# Ink tutorial – 1

# What is ink all about?

Ink is a framework for developing agile software using Java. It allows you to write code that can be customized and enhanced easily. Much of this possible customization can be achieved without writing any additional code.

Ink lets developers expose their code as a Domain Specific Language (DSL), a simple syntax to configure the behavior of your system.

DSLs are interpreted by the Ink framework at runtime in a just-in-time manner. This helps to keep the edit-execute cycle in the scale of seconds rather than minutes.

In this tutorial you will learn the concept of declarative DSLs and how to implement a simple DSL in Ink.

To try the tutorial code yourself, take a look at “Ink Tutorial– Installation Guide” at the end of this tutorial.

## An example system

To demonstrate the advantages and usage of Ink, we will now describe the system which we will use for the examples in this tutorial.  
This is a typical Java based system, to which we will gradually introduce the Ink framework as the tutorial progresses.   
  
Even though it’s a “fictitious” system, it shares many of the qualities of modern enterprise systems.

Imagine a magazine subscription management system. The system is used to subscribe readers to the magazine, renew their subscription, etc.

The system already has the following classes implemented in plain Java (no Ink):

* A Magazine class, implementing the interface A\_Product. Each instance represents a magazine, along with its price.

**public** **interface** A\_Product {

String getID();

String getName();

Double getPrice();

}

* A Customer class, implementing the interface A\_Customer:

**public** **interface** A\_Customer {

**public** String getName();

**public** String getEmail();

**public** String getCreditCardNumber();

**public** **CustomerType** getCustomerType();

}

* A Subscription class, implementing the A\_Subscription interface. This class represents the subscription of a single customer to a particular magazine.

**public** **interface** A\_Subscription {

// The customer

A\_Customer getCustomer();

// Subscription is to this magazine

A\_Product getMagazine();

// List price

**double** getPrice(); - is it same as product price? Multiplied by months?

// 1 year, 2 years, etc.

**int** getPeriod();

// Start period of the subscription

Date getPeriodStart();

// When the customer commits to the subscription, call confirm() – todo: is it really needed

**boolean** confirm();

// Indicates if this subscription is in effect or just draft– todo: is it really needed

**boolean** isConfirmed();

}

## New requirement

One day, a new functionality is required from the system – add the possibility of promotions and discounts. Since “discount” and “promotion” nowadays can mean more or less anything☺, this new ability needs to be flexible and easily adaptable to the new ideas of marketing and sales managers. This is where Ink DSLs are handy, allowing this new functionality to be easily implemented and easily configurable upon need.

## Solution

The new requirement implies that A\_Subscription should include the special terms of the subscription.

So we will add the following methods to A\_SubscriptionOffer:

**double** getPromotionalPrice(); // Price after discount

**int** getFreePeriods(); // # of free issues

String getPromotionalMessage(); // Promotional message

(maybe it would be more clear if we just add getBestOffer?)

The implementation of these methods will look for special offers that are relevant for the subscription and select the best offer.

We’ll leave the implementation of the above methods to later in the tutorial – they are “the glue” between the Java part and the Ink part.

Also, we will need to implement the promotions themselves: A\_SpecialOffer interface (see below), an abstract base class (BaseOfferImpl), and two concrete implementations of promotions:

* percentage discount (x% of original price)
* fixed-price offer (pay x instead of y).

**public** **interface** A\_SpecialOffer {

// Is this subscription eligible to the special offer?

**boolean** isEligible(A\_Subscription subscription);

**double** getPromotionalPrice(A\_Subscription subscription);

**int** getFreeIssues(A\_Subscription subscription);

String getPromotionalMessage(A\_Subscription subscription);

}

We’ll implement these classes as an Ink DSL. This will allow us to create multiple variations of such promotions, without writing additional code, or even restarting the system.

## Creating an Ink DSL

Ink DSLs actually define instances of Java objects, but are not written in Java.

