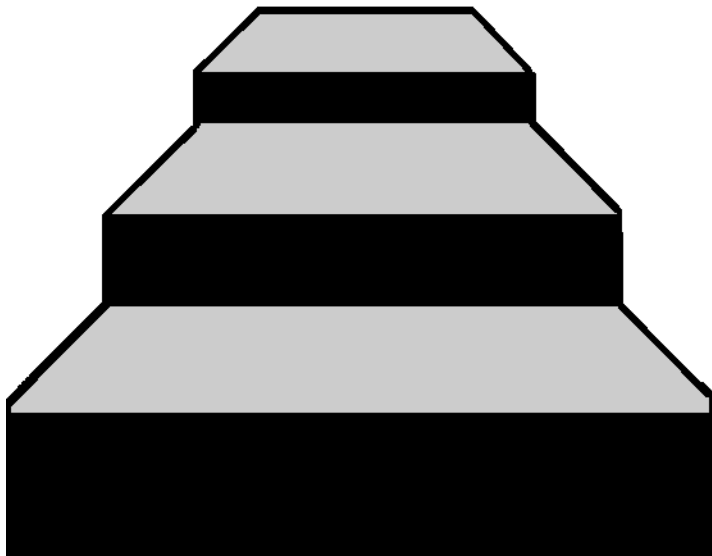
ST-Developer for Java Reference Manual



STEP Tools® Software

Contributors

Written by Joe Fritz.

© Copyright 1991-2010 STEP Tools, Inc. — All Rights Reserved.

This document contains proprietary and confidential information of STEP Tools, Inc. The contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior written permission of STEP Tools, Inc.

ST-Developer, ST-Machine, ST-Viewer, and the ST prefix are trademarks of STEP Tools, Inc. STEP Tools is a registered trademark of STEP Tools, Inc. Other brand or product names are the trademarks or registered trademarks of their respective holders.

STEP Tools, Inc.
14 First Street
Troy, New York 12180

Phone . . . . (518) 687-2848
Fax . . . . (518) 687-4420
E-Mail . . . . info@steptools.com
Web. . . . . http://www.steptools.com

Contents . . . iii

1 — Getting Started . . . 1
1.1 Overview . . . 1
1.2 A Simple Example . . . 2
1.3 Pre-installed Application Protocols . . . 3

2 — Reading, Writing, and Traversing Data . . . 5
2.1 Overview . . . 5
2.2 Models and Populations . . . 5
2.3 Managing Entity Instances . . . 6
2.4 Accessing Entity Instances by Type . . . 7
2.5 Reading Part 21 Files . . . 8
2.6 Writing Part 21 Files . . . 8
2.7 Entity Instance Identifiers . . . 9
2.8 STEP Header . . . 10

3 — From EXPRESS to Java Classes . . . 11
3.1 Overview . . . 11
3.2 Command Line . . . 12
3.3 Namespaces . . . 12
3.4 Primitive Types . . . 13
3.5 EXPRESS Schemas . . . 13
3.5.1 Population . . . 14
3.5.2 Schema . . . 15
3.6 Entity Types . . . 15
3.6.1 Single Inheritance . . . 16
3.6.2 Multiple Inheritance . . . 17
3.6.3 Complex Instances . . . 19
3.6.4 Views . . . 20

3.7 Enumeration Types 20
3.8 Select Types 21
3.8.1 Creating Select Instances 22
3.8.2 Selection Naming Conventions 23
3.8.3 Nested Selects 23
3.9 Aggregation Types 26

1 Getting Started

1.1 Overview

The ST-Developer for Java programming environment consists of a EXPRESS compiler that can generate Java classes (**express2java**), and a set of foundation classes (**stdev.jar**) that provide services such as reading and writing instances to STEP Part 21 exchange files.

You can build a Java application around any EXPRESS information model. The **express2java** compiler generates pure Java classes for every definition in the model, which you can then use to create and manipulate data sets. We include pre-generated classes for all STEP Application Protocols (**ap203.jar**, **ap214.jar**, etc), and other common models (IFC, CIS/2), so you may not even need the compiler to get started!

All of the classes that we generate or provide are pure Java, so they will work on any platform with a JVM. Just drop the jar files into your class path and go!

Below are some simple examples and following chapters describe the interface in more detail. **Reading, Writing, and Traversing Data** (Chapter 2, pp. 5) describes how instance data is grouped in memory and how to control the reading and writing of files. **From EXPRESS to Java Classes** (Chapter 3, pp. 11) describes how EXPRESS constructs are represented in the generated Java classes, how to get and set values, and other operations on the instance data.

