# Ink tutorial – 2

# 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 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 nor 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.

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.

## 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.

For example, BaseOffer class from Tutorial 1, has “State\_Behavior\_Interface” java\_mapping.



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 read more about it here – Link to PLDE paper.

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

**public** **abstract** **class** BaseOfferImpl<S **extends** BaseOfferState> **extends**

InkObjectImpl<S> **implements** BaseOffer {…}

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