In our case, a definition of a special offer would look something like this:

Object id="ExampleOffer" class="FixedPercentageDiscountOffer" {

percentage 20.0

customerType Student

validUntil 2011/11/01

renewalOnlyOffer false

freeIssues 0

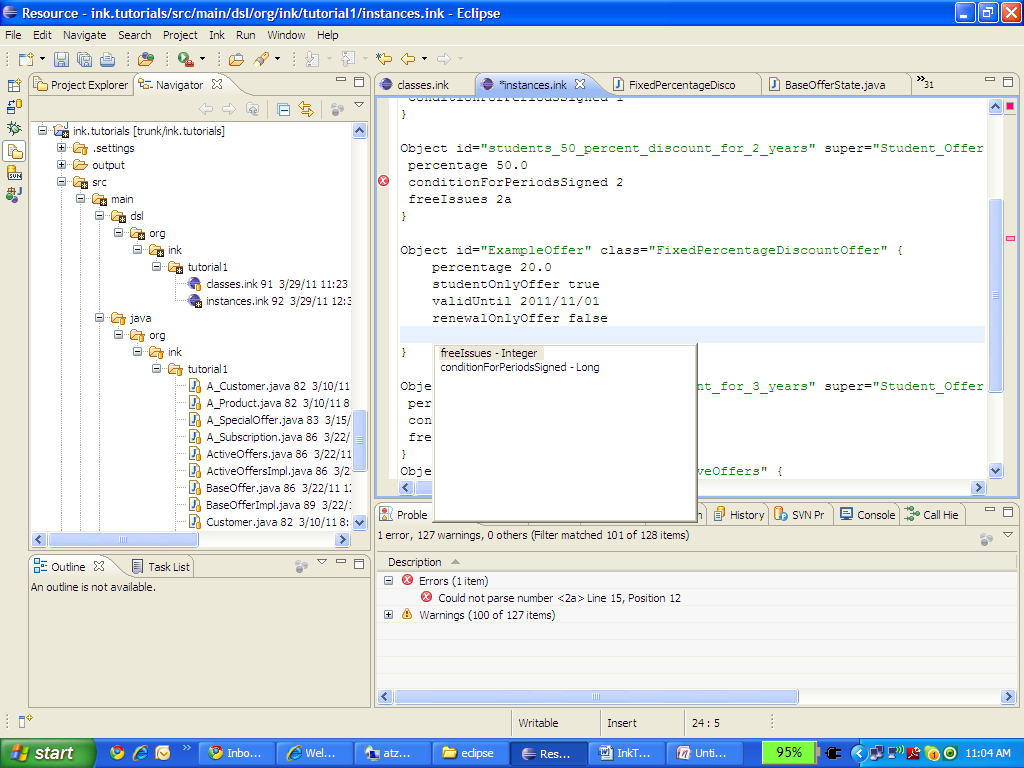
}

As you might have guessed, this is an offer that gives 20% discount to students that subscribe before 1/11/2011.

At runtime, the Ink VM reads the DSL script , creates the expected instance of FixedPercentageDiscountOffer class and injects the values from the Ink object (ExampleOffer) to the Java instance.

Writing such scripts is especially easy since the Ink plugins for eclipse offer you powerful IDE features similar to what eclipse JDT provides to Java developers.

This includes auto-completion, incremental compilation, navigation commands, etc., (see table of useful shortcuts at the end of this tutorial.)



# How does it work?

Ink is a type-safe language. Scripts are checked for consistency by the incremental compiler every time you save an Ink file.

In order to use the Ink script from the previous section, some ground work has to be done.  
Basically, what we have to do is to define a template of a BaseOffer instantiation.