1.2 A Simple Example

This is a Java version of the “tutorial1” demo included with the ST-Developer C++ API. This creates a small data set based on a simple “picture” schema with points and lines.

```
import java.io.IOException;

import com.steptools.stdev.*;
import com.steptools.stdev.p21.Part21Writer;
import com.steptools.schemas.picture.*;

public class tutorial1 {
    public static void main (String[] args) throws IOException {

        Model mod = new Model(Schema.SCHEMA);
        Population pop = (Population)mod.getPopulation();

      /* Create a point using the default constructor
       *  and use the update methods to set its values. */
        Point point1 = pop.newPoint();
        point1.setX(1.0);
        point1.setY(0.0);

        Point point2 = pop.newPoint ();
        point2.setX(2.5);
        point2.setY(4.0);

        Point point3 = pop.newPoint ();
        point3.setX(5.0);
        point3.setY(0.0);


        /* Create a Line Object */
        Line line = pop.newLine();
        line.setEnda(point1);
        line.setEndb(point2);

        /* Create a Circle with Center (0,0), radius 1.5 */
        Point point4 = pop.newPoint();
        point4.setX(0.0);
        point4.setY(0.0);

        Circle circle1 = pop.newCircle ();
        circle1.setRadius(1.5);
        circle1.setCenter(point4);

        /* Create a Text Object centered at point3 */
        Text text = pop.newText ();
        text.setLabel ("A Little Picture");
        text.setCenter(point3);
```

```
        /* Create another Circle Object */
        Circle circle2 = pop.newCircle();
        circle2.setRadius(1.5);
        circle2.setCenter(point2);

        Part21Writer writer = new Part21Writer();
        writer.write("tutorial1.stp", mod);
    }
}
```

To compile, you will need the location of the **stdev.jar** file. All jar files provided with ST-Developer are located in the installation “lib/java” directory. The ROSE environment variable is normally set to the ST-Developer installation directory, so we can use **$ROSE/lib/java** or **%ROSE%\lib\java** when setting our class path.

Our sample EXPRESS schema is in **picture.exp**. Call the EXPRESS compiler to generate the Java source files. The ***.java** source files will be put in the **java_classes** subdirectory:

```
> express2java picture.exp
```

Next we will compile the source files with the Java compiler to create ***.class** files. Create a directory called **obj** to hold the output. Note that the source files are under a special namespace for the “picture” schema.

```
> mkdir java_objs
> javac -classpath "%ROSE%"\lib\java\stdev.jar -d java_objs \
      tutorial1.java \
      java_classes\com\steptools\schemas\picture\*.java
```

Run the application. We need to set the class path to find both the ST-Developer base classes (**stdev.jar**) and the classes for our schema.

```
> java -classpath "%ROSE%"\lib\java\stdev.jar;java_objs tutorial1
```

The demos directory includes two sample applications that demonstrate how to use the interface. See the README.txt for tutorial1 for instructions on how to compile and run that sample application

1.3 Pre-installed Application Protocols

In the previous example, we generated the Java classes for our own example sche-

ma, but when you build your applications, you are more likely to work against one of the STEP application protocols, like AP203 or AP214. ST-Developer ships with pre-built Java libraries for the common APs and other models like CIS/2 and IFC.

You can start programming immediately, just by adding the appropriate library to your classpath, such as **ap203lib.jar**, **ap214lib.jar**, or others. The ST-Developer on-line documentation has the complete list of APs available as well as class listings, browsable EXPRESS definitions, recommended practices and more.

```
ap201lib.jar  com.steptools.schemas.explicit_draughting
ap202lib.jar  com.steptools.schemas.associative_draughting
ap203lib.jar  com.steptools.schemas.config_control_design
ap209lib.jar  com.steptools.schemas.structural_analysis_design
ap214lib.jar  com.steptools.schemas.automotive_design
ap215lib.jar  com.steptools.schemas.ship_arrangement_schema
ap216lib.jar  com.steptools.schemas.ship_moulded_form_schema
ap218lib.jar  com.steptools.schemas.ship_structures_schema
ap224lib.jar  com.steptools.schemas.feature_based_process_planning
ap225lib.jar  com.steptools.schemas.building_design_schema
ap227lib.jar  com.steptools.schemas.plant_spatial_configuration
ap232lib.jar  com.steptools.schemas.technical_data_packaging
ap238lib.jar  com.steptools.schemas.integrated_cnc_schema
cislib.jar    com.steptools.schemas.structural_frame_schema
ifclib.jar    com.steptools.schemas.ifc2x2_final
```

2 Reading, Writing, and Traversing Data

2.1 Overview

The ST-Developer for Java library includes foundation classes that provide data management and serve as the base which is extended by the EXPRESS compiler. These classes provide the high-level data management, store metadata for the EXPRESS schema, and implement a late-bound API to the STEP data.

Unless otherwise indicated, all classes and interfaces are declared in the **com.steptools.stdev** namespace. Definitions in the schema namespace are generated by the EXPRESS compiler under **com.steptools.schemas.<*schname*>**, where ***schname*** is the name of the schema.

2.2 Models and Populations

STEP instances are organized into model and populations. A model coresponds to an entire a STEP Part 21 file, and a population corresponds to the **HEADER** or **DATA** section within the file. Most models will contain a header and a data population, but some might contain several data populations (the second edition of Part 21 allows more than one data section). Populations contain the STEP entity instances. Each population is associated with exactly one EXPRESS schema.

A model is represented by the **Model** class in **com.steptools.stdev**. A population is represented by a generated class named **Population** in the schema namespace. The

Population class extends the **PopulationBase** class in **com.steptools.stdev**. There is no schema-specific model class, since a schema can contain instances from several schemas. Any method of **Model** that returns a population object is declared to return **PopulationBase**, which you may need to cast down to the intended type.

