

Ink tutorial – 2

Written by: Atzmon hen-tov

Empowering users to extend the system

In the previous tutorial you learned the basics of Ink. Specifically, you learned how to develop an Ink application that is configurable by end-users.

In this tutorial we will show you how you can further empower your users to extend the application model with new classes. This technique is known as Adaptive Object Model.

In the next tutorial we proceed on this theme and show how end users can control the system behavior, still without needing to write Java code.

The source code of this tutorial is located in the same project as tutorial 1, under separate “tutorial2” sub-folders.

Tutorial 1 and Tutorial 2 are defined as two separate DSLs in the projects (dsls.ink file). This provides, among other things, namespace separation and thus allows the elements with the same name (classes, instances) to reside in the same project. Each DSL definition defines the Java package for the DSL Java classes. The corresponding Java packages for tutorials 1 and 2 are `org.ink.tutorial1` and `org.ink.tutorial`.

The example system

We shall use the same fictitious Magazine Subscription System that we used in Tutorial 1.

New requirement

For reasons of Customer Relationships Management, the customer requires that some promotional offers will require the user to fill in a form.

Different offers may require different forms (e.g., in big discount offers, it is reasonable to require the subscriber to provide personal information for follow up).

There is no predefined set of forms. Users should be able to define new forms as needed without requiring developer involvement or deployment of a new software release.

Solution

Users will be able to define new forms. In each offer, the users will be able to define what type of form the subscriber is required to fill in. A form will be represented in the system as an Ink class.

Implementation

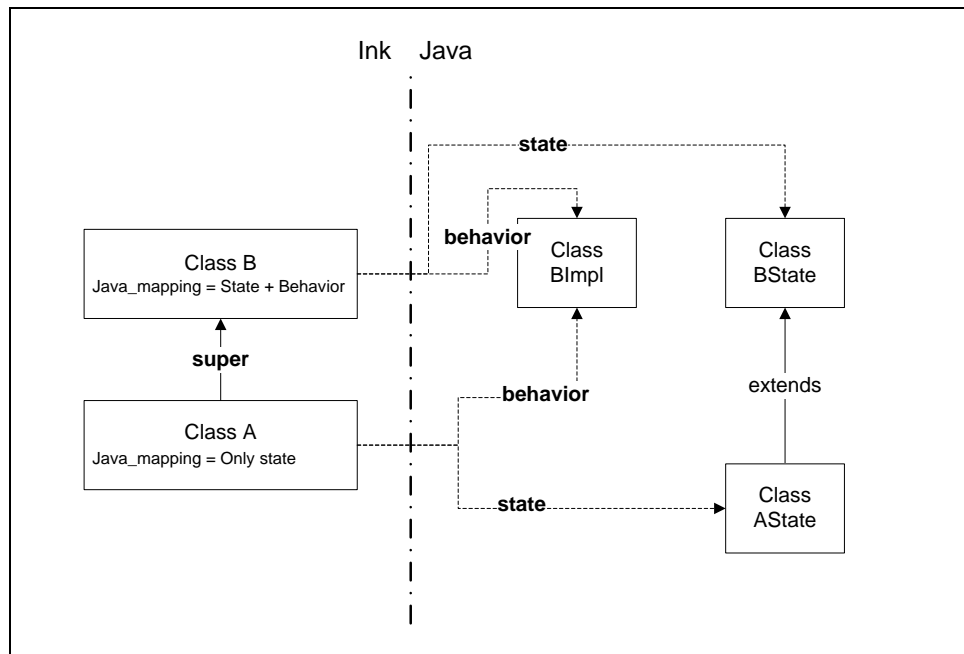
Recall from Tutorial 1, that there is mapping between Ink classes and Java classes. Each Ink class has two corresponding Java classes; the structure class (State) and the behavior class (Impl). In fact, Ink allows some flexibility in defining these mapping. The `java_mapping` property in an Ink class definition, defines how the Ink class will be mapped to Java.

```
Class id="PercentageDiscountOffer" class="ink.core:InkClass" super="BaseOffer" abstract=false{
  java_path ""
  java_mapping "State_Behavior"
  properties{
    property class="ink.core:DoubleAttribute"{
      name "percentage"
      mandatory true
    }
  }
}
```

In the example above, PercentageDiscountOffer is defined to be mapped to a State class and a Behavior class. This means the Ink compiler will generate a specific State class (PercentageDiscountOfferState) and the developer will write a specific Behavior class (PercentageDiscountOfferImpl).