These are the templates we’ve used:

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:BooleanAttribute"{

name "renewalOnlyOffer"

mandatory true

}

property class="ink.core:LongAttribute"{

name "conditionForPeriodsSigned"

mandatory false

}

property class="ink.core:IntegerAttribute" {

name "freeIssues"

mandatory false

}

property class="ink.core:DateAttribute"{

name "validUntil"

mandatory true

}

}

}

Class id="FixedPercentageDiscountOffer" class="ink.core:InkClass" super="BaseOffer" abstract=false{

java\_path ""

java\_mapping "State\_Behavior"

properties{

property class="ink.core:DoubleAttribute"{

name "percentage"

mandatory true

}

}

}

As you can see, we’ve defined two **Ink classes**: BaseOffer and FixedPercentageDiscountOffer.   
FixedPercentageDiscounterOffer has one property, “percentage” of type Double.   
Since it inherits the BaseOffer class, it has the properties defined in BaseOffer as well. Note that the Ink class definitions are written in Ink as well. The syntax used by Ink is called SDL. You can learn more about it here: <http://en.wikipedia.org/wiki/Simple_Declarative_Language>

Later you’ll see that these model classes have corresponding Java classes, but for the purpose of authoring Ink scripts it is inconsequential.

The classes define the DSL, and also specify which matching Java classes needs to be written (this is “the glue” we mentioned before – connecting the Java and the Ink to a single module). More about this in “Glue from Java to Ink” later in this tutorial.

The “ExampleOffer” we defined before is actually an instance of this Java class, that will be described later in more detail.

## Making marketing department happy

All this mumbo-jumbo made us almost forget about actually implementing the requirement – adding promotions and discounts to the system.

Here we will define some promotions, and the “pool” of possible promotions.

For promotions that share common logic and behavior, we can use DSL inheritance, just like with Java classes.

For example, let’s define an abstract offer for students, and two inheriting offers.

Object id="Student\_Offers\_Template\_For\_2010" class="ink.tutorial1:BaseOffer" **abstract=true** {

studentOnlyOffer true

renewalOnlyOffer false

validUntil 2011/11/01

}

Object id="students\_30\_percent\_discount\_for\_1\_year" **super="Student\_Offers\_Template\_For\_2010"** class="ink.tutorial1:FixedPercentageDiscountOffer"{

percentage 30.0

conditionForPeriodsSigned 1

}

Object id="students\_50\_percent\_discount\_for\_2\_years" **super="Student\_Offers\_Template\_For\_2010"** class="ink.tutorial1:FixedPercentageDiscountOffer"{

percentage 50.0

conditionForPeriodsSigned 2

freeIssues 2

}

In this example, students\_30\_percent\_discount\_for\_1\_year inherits the values of “studentOnlyOffer”, “renewalOnlyOffer” and “validUntil”, and assigns values to “percentage” and “conditionForPeriodsSigned”.

Note that Ink uses “abstract” and “super” keywords very similarly to Java.

The next step is to define the “pool” of all the active offers.

This is an instance that contains references to other instances. This is done by “ref” keyword, and then specifying the IDs of specific instances.

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"

}

}

Now that we already mastered the basics, we know that the structure of ActiveOffers class should also be defined:

Class id="ActiveOffers" class="ink.core:InkClass" super="ink.core:InkObject" {

java\_path ""

java\_mapping "State\_Behavior\_Interface"

properties {

property class="ink.core:ListProperty"{

type ref="ink.core:List"

name "offers"

mandatory true

inheritance\_constraints "Instance\_Can\_Refine\_Inherited\_Value" – maybe we can do without this?

list\_item class="ink.core:Reference"{

type ref="ink.tutorial1:BaseOffer"

name "offer"

}

}

}

}

The ActiveOffers class has one property of type “list of (Base) Offer”. Since instances have concrete textual representation in Ink, the property definition specifies the name of the property (offers) as well as the name of each item in the list (offer).

## Glue from Java to Ink

The last piece of the puzzle, to complete and run our code, is the part connecting Ink to Java.

An Ink class is mapped to two Java classes (see figure below)

### Structure class (class A in the diagram)

Named the same as the Ink class. Contains the properties as defined in the Ink class (e.g. percentage).