To create a **Model** with a **Population**, pass the schema to the constructor for the **Model**. The **Population** can then be obtained by calling **getPopulation()** on the **Model**. The following example shows how to create and populate a model. In this example, the schema is not imported in order to emphasize which definitions come from the ST-Developer class library, and which are EXPRESS compiler generated classes.

```
/* In this example, no schemas are imported.
* actual code is likely to be much less verbose.
*/
Model mod = new Model
(com.steptools.schemas.config_control_design.Schema.SCHEMA);
 com.steptools.schemas.config_control_design.Population pop
 = mod.getPopulation();

/* Create a instance */
com.steptools.schemas.config_control_design.Cartesian_point point
= pop.newCartesianPoint();
```

Generally, a **Model** will only contain a single **Population** of user instances. The Part 21 interface uses an additional **Population** to hold the STEP header information. It is possible, however, to create a **Model** with more than one **Population**. This can be used, to create a Part 21 file with multiple data sections. To create a **Model** with more than one **Population**, use the default, no argument, constructor for **Model**, and use the **newPopulation** method to create each **Population** in the model. Each **Population** must have an identifier, which is a **java.lang.Object** that is specified as a parameter to the **newPopulation** method. For models saved in Part 21 files, this identifier must be a string that gives the name of the data section in the Part 21 file.

To get a default **Population** from a **Model**, use one of the **getPopulation** methods. With no arguments, or with a null parameter **getPopulation** return the default **Population** of the **Model**. If you specify an argument it is interpreted as an identifier,and **getPopulation** returns the **Population** with the specified identifier, or null if no **Population** exists with that identifier.

2.3 Managing Entity Instances

STEP entity instances are represented by interfaces generated by the EXPRESS compiler. For every entity in a schema, the compiler creates a corresponding interface in the schema namespace. These generated interfaces extend the **EntityInstance**

interface.

The schema-specific **Population** class contains a method to create entity instance in the schema. This method is named **new<Entname>** where Entname is the name of the entity, as mapped to Java (first letter is capitalized, all others lowercase).

To create an entity instance in a **Population**, call the appropriate new method on the **Population**.

To remove an entity instance from a **Population** call the removeInstance method. The instance is removed from the **Population**, but it will not get deleted from memory until the garbage collector notices that the instance is unreferenced.

2.4 Accessing Entity Instances by Type

The **Population** classes provides several methods to find instances by type. This is the most common way to access the data after loading a **Model** from a file on disk. The **getExtent** and **getFolder** methods each return an **EntityExtent** containing all the entity instances of a given type. The **EntityExtent** that the **getExtent** method returns also contain instances of all subtypes of the specified type. The **EntityExtent** class implements the **java.util.Set** interface, so you can obtain an **Iterator**, and traverse over the instances.

The following code traverses over all the **product** entity instances in a **Population**.

```
import com.steptools.schemas.config_control_design.*;
import com.steptools.stdev.*;
import java.util.*;

/* much further down - inside a class */
void processProducts(Population pop) {
   EntityExtent prods = pop.getExtent(Product.DOMAIN);
   Iterator itor = prods.iterator();
   while (itor.hasNext()) {
      Product prod = (Product) itor.next();
      /* Now do something with the product */
   }
}
```

2.5 Reading Part 21 Files

The **Part21Parser** class reads a Part 21 file into memory. To read a file, create an instance of **Part21Parser**, then call one of the **parse** methods to read the **Model**. For flexibility, the parse method is overloaded to take either a file name, a **java.io.File** object, or a **java.io.Reader** object. Whichever version is called, a **Model** is returned. The **Part21Parser** object may be reused to read multiple files.

The following code demonstrates how to read a Part 21 file, given the file name.

```
/* at the top of your code: */
import com.steptools.stdev.*;
import com.steptools.stdev.p21.*;

/* inside a class: */
Model read_file (String filename) throws STDevException, IOException {
   Part21Parser parser = new Part21Parser();
   return parser.parse(filename);
}
```

The Part 21 file header contains string that identifies the STEP schema. Normally the schema is found by loading the class named **com.steptools.schemas.<schema-name>.Schema**. You can override this behavior, by subclassing **Part21Parser**, and overriding the **getSchema** method. The following code demonstrates how to read every Part 21 file as if it were an AP203 file:

```
import com.steptools.stdev.p21.*;

class AP203Parser extends Part21Parser {
   public SchemaBase getSchema(String name) {
      /* The schema object is in a static field of the Schema class */
      return com.steptools.schemas.config_control_design.Schema.SCHEMA;
   }
}
```

2.6 Writing Part 21 Files

The Part21Writer class is used to write a Model to secondary storage. To write a Part 21 file, create an instance of Part21Writer, and call one of the write methods. The write method is overloaded to take either a file name, a **java.io.File** object, or a **java.io.Writer** object for maximum flexibility. The same Part21Writer can be used to write multiple models to secondary storage.

The following example demonstrates how to write a Model to secondary storage:

```
/* at the top of your code: */
import com.steptools.stdev.*;
import com.steptools.stdev.p21.*;

/* inside a class: */
Model write_file (Model mod) throws STDevException, IOException {
   Part21Writer writer = new Part21Writer();
   return writer.write("test.stp", mod);
}
```

If you need to the control the name of the schema that gets written in the header of the file, you can subclass **Part21Writer** and override the **getSchema** method.

2.7 Entity Instance Identifiers

Every entity instance in a Part 21 file has an identifier. In the Part 21 file, the identifier is a “#” character followed by a number. ST-Developer includes the **EntityIDTable** to track the identifiers in a **Model**. The **EntityIDTable** is created when a **Model** is read from or written to a Part 21 file, or is can be created dynamically with the **EntityIDTable.forModel** static method. Identifiers in the **EntityIDTable** are of type **java.math.BigInteger**.

