EventBus

`EventBus` allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). It is designed exclusively to replace traditional Java in-process event distribution using explicit registration. It is *not* a general-purpose publish-subscribe system, nor is it intended for interprocess communication.

Example

```
// Class is typically registered by the container.
class EventBusChangeRecorder {
  @Subscribe public void recordCustomerChange(ChangeEvent e) {
    recordChange(e.getChange());
  }
}
// somewhere during initialization
eventBus.register(new EventBusChangeRecorder());
// much later
public void changeCustomer()
  ChangeEvent event = getChangeEvent();
  eventBus.post(event);
}
```

One-Minute Guide

Converting an existing `EventListener` -based system to use the `EventBus` is easy.

For Listeners

To listen for a specific flavor of event (say, a `CustomerChangeEvent`)...

- **...in traditional Java events:** implement an interface defined with the event -- such as `CustomerChangeEventListener` .
- **...with `EventBus` :** create a method that accepts `CustomerChangeEvent` as its sole argument, and mark it with the `@Subscribe` annotation.

To register your listener methods with the event producers...

- **...in traditional Java events:** pass your object to each producer's `registerCustomerChangeEventListener` method. These methods are rarely defined in common interfaces, so in addition to knowing every possible producer, you must also know its type.
- **...with `EventBus` :** pass your object to the `EventBus.register(Object)` method on an `EventBus` . You'll need to make sure that your object shares an `EventBus` instance with the event producers.

To listen for a common event supertype (such as `EventObject` or `Object`)...

- **...in traditional Java events:** not easy.
- **...with `EventBus` :** events are automatically dispatched to listeners of any supertype, allowing listeners for interface types or "wildcard listeners" for `Object` .

To listen for and detect events that were dispatched without listeners...

- **...in traditional Java events:** add code to each event-dispatching method (perhaps using AOP).
- **...with `EventBus`:** subscribe to `DeadEvent`. The `EventBus` will notify you of any events that were posted but not delivered. (Handy for debugging.)

For Producers

To keep track of listeners to your events...

- **...in traditional Java events:** write code to manage a list of listeners to your object, including synchronization, or use a utility class like `EventListenerList`.
- **...with `EventBus`:** `EventBus` does this for you.

To dispatch an event to listeners...

- **...in traditional Java events:** write a method to dispatch events to each event listener, including error isolation and (if desired) asynchronicity.
- **...with `EventBus`:** pass the event object to an `EventBus`'s `EventBus.post(Object)` method.

Glossary

The `EventBus` system and code use the following terms to discuss event distribution:

Event	**Any object that may be *posted* to a bus.**
Subscribing	The act of registering a *listener* with an `EventBus`, so that its *handler methods* will receive events.
Listener	An object that wishes to receive events, by exposing *handler methods*.
Handler method	A public method that the `EventBus` should use to deliver *posted* events. Handler methods are marked by the `@Subscribe` annotation.
Posting an event	Making the event available to any *listeners* through the `EventBus`.

FAQ

Why must I create my own Event Bus, rather than using a singleton?

`EventBus` doesn't specify how you use it; there's nothing stopping your application from having separate `EventBus` instances for each component, or using separate instances to separate events by context or topic. This also makes it trivial to set up and tear down `EventBus` objects in your tests.

Of course, if you'd like to have a process-wide `EventBus` singleton, there's nothing stopping you from doing it that way. Simply have your container (such as Guice) create the `EventBus` as a singleton at global scope (or stash it in a static field, if you're into that sort of thing).

In short, `EventBus` is not a singleton because we'd rather not make that decision for you. Use it how you like.

Can I unregister a listener from the Event Bus?

Yes, using `EventBus.unregister`, but we find this is needed only rarely:

- Most listeners are registered on startup or lazy initialization, and persist for the life of the application.

- Scope-specific `EventBus` instances can handle temporary event distribution (e.g. distributing events among request-scoped objects)
- For testing, `EventBus` instances can be easily created and thrown away, removing the need for explicit unregistration.

Why use an annotation to mark handler methods, rather than requiring the listener to implement an interface?

We feel that the Event Bus's `@Subscribe` annotation conveys your intentions just as explicitly as implementing an interface (or perhaps more so), while leaving you free to place event handler methods wherever you wish and give them intention-revealing names.

Traditional Java Events use a listener interface which typically sports only a handful of methods -- typically one. This has a number of disadvantages:

- Any one class can only implement a single response to a given event.
- Listener interface methods may conflict.
- The method must be named after the event (e.g. `handleChangeEvent`), rather than its purpose (e.g. `recordChangeInJournal`).
- Each event usually has its own interface, without a common parent interface for a family of events (e.g. all UI events).