You may define your class to have only State, only Behavior, or None. The meaning of such definition is that you choose not to have a **specific Java class** for your Ink class. When an Ink class doesn't have a specific Java class mapped to it, Ink will use the Java class that is mapped to the Ink Class' super-class in the corresponding role (State/Behavior).

Suppose we have Ink class A that inherits (super=) Ink class B. Ink class B has "regular" java_mapping (State and Behavior) and Ink class A has "only State". The actual mapping to Java classes will be as shown below.



This means that there isn't a specific behavior class for class A and Ink will use BImpl when instantiating A.

Note that absence of a specific State class doesn't prevent the Ink class to have additional properties. Properties defined on an Ink class that doesn't have a specific State class may be accessed via reflection API as in the code snippet below. See more on Mirror here <http://bracha.org/mirrors.pdf>

```
Mirror m = registrationForm.reflect();  
String firstName = (String)m.getPropertyValue("firstName");
```

In the Magazine Subscription System we will use “no java” mapping. This allows users to define new Ink classes without requiring new Java classes in order to execute their instances.

See more on Java mapping at the end of this tutorial.

Changes in the existing application

A new *register()* method is added to BaseOffer class. When an end-user subscribes to a magazine, this method is called with the user filled registration form. The method will validate the form is of the right type (as defined in the offer) and will return a digital receipt which is (for the purpose of the tutorial), a serialization of the registration form.

```
String register(BaseRegistrationForm registrationForm);
```

We will explain later in the tutorial how the user specifies in the offer which type of form is required for that offer.

Preparing for dynamically defined classes

In order to allow our end users to define new registration form classes as the one below, and for those to be functioning, we need to do some preparations.

```

Class id="Students_registration_form" class="MetaRegistrationForm"
super="BaseRegistrationForm" {
    java_path ""
    java_mapping "No_Java"
    properties{
        property class="ink.core:StringAttribute"{
            name "firstName"
            display_name "First Name"
        }
        property class="ink.core:StringAttribute"{
            name "lastName"
            display_name "Last Name"
        }
        property class="ink.core:StringAttribute"{
            name "email"
            display_name "Email address"
        }
    }
}

```

First of all we need to have a Base class for all registration form to inherit (super=).

```

Class id="BaseRegistrationForm" class="MetaRegistrationForm"
super="ink.core:InkObject" {
    java_path ""
    java_mapping "State_Behavior_Interface"
}

```

The Behavior class of BaseRegistrationForm will be mapped to all user-defined registration form classes.

The BaseRegistrationForm interface has one method.

```

public interface BaseRegistrationForm extends InkObject {

    String serialize();
}

```

This method is used by the *register()* method of BaseOfferImpl.

```
// verify the right type of registration form.  
/  
/// generate the receipt  
if (!ok) {  
    throw new RuntimeException("Bad registration form.");  
}  
else {  
    result = registrationForm.serialize();  
}
```

Since it is not known in advance which properties exist on the registration form (it can be any sub-class of BaseRegistrationForm), we use reflection to iterate over all properties, to generate the serialized string.

Note that for brevity, the code assumes that all properties are simple.

```

public class BaseRegistrationFormImpl<S extends BaseRegistrationFormState>
    extends InkObjectImpl<S> implements BaseRegistrationForm {

    @Override
    public String serialize() {
        String result = "";
        Mirror mirror = this.reflect();
        PropertyMirror[] propertiesMirrors = mirror.getPropertiesMirrors();
        for (PropertyMirror propertyMirror : propertiesMirrors) {
            String name = propertyMirror.getName();
            Object value = mirror.getPropertyValue(propertyMirror.getIndex());
            result = result + name + "=" + value + ",";
        }
        result = result.substring(0, result.length() - 1);
        return result;
    }
}

```

To fully understand the code above, see the Java-Doc of the Mirror API.

Now we will get back to the issue of specifying in each Offer the type of registration form to use.

Specifying which type of registration form to use

In the example from Tutorial 1, we had the following offers:

```

Object id="Active_offers" class="ActiveOffers" {
    offers{
        offer ref="students_30_percent_discount_for_1_year"
        offer ref="students_50_percent_discount_for_2_years"
        offer ref="students_60_percent_discount_for_3_years"
    }
}

```

Suppose we want two types of registration forms:

- Basic – first name, last name, email

- High value – for high value customers, the subscriber may “opt in” for receiving special offers via email.