To find an **EntityInstance** by identifier, use the **getInstance** method of the **EntityIDTable** instance. The following example shows how to find an instance:

```
void EntityInstance findByID (Model mod, BigInteger id) {

    /* Get the EntityIDTable for the model. This method will get the
     * table associated with the model. This will create the table if it
     * does not already exist.
     */

    EntityIDTable tab = EntityIDTable.forModel(mod);
    return tab.getInstance(id);
}
```

To get the identifier for an **EntityInstance**, use the **getId()** method. Initially, only objects that were read from a Part 21 file or written to one will have IDs. The **getId()** takes a second parameter, which, if true, will assign an identifier to object that do not yet have one; otherwise, the method returns zero.

2.8 STEP Header

The **HEADER** section of a Part 21 file contains a timestamps, author, organization, and related information. In memory, the header is represented as a **Population** of the **header_section_schema**, which is defined in the Part 21 specification. While you can directory access to population, it is more convient to use the **com.steptools.stdev.p21.Header** class.

The Header class includes methods **getFileName()** and **getFileDescription()** which return the **file_name** and **file_description** instances, respecively, from the Part 21 header. If these instances do not exist, they are created. In addition, a nunber of attributes of these entities are declared as lists of strings. These methods insure that these lists exist by creating them is they are set to null.

This example shows how you could add a string to the description of a file:

```
import com.steptools.stdev.p21.*;
import com.steptools.stdev.*;

void addDescription (Model mod, String desc) {

    Header head = Header.forModel(mod);
    return head.getFileName().getFileDescription().getDescription()
       .add(desc);
}
```

The following example returns the “originating system” attribute:

```
String getOrig (Model mod) {

    Header head = Header.forModel(mod);
    return head.getFileName().getOriginating_system();
}
```

3 From EXPRESS to Java Classes

3.1 Overview

This chapter describes how EXPRESS data structures are converted to Java classes and interfaces by the **express2java** compiler. The EXPRESS information modeling language is used to define the data structures in STEP Application Protocols.

The compiler is available as the command line **express2java** tool on all platforms. On Windows, it can also be run through the **EXPRESS/Java Converter** control panel found on the ST-Developer Launcher. The options and outputs are described in the following sections.

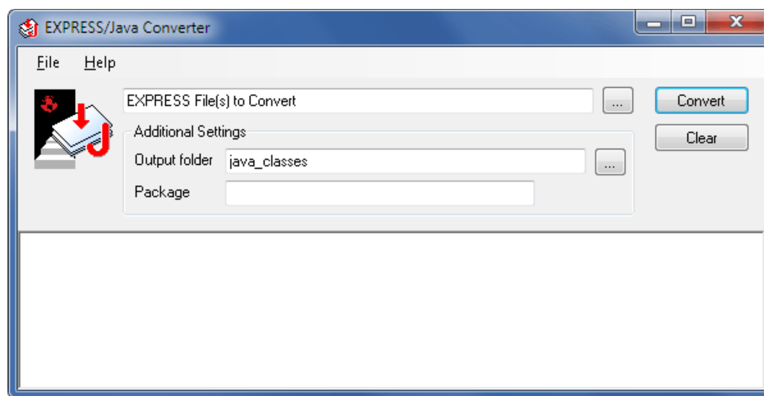


Figure 3.1 — EXPRESS to Java Control Panel

3.2 Command Line

express2java [options] expfile1 [expfile2 ...]

-help Display usage information

-o <path> Set the output location. If not specified, the output will be written to **java_classes**

-forceschema <name> Use a different name for the final package branch. By default, the name of the EXPRESS schema is used.

The EXPRESS to Java compiler reads a text file containing EXPRESS definitions, parses the definitions, and checks them for consistency. If multiple files are given, the tool behaves as if all files were concatenated together into a single file.

3.3 Namespaces

ST-Developer provides several namespaces under **com.steptools** for the Java classes that are provided either by the ST Developer base classes or by the classes generated by the EXPRESS compiler.

The **com.steptools.stdev** namespace contains the basic definitions for the STEP data and provides the base classes to manage the data, and to provide a late-bound interface to the instance data. The classes and interfaces in this interface include **EntityInstance**, **Domain**, and **PopulationBase**.

The **com.steptools.stdev.keystone** namespace provides definitions for EXPRESS primitive types (except **String**, **double** and **int** – which are provided by Java) and for single dimensional aggregates of all EXPRESS primitive types.

The **com.steptools.stdev.p21** namespace contains the classes that read and write Part 21 file and transfer STEP instance data between the Java classes provided by ST-Developer and Part 21 files.

The EXPRESS compiler generates a number of Java definitions for each EXPRESS schema that it processes. All of the definitions for are placed in a namespace of the form **com.steptools.schemas.<schname>**. Where **schname** is the name of the schema as defined in the EXPRESS "SCHEMA" entry. For example, for AP203 the namespace is **com.steptools.schemas.config_control_design** and for AP214, it is **com.steptools.schemas.automotive_design**.

3.4 Primitive Types

The EXPRESS primitive types consist of integer, real, boolean, logical, string and binary. These types are mapped to the corresponding Java type as specified in the table below. All of these types are either provided by the Java environment, or the ST-Developer class library.

EXPRESS	Java
INTEGER	int
REAL	double
STRING	String
BOOLEAN	com.steptools.keystone.ExpBoolean
LOGICAL	com.steptools.keystone.Logical
BINARY	com.steptools.keystone.Binary

As indicated in the table, attributes of type integer, real, and string are mapped to the corresponding Java type. For all the other types, ST-Developer provides a class in the **com.steptools.keystone** namespace to represent the EXPRESS data type.