The difficulties in implementing this cleanly has given rise to a pattern, particularly common in Swing apps, of using tiny anonymous classes to implement event listener interfaces.

Compare these two cases:

```
class ChangeRecorder {
  void setCustomer(Customer cust) {
    cust.addChangeListener(new ChangeListener() {
      public void customerChanged(ChangeEvent e) {
        recordChange(e.getChange());
      }
    };
  }
}
```

versus

```
// Class is typically registered by the container.
class EventBusChangeRecorder {
  @Subscribe public void recordCustomerChange(ChangeEvent e) {
    recordChange(e.getChange());
  }
}
```

The intent is actually clearer in the second case: there's less noise code, and the event handler has a clear and meaningful name.

What about a generic `Handler<T>` interface?

Some have proposed a generic `Handler<T>` interface for `EventBus` listeners. This runs into issues with Java's use of type erasure, not to mention problems in usability.

Let's say the interface looked something like the following:

```
interface Handler<T> {
  void handleEvent(T event);
}
```

Due to erasure, no single class can implement a generic interface more than once with different type parameters. This is a giant step backwards from traditional Java Events, where even if `actionPerformed` and `keyPressed` aren't very meaningful names, at least you can implement both methods!

Doesn't `EventBus` destroy static typing and eliminate automated refactoring support?

Some have freaked out about `EventBus`'s `register(Object)` and `post(Object)` methods' use of the `Object` type.

`Object` is used here for a good reason: the Event Bus library places no restrictions on the types of either your event listeners (as in `register(Object)`) or the events themselves (in `post(Object)`).

Event handler methods, on the other hand, must explicitly declare their argument type -- the type of event desired (or one of its supertypes). Thus, searching for references to an event class will instantly find all handler methods for that event, and renaming the type will affect all handler methods within view of your IDE (and any code that creates the event).

It's true that you can rename your `@Subscribed` event handler methods at will; Event Bus will not stop this or do anything to propagate the rename because, to Event Bus, the names of your handler methods are irrelevant. Test code that calls the methods directly, of course, will be affected by your renaming -- but that's what your refactoring tools are for. We see this as a feature, not a bug: being able to rename your handler methods at will lets you make their meaning clearer.

What happens if I `register` a listener without any handler methods?

Nothing at all.

The Event Bus was designed to integrate with containers and module systems, with Guice as the prototypical example. In these cases, it's convenient to have the container/factory/environment pass *every* created object to an `EventBus`'s `register(Object)` method.

This way, any object created by the container/factory/environment can hook into the system's event model simply by exposing handler methods.

What Event Bus problems can be detected at compile time?

Any problem that can be unambiguously detected by Java's type system. For example, defining a handler method for a nonexistent event type.

What Event Bus problems can be detected immediately at registration?

Immediately upon invoking `register(Object)`, the listener being registered is checked for the *well-formedness* of its handler methods. Specifically, any methods marked with `@Subscribe` must take only a single argument.

Any violations of this rule will cause an `IllegalArgumentException` to be thrown.

(This check could be moved to compile-time using APT, a solution we're researching.)

What `EventBus` problems may only be detected later, at runtime?

If a component posts events with no registered listeners, it *may* indicate an error (typically an indication that you missed a `@Subscribe` annotation, or that the listening component is not loaded).

(Note that this is *not necessarily* indicative of a problem. There are many cases where an application will deliberately ignore a posted event, particularly if the event is coming from code you don't control.)

To handle such events, register a handler method for the `DeadEvent` class. Whenever `EventBus` receives an event with no registered handlers, it will turn it into a `DeadEvent` and pass it your way -- allowing you to log it or otherwise recover.

How do I test event listeners and their handler methods?

Because handler methods on your listener classes are normal methods, you can simply call them from your test code to simulate the `EventBus` .

Why can't I do <magic thing> with `EventBus` ?

`EventBus` is designed to deal with a large class of use cases really, really well. We prefer hitting the nail on the head for most use cases to doing decently on all use cases.

Additionally, making `EventBus` extensible -- and making it useful and productive to extend, while *still* allowing ourselves to make additions to the core `EventBus` API that don't conflict with any of your extensions -- is an extremely difficult problem.

If you really, really need magic thing X, that `EventBus` can't currently provide, you should file an issue, and then design your own alternative.