PSL: Portfolio Simulation Language

Alexander Besidski
ab2012@columbia.edu

Jian Huang
jh2353@columbia.edu

Xin Li
x174@columbia.edu

Wei-Chen Lee
wl2135@columbia.edu

December 18, 2004

1 Introduction

1.1 Background

Over the last few decades, the average person's interest in the stock market has grown exponentially. What was once a toy of the rich has now turned into the vehicle of choice for growing wealth. This demand coupled with advances in trading technology has opened up the markets so that nowadays nearly anybody can own stocks. While there exist various tools designed for creating complex financial models based on market data, these tools are all geared towards sophisticated business users. Portfolio simulation language is designed specifically for the purpose of simulating and modeling behaviors of financial assets in a way accessible to inexperienced users having knowledge of only a few fundamental concepts. In addition to being easily accessible the language is also designed to be flexible for the more advanced investors. As a simple programming language, it is particularly suitable as a flexible and convenient teaching tool for demonstrating and practicing basic financial market operations for students majoring in finance.

1.2 Fundamentals

Two concepts will lie at the heart of our language portfolio and asset. Portfolios are essentially collections of assets with a few additional attributes. PSL will define constructs to manipulate portfolio content and define functions to operate on them. Two examples of assets are stocks and bonds. At the very minimum all assets share two common characteristics rate of return and variance. Having these two attributes at the core of every asset will allow us to create portfolios comprised of different types of assets and to run meaningful analysis on their content, which is the goal of this project. By making financial instruments into first class citizens of our language we will obtain the ability to define portfolios with ease and run various types of operations with a few lines of code.

1.3 Features

1.3.1 Simplicity

Since we would like to create a language that is easy enough for a user who is inexperienced in both programming and finance the language will have only a few basic types in addition to assets and portfolios primarily added in order to manipulate assets and create custom functions.

1.3.2 Hierarchical

Being an intrinsically object-oriented programming language, PSL adopts a hierarchy of data types. Each of these data is customized by different properties (data members) and behaviors (member functions), while all of them sharing certain set of common characteristics. Basically, three hierarchical levels of abstraction exists:

a. primitive data types

These basically include numbers, boolean, strings, and constants, to support basic arithmetic and logical operations, which is indispensable for implementing any full-edged programming languages.

b. financial instruments (assets)

These include [stock], [bond] and [cash]. Here [cash] is modeled as a risk-less financial instrument having a fixed rate of return. [bond] is similar to [cash] except that rather than simply compounded, it has a certain maturity date and a fixed coupon yield. At maturation [bond] is automatically converted to [cash]. [stock] has a property named volatility to measure its risk, as well as an expected rate of return. The stochastic behavior of [stock] can be modeled by lognormal distribution which is implemented in the member functions of the [stock] data type.

c. financial derivatives

These are basically more complicated financial instruments derived from the basic financial instruments in (b), including [option] and [future], which must have an underlying asset (financial instrument) in order to be fairly priced. Due to the intrinsic stochastic nature of the underlying assets (stocks), simulation and modeling operations are also supported for financial derivatives.

d. portfolio

This is the highest level data type in PSL, and lies at the core of the language. Basically [portfolio] is a composite data type, consisting of arbitrary combinations of (b) and (c) (satisfying certain constraints). A [portfolio] data type supports many operations include risk/return optimization and VaR analysis, as well as Monte Carlo simulation. In addition, the array type, which stores a fixed number of homogeneous data types, is supported for all the basic data types in (a)-(d).

1.3.3 Flexibility

PSL aims to cater to make the language accessible and useful for users with various levels of expertise in programming experience and financial knowledge. It is designed such that users with the minimum knowledge of computer programming would quickly master the basic language elements and be able to write practical programs in PSL. Although internally designed as totally objected-oriented, PSL provides users with a more traditional procedure based programming interface. One could simply write a script-like program, using all the normal features supported by most other languages such as loops, conditional branches, etc, without any need to know anything about objected-oriented programming concepts. Users will be able to construct complicated portfolios and perform highly professional analysis, using the built-in financial instrument types such as [stock], [bond], [future], etc. For most common practices in finance these should be enough. For more advanced users, e.g., traders and financial analysts who may want to manipulate more complicated financial instruments in the portfolio, PSL provides them with the options to build up their own financial derivatives by expanding the basic financial instrument types. This would require the users to know more about the advanced features of PSL, particularly the objected oriented concepts, since the user have to define their new object types and specify their properties (data members) and behaviors (member functions). However, this is by no way a requisite to master the basic programming in PSL.