We want all student offers to have the Basic registration form but the “3 years” offer should have the High Value registration form.

The registration forms:

```
Class id="basic_registration_form" class="MetaRegistrationForm" super="BaseRegistrationForm" {
  java_path ""
  java_mapping "No_Java"
  properties{
    property class="ink.core:StringAttribute"{
      name "firstName"
      display_name "First Name"
    }
    property class="ink.core:StringAttribute"{
      name "lastName"
      display_name "Last Name"
    }
    property class="ink.core:StringAttribute"{
      name "email"
      display_name "Email address"
    }
  }
}

Class id="high_value_registration_form" class="MetaRegistrationForm"
super="basic_registration_form" {
  java_path ""
  java_mapping "No_Java"
  properties{
    property class="ink.core:BooleanAttribute"{
      name "optIn"
      display_name "Can we send special offers to your email?"
      mandatory true
    }
  }
}
```

Defining the Basic registration form to all students offers can be done on the base instance for all student offers.


```
Object id="Student_Offers_Template_For_2010" class="ink.tutorial2:BaseOffer" abstract=true{
  studentOnlyOffer true
  renewalOnlyOffer false
  validUntil 2013/11/01
  registrationFormType ref="basic_registration_form"
}
```

In the “3 years” offer, we define the High Value registration form.

```
Object id="students_60_percent_discount_for_3_years" super="Student_Offers_Template_For_2010"
class="ink.tutorial2:PercentageDiscountOffer"{
  percentage 60.0
  conditionForPeriodsSigned 3
  freeIssues 3
  registrationFormType ref="high_value_registration_form"
}
```

Now we get to the tricky part. Note that the value provided to the registrationFormType property is a reference to an Ink class. This is to designate the type of registration form required by this offer. The registrationFormType property (defined on BaseOffer), is of type “type”. We don’t want the user to be able to select just any Ink class as the registrationFormType. For that we need to constraint the definition. As you can see below, this is indeed so.

```
54         mandatory true
55     }
56 }
57 }
58
59 Object id="students_60_percent_discount_for_3_years" super="Student_Of
60 percentage 60.0
61 conditionForPeriodsSigned 3
62 freeIssues 3
63 registrationFormType ref="
64 }
65
66 Object id="Student1_registr
67     firstName "Lior"
68     lastName "Schachter"
69     email "lior@ink.org"
70 }
```

BaseRegistrationForm - ink.tutorial2
basic_registration_form - ink.tutorial2
high_value_registration_form - ink.tutorial2

Problems Tasks Progress Search

ished after 1.045 seconds

Runs: 9/9 Errors: 0 Failures: 0

org.ink.tutorial2.TestTutorial2Test [Runner: JUnit 4] Failure Trace
org.ink.tutorial1.TestTutorial1Test [Runner: JUnit 4]

The way to achieve this in Ink is to define a new meta-class. A meta-class is the class of a class.

`java.lang.Class` is the meta-class of all Java classes. But... it is **final** ☹. You can not inherit `java.lang.Class` to define sub-sets of classes such as “all registration forms”. In Ink, you can.

The class of all classes in Ink is `InkClass` (analog to `java.lang.Class`), but it is not **final**.

To define a meta-class in Ink, inherit (`super=`) `InkClass` or one of its sub-classes.

By having MetaRegistrationForm as the meta-class of all registration-forms, we can specify in the registrationFormType property that we want to allow just types of registration-forms and not just any type.

```
Class id="MetaRegistrationForm" class="ink.core:InkClass" super="ink.core:InkClass" abstract=true{
  java_path ""
  java_mapping "No_Java"
  properties{
    property class="ink.core:StringAttribute"{
      name "author"
      mandatory true
    }
  }
}

Class id="BaseOffer" class="ink.core:InkClass" super="ink.core:InkObject" abstract=true{
  java_path ""
  java_mapping "State_Behavior_Interface"
  properties{
    property class="ink.core:BooleanAttribute"{
      name "studentOnlyOffer"
      mandatory true
    }
    property class="ink.core:DateAttribute"{
      name "validUntil"
      mandatory true
    }
    // ...
    property class="ink.core:Reference"{
      name "registrationFormType"
      type ref="MetaRegistrationForm"
      mandatory false
    }
  }
}
```