This class is generated by the Ink compiler. The generated class names matches their counterpart Ink class id.

### Behavioral class (class AImpl in the diagram)

Named the same as the Ink class + “Impl” suffix. This class is written by the DSL developer (you), it implements the required behavior. When Ink VM instantiates the behavioral class it injects to the new instance the corresponding instance of the structure class.



The basic idea behind Ink DSLs is that behavior classes use the injected ink instances as configuration.

The injected ink instance (in the form of an instance of the Java structure class) is available to the behavior class by the getState() method (see below). Note the use of Generics in the class definition to provide for automatic casting of the state instance.

In the example below, the promotional price calculation uses the percentage property as configuration.

**package** org.ink.tutorial1;

**public** **class** FixedPercentageDiscountOfferImpl**<S extends FixedPercentageDiscountOfferState>**

**extends** BaseOfferImpl**<S>** **implements** BaseOffer {

**int** nInvocations = 0;

@Override

**public** **double** getPromotionalPrice(A\_Subscription subscription) {

**double** result = 0.0;

countInvocations();

**if** (isEligible(subscription)) {

result = subscription.getPrice()

\* (100.0 - **getState().getPercentage()**) / 100.0;

} **else** {

result = subscription.getPrice();

}

**return** result;

}

…

}

## Calling Ink from non-Ink Java

When you integrate Ink into you existing Java application, you need to instantiate the Ink VM. This is done by calling the **static** instance() method of InkVM class.

@Override

**public** **double** getPromotionalPrice() {

ActiveOffers offers = InkVM.*instance*().getContext().getState("ink.tutorial1:Active\_offers").getBehavior();

**return** offers.getPromotionalPrice(**this**);

}

## Exercise

Implement FixedPriceOffer, write instances, enhance the unit-tests (TestTutorial1Test) to cover it.

## Summary

In this tutorial you learned how to create a new Ink DSL. The steps are as following:

* Language developer (you) defines the DSL – by writing Ink classes
* Language developer implements the DSL – by writing Java behavior classes
* Language user uses the DSL – by writing Ink instances.

Note that using the DSL does not require changes in Java-code and thus may provide a faster way to deliver functionality to production.

In the next tutorial you will learn how to allow super users to define new, functioning Ink classes without need for writing Java classes at all.

## Ink tools

|  |  |  |
| --- | --- | --- |
| Tool | Activated by | Does |
| Compiler | Saving an ink file | Validates consistency of the Ink scripts |
| Open Ink Element (similar to alt-ctrl-R in Java) | alt-ctrl-q | Search for and open an Ink element by it’s name (id) |
| Open declaration (similar to F3 in Java) | F3 | Navigate to selected Ink element |
| Open Java implementation | F4 | Navigate from Ink script to it’s Java implementation class |