1.3.4 Finance-oriented

PSL is mainly designed to perform financial analysis and portfolio management/simulation. So manipulation of finance instruments and their compositions lies at the heart of the language. PSL provides many built-in financial analysis tool, imbedded in the basic financial instrument types as built-in functions. For example, PSL provides tools for portfolio risk/return optimizations, VaR analysis, historical data extraction, stock movement modeling, option/future pricing, and Monte Carlo simulation of stocks/portfolio paths, to name a few. All these powerful operations provide simple and user-friendly interfaces to the users and all the mathematical details are hidden. Users with little or no mathematical/financial background would still be able to write complicated financial analysis software using our PSL language.

2 Tutorial

2.1 Overview

PSL program consist of parts: (1) financial instrument name and related value definition. (2) covariance between the two portfolios. (3) leverage Necessity. (4) portfolio simulation. User can define various financial instrument (stock and bond)'s name and related value, for example, stock IBM = new (Price=50.0), IBM.Return = 0.05, etc. Covariance can be viewed under user-define financial instrument and their covariance arguments. Leverage Necessity can remind users if their portfolio operation bond is over user-define initial capital quota. Portfolio simulation can predict the profit of user-define financial instrument combination after a certain time (ex: ten year).

2.2 Financial Instrument Name and Related Value Definition

To declare a financial instrument, say stock, and we can initialize its related value at beginning.

```
stock google = new (Price=150.0);
```

or initialize its value later

```
stock google
google.Price = 150.0
```

To clarify variable google, we need to give it a name

```
google.Name="GOOGLE"
```

We can also specify other attributions (Price, Return, Volatility and Dividend) (if necessary)

```
google.Return = 5;
google.Volatility = 30;
```

Then Print out its price

```
google.Price.print();
```

For the other financial instrument, bond, we can do the same thing with no doubt.

```
bond Treasury10Y = new(Coupon=3.0, Maturity=10, InterestRate=5.0);
Treasury10Y.Name = "Treasury10Y";
```

2.3 Covariance between the Two Portfolios

We have a build-in function can calculate covariance between two different financial instruments.

Portfolio pf;

```
pf.setCov("google", "IBM", 0.1);
real cv = pf.getCov("google", "IBM");
```

The variable cv represents the covariance between stock google and IBM.

2.4 Leverage Necessity

During the process that users define and add financial instruments into the portfolio combination anytime, and we have a build-in function which can help users check if their current financial operation is leverage or not.

```
if ( pf.isLeverage() ) then
{
      (“Portfolio is leveraging.”).print();
}
else
{
        (“Portfolio is NOT leveraging.”).print();
}
```

2.5 Portfolio Simulation

Inside the .psl program, users can add their new declared financial instruments and change their related-values or portfolio capital value anytime before simulation starting.

```
portfolio pf = new(Capital=3000.0);
pf.addStock(google, 20);
pf.addBond(Treasury10Y, 5);
pf.addStock(IBM, 8);
pf.addStock(google, 16);
pf.addCapital(1000.0);
```

When everything is ready, we can show the final total asset value and charts helping users understand the trend in a certain of time, say years.

```
real final_value = pf.getTotalAsset();
pf.simulate(20);
```

User can load history financial database related to user-define financial instrument for future price variation simulation