Note that BaseRegistrationForm and all user defined registration forms have MetaRegistrationForm as their class.

```
Class id="BaseRegistrationForm" class="MetaRegistrationForm" super="ink.core:InkObject" abstract=true{  
    java_path ""  
    java_mapping "State_Behavior_Interface"  
}
```

```
Class id="basic_registration_form" class="MetaRegistrationForm" super="BaseRegistrationForm" {...}
```

```
Class id="high_value_registration_form" class="MetaRegistrationForm" super="basic_registration_form"  
{...}
```

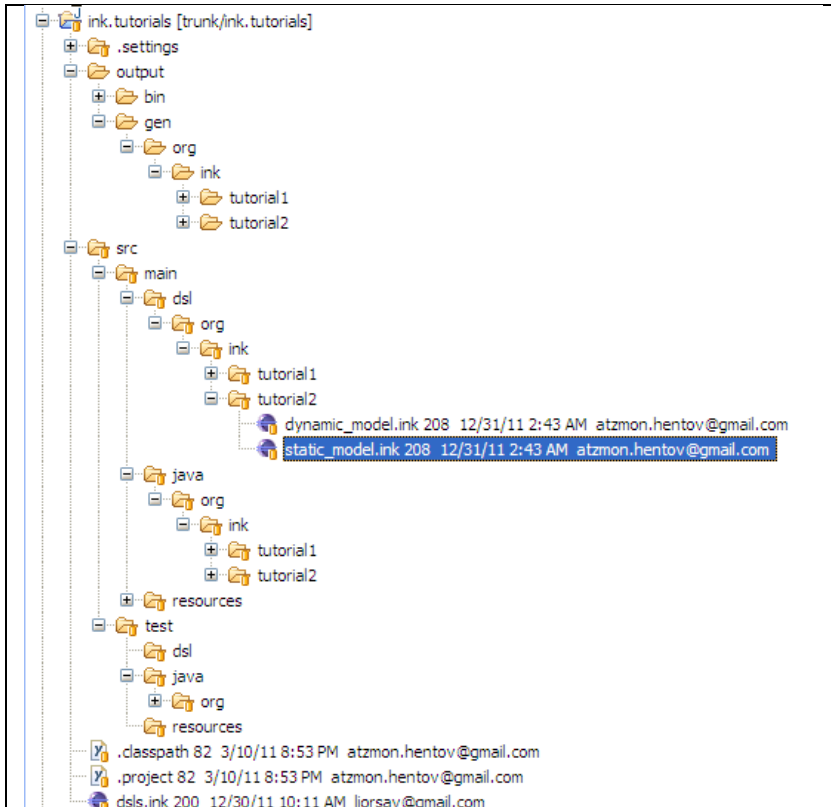
Summary

In this tutorial you learned how to develop applications that allow the user to extend the application model by defining their own classes. You also learned that Ink supports meta-class extensibility. This trait of Ink allows for better reuse when writing Ink application. More about this in Tutorial 4.