Although Java has the primitive type **boolean**, is not used for boolean values because it has no way to represent the STEP notion of an “unset” value.

EXPRESS Boolean values are represented by the constants of type **ExpBoolean**: **ExpBoolean.FALSE** and **ExpBoolean.TRUE**. Logical values are represented by constants of type Logical: **Logical.FALSE**, **Logical.TRUE** and **Logical.UNKNOWN**.

ExpBoolean and **Logical** override the **toString()** method which will return the value as a string: either “true”, “false” or “unknown”. There is also a **booleanValue()** method which return the value of a **Logical** or **ExpBoolean** as a Java primitive boolean value. (Following the EXPRESS conventions, **unknown** is converted to **true**.)

3.5 EXPRESS Schemas

For each schema the EXPRESS compiler generates two classes in the schema namespace: one is named **Schema** and one is named **Population**. Theses classes provide the meta-data for the late-bound methods to use, and also include methods for creating entity instances.

Since every schema-specific namespace contains a **Population** and and a **Schema** class; importing the definitions from more than one schema, will cause a name conflict. For example, consider the import statements:

```
import com.steptools.stdev.*;
import com.steptools.schemas.config_control_design.*;
import com.steptools.schemas.automative_design.*
```

There is a conflict for the **Population** and **Schema** symbols, and so you must use a fully qualified class name to reference the schema-specific class, or else you will need to keep the code that imports the definitions from one schema in a different Java source file from code that references a different schema. However, since most applications only deal with a single schema at a time, this should generally not effect your code.

Using the above declarations, you can reference the AP214 schema as follows:

```
com.steptools.schemas.automative_design.Schema
```

3.5.1 Population

The **Population** holds the entity instances in a model. For each data section in a Part 21 file, there will be one population instance to contain the entity instances in the section. The schema-specific **Population** class extends the abstract **com.steptools.stdev.PopulationBase** class.

The **Population** class is used to create persistent STEP entity instances. For every entity in the schema, the EXPRESS compiler generates a method which creates an instance of it. This method is named **new<Entname>** where Entname is the capitalized name of the corresponding EXPRESS entity. Given the EXPRESS schema:

```
SCHEMA test;

ENTITY foo;
   name: STRING;
END_ENTITY;

ENTITY bar;
   length : REAL;
END_ENTITY;

END_SCHEMA;
```

The compiler will generate the following **Population** class. (This is a simplified example. The actual class includes code and additional members for internal use.)

```
package com.steptools.schemas.test;

import com.steptools.stdev.PopulationBase;

public class Population extends PopulationBase {
   Foo newFoo();
   Bar newBar();
}
```

3.5.2 Schema

The **Schema** class represents the EXPRESS schema as a whole to the run-time system. Unlike the **Population** class, there is only one instance of the **Schema** class for an EXPRESS schema. This static instance is stored in the static final field named **SCHEMA** of the **Schema** class. The following code gets the Schema for AP203:

```
import com.steptools.stdev.*;
import com.steptools.schemas.config_control_design.*;

/* then inside some method: */
Schema ap203_sch = Schema.SCHEMA;

/* Do something with the schema */
Model mod = new Model(ap203_sch);
Population pop = (Population) mod.getPopulation();
```

3.6 Entity Types

Each ENTITY type is mapped to a Java interface in the namespace corresponding to the schema. The name of the interface consists of the EXPRESS name with the first letter converted to uppercase, and all of the other letters in lowercase.

The interface for the entity provides a get and a set method for each attribute. Thus, for an attribute named **items**, there are methods named **getItems** and **setItems**. For example, consider the following EXPRESS entity definition:

```
ENTITY action;
   name : label;
   description : text;
   chosen_method : action_method;
END_ENTITY; -- action
```

The EXPRESS compiler generates a Java interface that includes the following

members.

```
public interface Action extends EntityInstance {
public static final StaticEntityDomain DOMAIN /* stuff deleted */
   void setName(String val);
   String getName();

   void setDescription(String val);
   String getDescription();

   void setChosen_method(Action_method val);
   Action_method getChosen_method();
};
```

In the EXPRESS, the label and text types are defined types where the underlying type is **String**. In the Java code, such simple defined types are fully resolved to the underlying primitive type. The get and set methods provided by the interface conform to the naming conventions for JavaBeans classes.

The **final static** field named **DOMAIN** contains the type information for the entity. This can be used in late-bound applications to query the structure of the EXPRESS information model, or it can be used to find all the object of a given type.

Since the entity is mapped to an interface and not to a class, there is no constructor to create an instance. Instead, you must call the **newXXX** method on the **Population**.

3.6.1 Single Inheritance

When an EXPRESS entity has a SUBTYPE OF clause that specifies a single supertype, it is mapped to a Java interface that extends the interface that corresponds to its supertype, rather than the base interface **EntityInstance**. The **calendar_date** entity is defined as follows:

```
ENTITY calendar_date
SUBTYPE OF (date);
   day_component : day_in_month_number;
   month_component : month_in_year_number;
WHERE
   WR1: valid_calendar_date(SELF);
END_ENTITY; -- calendar_date
```

The corresponding Java interface includes the following members:

```
public interface Calendar_date extends Date {
   public static final StaticEntityDomain DOMAIN;

   void setDay_component(int val);
```

```
    int getDay_component();

    void setMonth_component(int val);
    int getMonth_component();
};
```

Since the interface extends the **Date** interface, all of the methods of **Date** are also available in the **Calendar_date** interface.