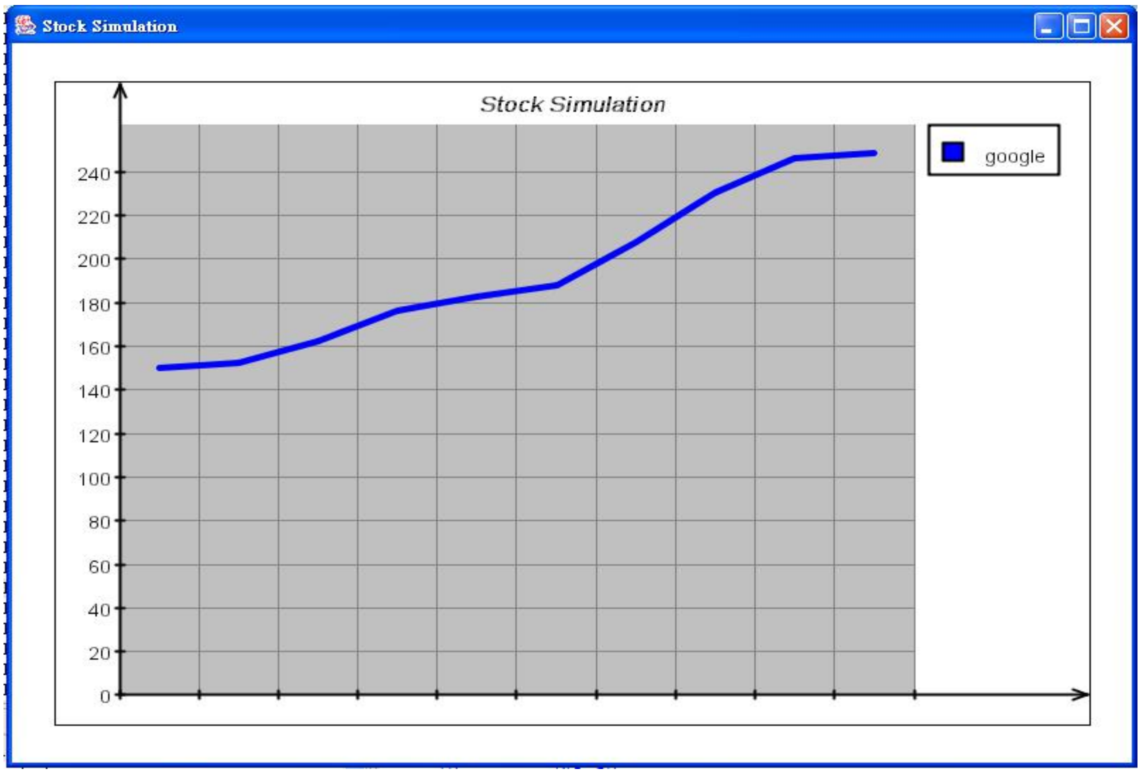
```
real bref[];
bref = loadData (“test/msft.txt”);
```

We define expected future price for financial instrument.

```
// The value p represents stock google’s value after 100 days.
real p = google.getExpectedFuturePrice(100.0);
```

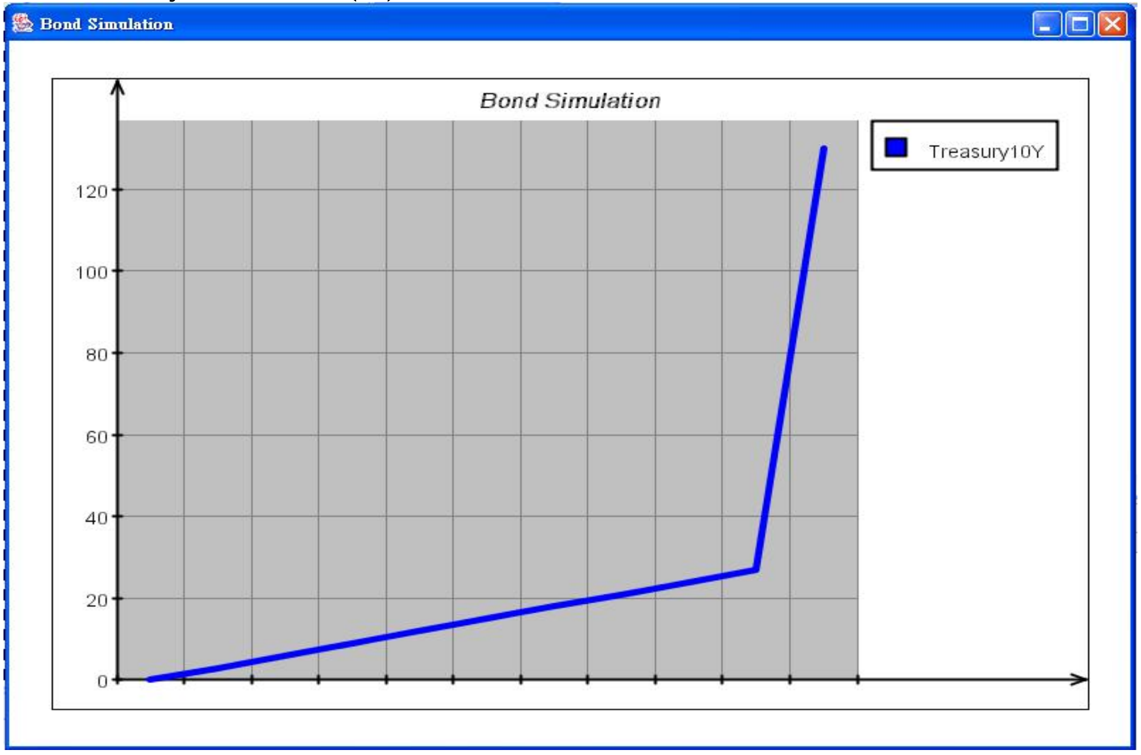
Simulate stock google chart

```
// simulation
google.simulate(10);
```