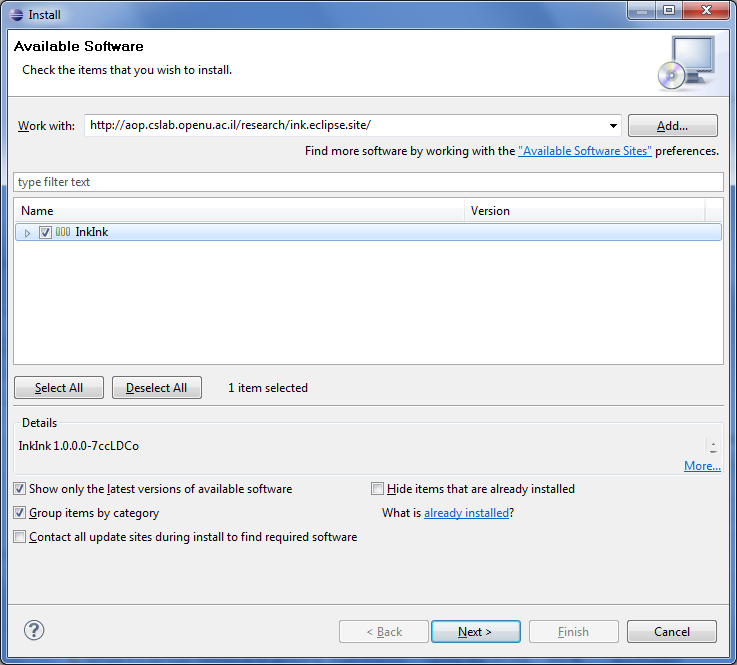
# Ink Tutorial– Installation Guide

1. Install Eclipse IDE:

Download the Eclipse IDE latest version for Java developers from <http://www.eclipse.org/downloads/>

TODO add eclipse type (regular \ EE etc, ad icon)!

1. Install the Ink-Framework Eclipse plugin:
   1. In Eclipse, go to ‘Help’🡪’Install New Software…’
   2. Type in the Ink-Framework update site URL : <http://aop.cslab.openu.ac.il/research/ink.eclipse.site/> :

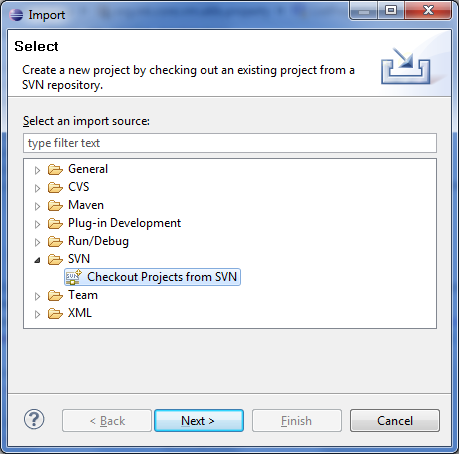
Mark the ‘InkInk’ flag and press ‘Next’.

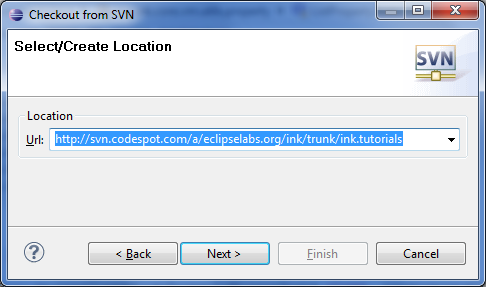
* 1. After the installation process is completed, restart Eclipse.

1. Install Eclipse SVN plugin :
   1. Follow the instructions here : <http://subclipse.tigris.org/servlets/ProjectProcess?pageID=p4wYuA>

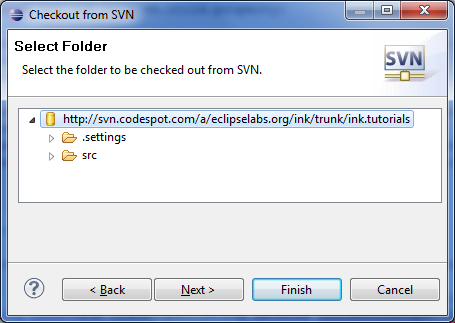
Or

* 1. Simply go to ‘Help’🡪’Install New Software’ 🡪 type in the update site URL <http://subclipse.tigris.org/update_1.6.x> 🡪 ‘Next’…

1. Download the ‘ink-tutorial’ Eclipse project:
   1. Go to ‘File’🡪’Import’ and choose ‘Checkout Projects from SVN’.
   2. Choose ‘Create a new repository location’ 🡪 ‘Next’.
   3. Paste the URL: <http://svn.codespot.com/a/eclipselabs.org/ink/trunk/ink.tutorials> and press ‘Next’.



* 1. Choose the root element and press ‘Finish’.



* 1. Restart Eclipse IDE.
  2. Launch Clean-Build : ‘Project’🡪’Clean’ (make sure ‘Clean all projects’ is marked)🡪Press ‘OK’.

TODO Need to explain debug settings to skip proxy - org.ink.core.vm.proxy