaobject */
Robot ourRobot= new Robot(...);

/* Obtain its (meta) class */
Class rClass= ourRobot.getClass();
```

rClass represents (**reifies**) the **class** meta-level concept at the base level.

- **Locate** classes, methods, data accesses
- **Allocate** new classes, methods, fields
- **Remove** classes, methods, fields

Reflection is **computations** about the metamodel in the base model



Discovering beans' features (1)

- **Automatic (implicit)**
  - adhering to design patterns makes a bean's features discoverable through **introspection**

```
import java.io.Serializable;

public class MyJavaBean implements Serializable {

      private String first_name;

      public String getFirst_name() {
            return first_name;
      }
      public void setFirst_name(String first_name) {
            this.first_name = first_name;
      }
      ...
```

Discovering beans' features (2)

- **Manual (explicit)**
  - **BeanInfo class** (visual components)
    - Code that defines and initializes properties
      - Make properties visible / invisible, etc.
      - Expose / hide methods that the bean implements
        - `setHidden()` etc.
    - Use when
      - Bean code does not follow the standard naming convention (no introspection possible)
      - You intend to hide some features

Further topics

- Java techniques
  - Serialization and persistence in JavaBeans
- JavaBeans
  - More on customizers
  - More on properties

Evaluation

- Strengths
  - Simple - easy to use
  - Standard - mix vendors
  - Applicable for GUI development
- Weaknesses
  - Only suitable for GUI development
  - Not usable for non-programmers
  - Weak component market