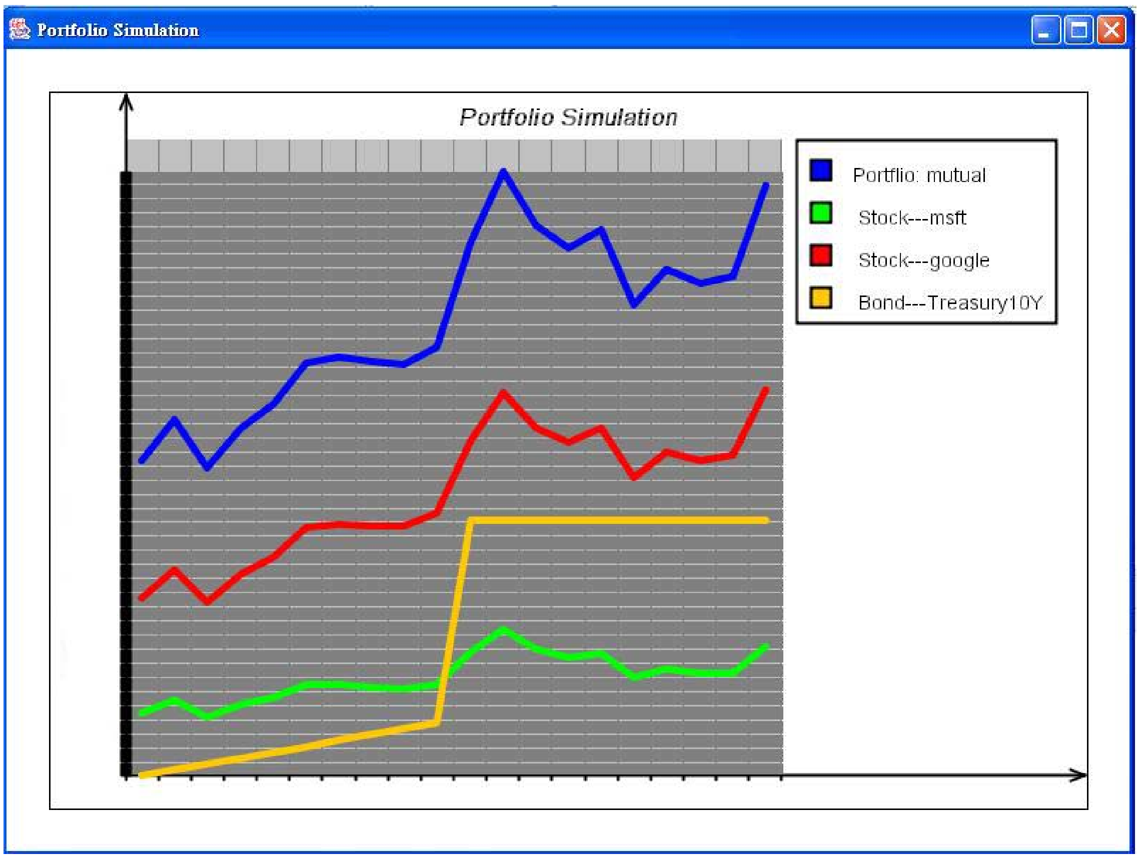
Simulate bond Treasury10Y chart

```
// simulation
Treasury10Y.simulate(11);
```