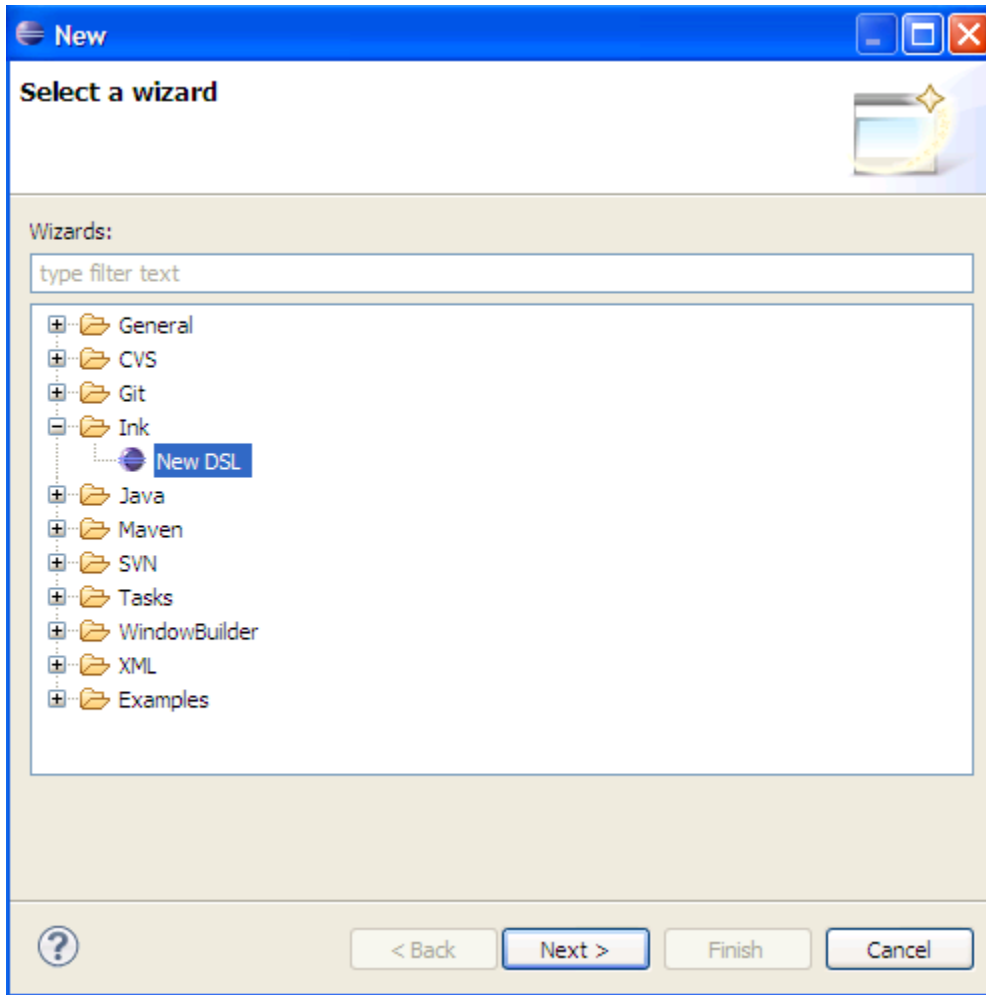
The next tutorial will enhance on AOM in Ink, showing how users can control the system behavior in the classes they define.

More on the Ink environment in eclipse

The structure of an ink project is as following:

	<ul style="list-style-type: none">• “src” folder contains source code and sub-divided into “main” and “test” sub-folders.<ul style="list-style-type: none">○ “main” folder contains the source code and sub-divided into “dsl” and “java” sub-folders that contains the ink and Java source code respectively.○ “test” folder contains unit-test code and has the same structure as “main”.• “output” folder contains generated artifacts:<ul style="list-style-type: none">○ “bin” – Compiled java classes (.class files)○ “gen” – Java source code generated by the Ink plugin. Mostly State classes.
--	--

“dsl.ink” defines the DSLs in the project. You don’t have to edit it manually. Use the DSL wizard as shown below.



More on Java mapping

In addition to the State and Behavior classes, an Ink class may be mapped to a Java interface. The interface is written by the developer. This allows for better abstraction in the Java implementation of Ink applications.

When working with interfaces, a new interface should be defined when a sub-class has new public methods. Otherwise the super's interface may be used.

For example, BaseOffer class from Tutorial 1, has "State_Behavior_Interface" java_mapping.

```
Class id="BaseOffer" class="ink.core:InkClass" super="InkObject"
  java_path ""
  java_mapping "State_Behavior_Interface"
```

The corresponding Java elements are as following.

```
public interface BaseOffer extends InkObject {...}
public abstract class BaseOfferImpl extends InkObjectImpl implements BaseOffer {...}
public interface BaseOfferState extends org.ink.core.vm.lang.InkObjectState {...}
```

You probably noticed that BaseOfferState, which is generated by Ink, is an interface and not a class. This is for technical reasons.

You may use Java Generics to have more convenient access to the injected State object.

```
public abstract class BaseOfferImpl<S> extends BaseOfferState<S> extends
    InkObjectImpl<S> implements BaseOffer {...}

// Makes the return type of getState() to be BaseOfferState. No need for casting.
```

The full list of java_mapping options is shown in the screenshot below.


```
Class id="BaseRegistrationForm" class="MetaRegistrationForm" super="ink.core:InkObject" abstract=true {  
  java_mapping "State_Behavior_Interface"  
}
```

State_Behavior_Interface

State_Behavior

State_Interface

Behavior_Interface

Only_State

Only_Behavior

Only_Interface

No_Java