3.6.2 Multiple Inheritance

When an EXPRESS entity has a SUBTYPE OF clause that specifies two or more supertypes, there are several possible ways to map it to a Java interface.

When possible, the single inhertiance method documented above is used, except that the Java interface will extend all of the interfaces associated with the entity's supertypes. For example: consider the follow entity:

```
ENTITY poly_loop
SUBTYPE OF (loop, geometric_representation_item);
     polygon : LIST [3:?] OF UNIQUE cartesian_point;
END_ENTITY; -- poly_loop
```

This maps to the following Java interface:

```
public interface Poly_loop extends Loop, Geometric_representation_item {
    public static final StaticEntityDomain DOMAIN;
    void setPolygon(ListCartesian_point val);
    ListCartesian_point getPolygon();
};
```

Unfortunately, it is not always possible to map entities with multiple inheritance in this simple manner. Specifically, when different supertypes each declare an attribute with the same name. In this case, there are two EXPRESS attributes that map to the same Java accessor methods, and so an alternative representation must be used. The following EXPRESS definitions have such a conflict:

```
ENTITY foo;
    conflict: INTEGER;
    f_att: STRING;
END_ENTITY;

ENTITY bar;
    conflict: INTEGER;
    b_att: STRING;
END_ENTITY;
```

```
ENTITY sub
SUBTYPE OF (foo,bar);
   size : INTEGER;
END_ENTITY;
```

In this example, **sub** inherits the attribute named **conflict** from both **foo** and **bar**. It is not possible to generate a corresponding Java interface using the above rules, so this is treated as a special case. The interfaces for the supertypes are generated using the usual rules:

```
public interface Foo extends EntityInstance {
   void setConflict(int val);
   int getConflict();

   void setF_att(String val);
   String getF_att();
};

public interface Bar extends EntityInstance {
   void setConflict(int val);
   int getConflict();

   void setB_att(String val);
   String getB_att();
};
```

For the subtype, the following interface is generated:

```
public interface Sub extends EntityInstance {
   void setFooConflict(int val);
   int getFooConflict();

   void setBarConflict(int val);
   int getBarConflict();

   void setF_att(String val);
   String getF_att();

   void setB_att(String val);
   String getB_att();

   Foo asFoo();
   Bar asBar()
};
```

For attributes with name clashes, like the **conflict** example, a qualified accessor is provided. The name of a qualified accessor consists of **get** or **put** followed by the name of the entity that defined the attribute (before any redeclaration), followed by the capitalized schema name, followed by the capitalized attribute name. The other attributes (**foo_att** and **bar_att** in this example are left unqualified.

There is still the problem of calling a method that takes an argument of type **Foo** when you have an instance of type **Sub**. Normally, Java would cast the value to the supertype as requested, but in this case, there is no "extends" relationship between the interfaces. A type cast method is generated for each EXPRESS supertype which does not appear in the "extends" clause. This cast method is named after the target class, with "as" prepended. The cast method returns a view of the underlying object, so any changes made are immediately reflected in the original object.

3.6.3 Complex Instances

Complex entity combinations (sometimes called "AND/OR" instances) are instances with a set of types rather than a single type. This is just multiple inheritance, but the EXPRESS schema does not define an ENTITY for the combination.

ST-Developer Java libraries do not have specific classes or interfaces for the combinations. Instead, they are handled through the late-bound interface provided by the **EntityInstance** interface.

To create a complex instance, call the **newInstance** method of a **Population**, specifying all the supertypes of the complex instance. The following example contains a method that constructs a complex instance.

```
/* These import are assumed to be in effect
import com.steptools.schemas.config_control_design;
import com.steptools.stdev.*;
import java.util.List;
*/

public EntityInstance create_millimeter_unit (Population pop) {
    /* Must be a SI_UNIT and LENGTH_UNIT combination */
    EntityDomain[] supers = {Length_unit.DOMAIN, Si_unit.DOMAIN};
    EntityInstance ret = pop.newInstance(supers);
    return ret;
}
```

To work with data defined by one of the supertypes of a complex instance, use the **castTo** method of **EntityInstance**. The **castTo** method is declared to return **EntityInstance**, so you must also use a Java cast to convert to object to the requested subtype. The following example builds upon the previous one to

```
public EntityInstance create_millimeter_unit (Population pop) {
    /* Must be a SI_UNIT and LENGTH_UNIT combination */
    EntityDomain[] supers = {Length_unit.DOMAIN, Si_unit.DOMAIN};
    EntityInstance mm = pop.newInstance(supers);

    /* Set the SI_UNIT attributes */
```

```
        Si_unit mm_as_si_unit = (Si_unit) mm.castTo(Si_unit.DOMAIN);
        mm_as_si_unit.setPrefix (Si_prefix.MILLI);
        mm_as_si_unit.setName (Si_unit_name.METRE);

        return mm;
    }
```

3.6.4 Views

Due to differences between the Java and EXPRESS languages, the binding provides view objects when necessary. This has already been demonstrated above with the **asSuper** methods in some multiple inheritance cases, and also in the case of complex instances. A view object can be used interchangeably with its underlying instance. Changes to the view are instantly reflected in the underlying object. The **castTo** method can be applied to any view of an instance to retrieve any other view of the instance. The **isa** method, likewise, returns the same value regardless of which views of an instance it is called on.