Simulate portfolio combination chart

```
pf.simulate(20);
```



We can also save the simulation result in a text file

```
real simref[];
simref = google.simulae(10);
saveData (simref,”test/google.txt”);
```

2.6 Environment for Compiling and Running PSL Program

We choose Eclipse Platform as our demonstration environment. To execute PSL program, inside **Run** dialog, **(x)=Arguments** option, users can add their .psl file location in **Program arguments** field, then press Run button to compile and run the program.

3 Language Reference Manual

3.1 Lexical Conventions

PSL defines 4 types of tokens – identifiers, keywords, constants, expression operators and separators.

3.1.1 Comments

Single line C-style // comments

3.1.2 Identifiers

Identifier is a sequence of letters and digits where the first character must be either a character or an underscore. PSL is a case-sensitive language so that identifiers with different casing are distinguished.

3.1.3 Keywords

The following is a list of all keywords supported by PSL.

for	if	then	else	break
string	continue	function	true	false
return	while	int	real	new
stock	bond	portfolio	boolean	

3.1.4 Constants

PSL supports 4 types of constants – Boolean, string, integer and real.

3.1.4.1 String

A string is a sequence of characters surrounded by double quotes (“”). In order to express double quotes within a string they need to be escaped via the backslash character. (‘\”’)

3.1.4.2 Integer

An integer constant is simply a sequence of digits. Integer values are stored as 32 bit signed numbers.

3.1.4.3 Real

Real numbers are stored as 64 bit fixed precision values and consist of an integer part followed by a decimal point, a fraction part and an optionally signed integer exponent.

3.1.4.4 Boolean

These constant can only take on two values “true” or “false”.

3.2 Expressions

PSL has 7 expression precedence levels that are explained in the sections below in the order of decreasing priority.

3.2.1 Primary Expressions

3.2.1.1 All four types of constants

Integer, Real, String, Boolean

3.2.1.2 Identifiers
All identifiers must be properly declared before being used.

3.2.1.3 (*expression*)
Any parenthesized expression is considered to be a primary expression that maintains all of the attributes of the un-parenthesized version.

3.2.1.4 *primary-expression* [*expression*]
This notation is only applicable to arrays with the type of the expression being an integer.
PSL will only support one-dimensional arrays.

3.2.1.5 *identifier* (*expression-list* $_{opt}$)
This notation is applicable to function calls where primary-expression. PSL allows recursive function calls. Simple types are passed in by value while complex types such as portfolio and stock and passed in by reference.

3.2.1.6 *identifier*. *method-name* (*expression-list* $_{opt}$)
This notation applies to complex types such as stock and bond. It is used to modify and read their underlying data. The dot operator is left associative.
(yahoo.getExpectedFuturePrice())

3.2.1.7 *identifier.property-name*
This notation applies to complex types such as stock and bond. It is used to access properties in a Style similar to that of C# programming language. In PSL all properties can be read and set. (yahoo.price)

3.2.2 Unary expressions.
Unary operators are right associative
3.2.2.1 - *expression*
Negative of an expression.

3.2.3 Multiplicative operators
These types of operators are left associative.
3.2.3.1 *expression* * *expression, expression* / *expression*
The multiplication operators are valid for real and integer data types. If a real is multiplied by an integer the resulting value is a real.

3.2.4 Additive operators
These types of operators are left associative
3.2.4.1 *expression* + *expression*
The + operator is valid for all types except for boolean. If either expression is a string then both expressions must be strings and the resulting value is a string. Otherwise the result is converted to a real if at least one expression is a real and to an integer if bother operands are integers.

3.2.4.2 *expression* - *expression*
Same semantics as for the + operator with the exception of strings not being allowed.

3.2.5 Relational operators

3.2.5.1 *expression* >= *expression*
3.2.5.2 *expression* > *expression*
3.2.5.3 *expression* < *expression*
3.2.5.4 *expression* <= *expression*
These operators yield a boolean true if the specified expression is true and false otherwise.

3.2.6 Equality operators
3.2.6.1 *expression* == *expression*
3.2.6.2 *expression* != *expression*
These operators are right associative and have boolean return values.

3.2.7 Logical operators
3.2.7.1 *expression* && *expression*
3.2.7.2 *expression* || *expression*
These operators have the semantics that are similar to C and Java. Both are left associative and require both operands to be of type boolean.

3.2.8 Assignment operator
3.2.8.1 *lvalue* = *expression*
In PSL the assignment operator is not recursive unlike its C and Java counterparts. The operator is applicable to all PSL data types and arrays. ***lvalue*** concept is analogous to that of C.

3.3 Declarations
Declarations are used within function definitions and the main body of a PSL program. Their purpose is to provide the compiler with information about storage and behaviors of the identifiers. A declaration consists of a type specifier followed by a declarator.

3.3.1 Type specifiers
PSL allows the following type specifiers:
int, real, string, boolean, portfolio, stock, bond, cash

3.3.2 Declarators
Two forms or declarators are permitted.
identifier – declares a single reference to a type object.
identifier[*expression*] – declares an array of specified size of for the type.
Expression must evaluate to an integer.

3.4 Statements
A PSL program is composed of a series of statements that are executed sequentially.

3.4.1 Expression statements
This is the simplest type of all statements and has the form shown below. Expressions are usually either assignment statements of function calls.

expression **;**

3.4.2 Compound statements

It is often necessary to treat a several statements as one. For that purpose statements can be grouped together using the C-style bracket notation. Opening bracket of a compound statement creates a new scope while the closing bracket destroys it. Thus, a compound statement, which contains a variable declaration, could be executed within a loop.

```
{
        expression₁;
        expression₂;
              …
}
```

3.4.3 Conditional statements

These statements should be used when program flow needs to be changed based on the value of a boolean expression.
Two syntactic alternatives are possible.

if* (*expression*) then *statement

***if* (*expression*) then *statement*$_1$ else *statement*$_2$**

Else ambiguity is resolved by associating the else with the last encountered if.

3.4.4 While statement

C-style loop where expression must evaluate to a boolean.

```
while ( expression)
statement
```

3.4.5 For loop

C-style loop where *expression*$_2$ must evaluate to a boolean.
expression$_1$ is evaluated upon entering the loop for the first time.
The loop is executed while *expression*$_2$ is true.
expression$_3$ is evaluated after every successful loop iteration.

```
for (expression₁; expression₂; expression₃)
   statement
```

3.4.6 Break statement

Causes termination of the inner most while or for loop.

```
while ( expression )
{
            …
            break;
            …
      }
```

3.4.7 Continue statement

Causes termination of the current while or for loop iteration and passes control onto the next iteration.

```
for (expression₁; expression₂; expression₃)
{
      …
      continue;
      …
 }
```

3.4.8 Return statement

Causes the function to return to the caller possibly with a return value.

Examples below illustrate two possible scenarios.

return;
return *expression*;

3.5 Scope Rules

PSL defines scope rules similar to those of Java or C++. A variable can only be used for the duration of its inner most enclosing brackets. Opening bracket of a compound statement creates a new scope while the closing bracket destroys it. It is illegal to declare a variable within a new scope if its name is already used by a variable in any of the outer scope. If a variable is declared within a function it cannot be referenced from any other function or the main body of the program. Since PSL programs consist of only a single file these scope definitions suffice.

3.6 Reference & Value type semantics

PSL data types can be broken up into value types and reference types. The assignment to a variable of a value type creates a copy of the assigned value, while the assignment to a variable of a reference type creates a copy of the reference but not of the referenced object. The following table lists PSL types and their corresponding assignment semantics.

Type	Primitive/Complex	Assignment by
int	Primitive	Value
boolean	Primitive	Value
real	Primitive	Value
string	Primitive	Value
array	Complex	Reference
portfolio	Complex	Reference
stock	Complex	Reference
bond	Complex	Reference

Value types can be used without being initialized, whereas reference types require initialization prior to use.