You must use the **castTo** method before applying a Java cast whenever there is a possibility that a view is involved. The best rule of thumb is to use **castTo** any case where you are casting an **EntityInstance** down the inheritance hierarchy. If you do not use a **castTo** where you need to, the system throws a **ClassCastException**.

3.7 Enumeration Types

EXPRESS enumeration types in are represented using the type-safe “enum” design pattern. For each enumeration, the EXPRESS compiler generates a class which contains a **static final** field for each item in the enumeration. For example, the following EXPRESS definition:

```
TYPE ahead_or_behind = ENUMERATION OF (ahead, behind);
END_TYPE; -- ahead_or_behind
```

Results in the following Java class. (this is a simplified version of the class):

```
public class Ahead_or_behind extends Enumeration {
   public static final Ahead_or_behind AHEAD;
   public static final Ahead_or_behind BEHIND;
}
```

An attribute of this type can be set as follows following code:

```
offset.setSense(Ahead_or_behind.BEHIND);
```

Also note that due to the type-safe enum pattern used here, there is only one instance of **AHEAD** and **BEHIND**, and so it is always safe to compare any instance of **Ahead_or_behind** to the constants **Ahead_or_behind.AHEAD**, **Ahead_or_behind.BEHIND** or any other instance of **Ahead_or_behind**.

3.8 Select Types

An EXPRESS select type represents a union of several a set of underlying types, known as selections. In Java, a select type is mapped to a subclass of **com.steptools.stdev.Selection**. For each selection, the class contains a get method and a query (is) method. The get method consists of the keyword “get” prepended to the name of the desired type. This will return the value in the select, or throw **SelectTypeException** if the underlying value is of a different type.

The select class includes a nested class member named **Selection**. This class serves as an enumeration which identifies the type of the select. The select class include one instance of **Selection** for each selection type defined in the EXPRESS. The Selection of a select instance can be determined by call the **selection()** method.

Consider the following EXPRESS definition.

```
TYPE axis2_placement = SELECT (axis2_placement_2d, axis2_placement_3d);
END_TYPE; -- axis2_placement
```

This will result in the following class. (the actual class includes additional members that have been omitted for clarity.):

```
public abstract class Axis2_placement extends Select {
   public static final class Selection extends SelectionBase;

   public static final Selection SELAxis2_placement_2d;
   public static final Selection SELAxis2_placement_3d;

   public SelectionBase selection();

   /* Accessor methods */
   public Axis2_placement_2d getAxis2_placement_2d()
   throws SelectTypeExceptionl

   public Axis2_placement_3d getAxis2_placement_3d()
   throws SelectTypeException

   /* Type query methods */
```

```
    public boolean isAxis2_placement_2d();
    public boolean isAxis2_placement_3d();
}
```

Using this example, we can process an instance of **Axis2_placement** with the following method.

```
void process_2d_placement(Axis2_placement_2d pl);
void process_3d_placement(Axis2_placement_3d pl);

void process(Axis2_Placement pl, PrintStream out) {
    if (pl.isAxis2Placement_2d()) {
        out.println ("Have 2D placement");
        process_2d_placement (pl.getAxis2_placement_3d());
    }
    else if (pl.isAxis2Placement_3d()) {
        out.println("Have 3D placement");
        process_3d_placement (pl.getAxis2_placement_3d());
    }
}
```

3.8.1 Creating Select Instances

A select instance is an immutable object. Once created, the value of a select cannot be changed. An entity attribute holding a select is changed by creating and assigning a new select instance.

The EXPRESS compiler generates a method in both the **Schema** and **Population** classes to initialize each select in the schema for each possible type. These methods are named “new” followed by the name of the select type, any intermediate types and, if the underlying is a primitive, the name of the selection. Given the following EXPRESS definition:

```
TYPE measure_value = SELECT
 (area_measure,
  context_dependent_measure,
  count_measure);
END_TYPE; -- measure_value
```

The **Schema** and **Population** classes contain the following methods:

```
public static Measure_value
    newMeasure_valueArea_measure (double v);

public static Measure_value
    newMeasure_valueContext_dependent_measure (double v);

public static Measure_value
```

```
newMeasure_valueCount_measure(double v);
```

Using these methods, you can create a **Measure_value** as follows:

```
void Measure_value create (Population pop) {
   Measure_value val = pop.newMeasure_valueAreaMeasure(5.76);
}
```

3.8.2 Selection Naming Conventions

The elements of EXPRESS select types must be entity or named defined types. The name of the methods are determined by the names of the entities or defined types in the information model. Consider the following EXPRESS type:

```
TYPE measure_value = SELECT
 (area_measure,
  context_dependent_measure,
  count_measure);
END_TYPE; -- measure_value
```

This results in a class that includes the following members:

```
public class Measure_value extends Select {
    public static final class Selection extends SelectionBase {
    }
    public static final Selection SELArea_measure;
    public static final Selection SELContext_dependent_measure;
    public static final Selection SELCount_measure;

    public double getArea_measure();
    public double getContext_dependent_measure();
    public double getCount_measure();

    public boolean isArea_measure();
    public boolean isContext_dependent_measure();
    public boolean isCount_measure();
}
```

Note that the accessor, and query methods, and names of the selection enumerators are based in the name of the EXPRESS defined types (e.g. **area_measure**), and not the underlying primitive types (e.g. real). The return types of those attributes are the primitive types, however.