3.7 Arrays

PSL allows creation of one-dimensional arrays. Arrays can either be declared to be of certain size or can be left with unassigned size and later set to reference an already created array.

Thus the following code would produce an array with undetermined size:

int a[];

Accessing elements of this array would result in error. The following code would produce an array of fixed size that is ready for use.

int a[2];

When an array is declared to store value types, each cell is automatically populated with correct default value. If, on the other hand, an array is created to store reference types accessing individual elements will result in error since reference types must be assigned prior to being used.

3.8 Complex Types

PSL supports a set of built-in complex types and provides a user-friendly object-oriented interface for users to manipulate them in a convenient way. The interface for accessing each built-in type is similar, and users can manipulate the target financial instruments by calling the member functions and properties. The current version of PSL supports three kinds of complex types: Stock, Bond, and Portfolio, which incorporate the three most common kinds of instruments in the financial market. In general, stock is the most common type of equity securities and bond is the most common type of fix-income securities.

3.8.1 Stock

User creates a Stock object in the following way:

stock <ID> = new ([member initialization list]);

Where <member initialization list> composed one or more member initializations, separated by ",", each member initialization takes the form:

<member initialization> : <ID>=<Value>

Where the <ID> to the left of the "=>"denotes one particular property of the Stock object. The possible properties of a Stock object includes

Name <String> [compulsory, read only once defined]

Price <String> [compulsory, must be positive]

Return <Real> [optional, default to the constant NTEREST__RATE,

must be between 0 and 1]

Volatility <Real> [optional, default to the constant

DEFAULT_VOLATILITY]

Dividend <Real> [optional, default to 0.0]

A simple example of a Stock definition is

stock S = new (Name="IBM", Price=24.5, Return=0.30, Volatility=0.75, Dividend=0.20);

Which defines a Stock object called S0 that intends to characterize a real stock share issued by IBM and priced at $24.5, with an expected return rate of 30% and volatility of 75%. In addition, the stock comes with a yearly dividend rate of 20%.

The order of member initializations within the member initialization list is not relevant. For the above example, the definition could be well rewritten as

stock S = new (Price=24.5, Dividend=0.20, Return=0.3, Volatility=0.75, Name="IBM");

Or any other order desired.

Alternatively, the volatility may be calculated from a time series of historical quotas, provided by the user in the form of an array of real numbers, as illustrated by the following example:

```
Real a[10];
// Fill up the elements of a[10]
….
S.setVolatilityFromTimeSeries(a, 10);
```

The available member functions and attributes for the Stock objects include:

```
// Get/Set the initial quoted price of the stock at time 0
Real Price;

// Get/Set volatility of the stock
Real    Volatility;

// Get/Set return of the stock
Real Return;

// Get/Set dividend of the stock
Real    Dividend;

// Get/Set name of the stock
String Name;

// Get the expected future price at time t(measured in years)
Real    getExpectedFuturePrice(real t);

// Calculate the volatility of the stock    from a time series, or an array, of
//historical quotas
  Void setVolatilityFromTimeSeries (real a[], int N);

// Simulate a particular path of the stock movement The Stock will follow
the //lognormal stochastic process. The plot parameter indicates whether
a //graphic output is desired or not.
  Void Simulation(Real timestep, Int nstep, boolean plot);
```

An important property of stocks, which makes the portfolio optimization an interesting and non-trivial practice, is the correlation of stocks. Basically, this means that the stocks are correlated in some way or the other. The correlation between two different kinds of stocks is quantified statistically by the covariance, or correlation coefficient. PSL provides a global function for specifying the correlation coefficient between two (defined) stock objects:

```
Void SetCov(Stock s1, Stock s2, real corr);
```

Where corr must be a real number between –1 and 1.

One can also get the correlation coefficient between two stock objects through another global function:

Real getCov(Stock s1, Stock s2);

By default, all stocks are mutually uncorrelated (correlation coefficient is 0).