3.8.3 Nested Selects

In an EXPRESS a select type may contain selections which are themselves select types. In this case, all of the selections of the included select types are incorporated into the class for the outermost select type as if they were declared in the outer select type. If there are any duplicate selections, only a single version of the duplicate selection produced. This behavior is consistent with the EXPRESS semantics as interpreted in the Part 21 specification.

For example, consider the following EXPRESS definitions:

```
TYPE characterized_definition = SELECT
 (characterized_product_definition,
  shape_definition);
END_TYPE; -- characterized_definition

TYPE characterized_product_definition = SELECT
 (product_definition,
  product_definition_relationship);
END_TYPE; -- characterized_product_definition

TYPE shape_definition = SELECT
 (product_definition_shape,
  shape_aspect,
  shape_aspect_relationship);
END_TYPE; -- shape_definition
```

The **characterized_definition** select type includes two other select types within it. This class for the EXPRESS compiler generates for this type is equivalent to what would be generated if the select members were expanded as follows:

```
TYPE characterized_definition = SELECT
 (product_definition,
  product_definition_relationship,
  product_definition_shape,
  shape_aspect,
  shape_aspect_relationship);
END_TYPE; -- characterized_definition
```

Note that the **characterized_product_definition** and **shape_definition** selections do not appear in the combined select. The nested example above will produce the following “new” methods in the the **Schema** and **Population** classes.

```
public static Characterized_definition
   newCharacterized_definition (Product_definition v);

public static Characterized_definition
   newCharacterized_definition (Product_definition_relationship v);
```

```
public static Characterized_definition
   newCharacterized_definition (Product_definition_shape v);

public static Characterized_definition
   newCharacterized_definition (Shape_aspect v);

public static Characterized_definition
   newCharacterized_definition (Shape_aspect_relationship v);
```

EXPRESS also allows a second form of nested select type. In this case, the nested select is enclosed by a simple defined type. Note that this usage doe not occur in any of the STEP integrated resources. For example:

```
TYPE outer = SELECT
 (nested,
  length);
END_TYPE;

TYPE inner = SELECT
 (length,
  color);
END_TYPE;

TYPE length = REAL;
END_TYPE;

TYPE nested = inner;
END_TYPE;
```

In **outer**, the **nested** element is a defined type whose underlying type is a select. In STEP, this is considered a distinct type, and so **inner** cannot be folded into **outer**, as was shown in the previous example. A value of type **outer** can be:

```
nested.length
nested.color
length
```

For the compound types, the name of the Java definition is obtained by capitalizing the first letter of each type, and concatenating the names together. This results in the following class for **outer**.

```
public class Outer extends Select {
    public static final class Selection extends SelectionBase {
    }

    public static final Selection SELNestedLength;
    public static final Selection SELNestedColor;
    public static final Selection SELLength;

    public double getNestedLength();
    public Color getNestedColor();
```

```
        public double getLength();

        public boolean isNestedLength();
        public boolean isNestedColor();
        public boolean isLength();
    }
```

This example will produce the following "new" methods in the the **Schema** and **Population** classes:

```
public static Outer newOuterNestedLength(double v);
public static Outer newOuterNested(Color v);
public static Outer newOuterLength(double v);
```

3.9 Aggregation Types

Aggregation types in EXPRESS represent collections of values. ST-Developer provides some predefined aggregates in the **com.steptools.stdev.keystone** namespace. All other aggregates are generated by the EXPRESS compiler in the namespace of the schema that uses the aggregate. The built-in aggregates consist of list, bag, set and arrays of boolean, integer, logical, real and strings.

The aggregate classes (both built-in and generated) implement the **java.util.List** interface. This means that STEP aggregates can be search, sorted and processed using the utility methods provided by the Java Collections Framework.

The class names concatenate the aggregate type (e.g. **List**, **Bag**, **Set** or **Array**) with the name of the underlying element. Thus a **List of Foo** is mapped to a class named **ListFoo**.

Since aggregates implement the Java Collections **List** interface, each aggregate class provides the following methods: **add**, **addAll**, **addAll**, **clear**, **contains**, **containsAll**, **equals**, **get**, **hashCode**, **indexOf**, **isEmpty**, **iterator**, **lastIndexOf**, **listIterator**, **remove**, **removeAll**, **retainAll**, **set**, **size**, **subList**, **toArray**. These methods are loosely typed, so they throw a **ClassCastException** if an attempt is made to insert an object that does not belong in the aggregate.

For aggregates of double and integer, you must use the Java wrapper classes **Integer** or **Double** when you put a value using the loosely typed interface. Likewise, the **get()** method is declared to return an **Object** so doubles and integers will be returned in a wrapper object. (All other possible types are already subtypes of **Object**, so they merely need to be cast to the desired type.)

The aggregate class also provides strongly-typed version of the **add**, **get** and **set** methods. The **add** and **set** methods are overloaded to take a parameter of the element type. The strongly-typed **get** method is named **getValue()**.

Consider the following EXPRESS:

```
TYPE length_measure = REAL;
END_TYPE; -- length_measure

ENTITY cartesian_point
   coordinates : LIST [1:3] OF length_measure;
END_ENTITY; -- cartesian_point
```

The following example creates and populates a **cartesian_point** instance containing a list of doubles.

```
Cartesian_point make_point(Population pop, double x, double y, double z) {
    Cartesian_point ret = pop.newCartesian_point();
    ListReal coords = new ListReal();
    coords.add(x);
    coords.add(y);
    coords.add(z);
    ret.setCoordinates(coords);
    return ret;
}
```