3.8.2 Bond

User creates a Bond object in pretty much the same way as the Stock object, i.e.:

Bond <ID> = Bond([member initialization list]);

Where again <member initialization list> composed one or more member initializations, separated by ",", each member initialization takes the form:

<member initialization> : <ID>=<Value>

Where the <ID> to the left of the "=>"denotes one particular property of the Bond object. The possible properties of a Stock object includes

Name <String> [compulsory, read only once defined]

Price <String> [compulsory, must be positive]

Yield <Real> [optional, must be between 0 and 1]

Maturity <Real> [optional, must be positive]

Coupon<Real> [optional, must be positive]

Which gives the name, market price, yield, maturity and coupon of the bond (we assume coupon to be just a single cash flow along with the face value paid at the end of maturity). It is important that either Yield, or Maturity, but not both, must be specified explicitly in the member initialization list (as they are functionally related).

A simple example of a Bond definition is

Bond b = Bond(Name=>"Gov", Price=>97.5, Yield=>0.30, Dividend=>0.20);

Which defines a Bond object called b that intends to characterize a real bond (either corporate or governmental) with an issue price of $97.5 (corresponding to a face value of $100), a yield of 30% and dividend rate of 20% (yearly).

The order of member initializations within the member initialization list is not relevant, just as in the Stock object. Users may specify the member initialization list in whichever order they like.

The available member functions for the Bond objects include:

```
// Get/Set the initial price of the bond at time 0
Real Price;

// Get/Set yield of the bond
// This affects the maturity as well as the yield and maturity are
functionally //related.
Real Yield;

// Get/Set coupon rate of the bond
Real Coupon;
```

```
// Get/Set name of the bond
String Name;
```

3.8.3 Portfolio

Portfolio is an even higher-level object than Stock or Bond, in the sense that it is actually nothing but an assembly of these basic financial instruments. In the financial world, a portfolio consisting of a bunch of different financial instruments is often used to hedge the potential market risk, as by carefully select the ratio (or percent) of the constituent financial instruments (which is known as portfolio optimization), the risks of any individual investment instrument tend to be "diversified away" and the whole portfolio would be insensitive to any particular movement in the financial market.

To define a portfolio, one should conform to the following syntax:

```
Portfolio <ID> = Portfolio
(
     Capital = <REAL>;
     Components = {<COMPONENT_LIST>}
 );
```

Where <ID> is the name of the portfolio and must be a valid identifier name. The <REAL> followed by the "Capital=" is a positive real number specifying the total amount of capital available for the portfolio investment.

The <COMPONENT LIST> consists of one or more elements separated by the comma, which must be either the array or single element of the defined financial instrument types.

An example:

```
Portfolio P1 = Portfolio
(
     capital = 1000,
     components = {s[10], b[5]}}
);
```

Where we defined a portfolio called P1, with an initial investment of $1000 (at time zero). The portfolio is made up of 10 shares of stocks s and 5 bonds b (see the previous two sections for the definition of s and b).

The above definition actually has another implications. Since when the portfolio is created (implicitly at the time zero), stock s is priced at $24.5 and bond b priced at $97.8, the total amount of capital spent on the financial instruments is 24.5*10 + 5*97.5= $732.5. Therefore there is an extra amount of capital amounting to 1000-732.5 = $267.5. This extra amount of $267.5 capital is implicitly treated as cash deposited in the money market which is stored in a member variable called Cash and may be accessed by the member property Cash. The cash earns a fixed interest rate specified by the INTEREST_RATE constant whose value is 0.020 (more or less the average interest rate for the saving account in U.S. banks for the previous three years). Note that Cash may be negative, in which case the portfolio achieves the investment by borrowing money at the same interest rate. In finance this is called "leverage".

The portfolio object lies at the heart of our PSL language. Basically we can perform a lot of interesting tasks on a portfolio. Available member functions are:

```
    // Get/Set the cash amount in the portfolio
Real Cash;

    // Get number of different kinds of stocks
      int getNumStock();

    // Get number of different kinds of bonds
```

int getNumBond();

// Explicitly set the percent ratio of the Stock with a specify name, which must
//be a <String> type such as "IBM"
Void setStockPortion(String Name);

// return the percent ratio of the Stock with a specify name, which must be
//a <String> type such as "IBM"
Real getStockPortion(String Name);

// Explicitly set the percent ratio of the Bond with a specify name, which
//must be a <String> type such as "Gov"
Void setBondPortion(String Name);

// return the percent ratio of the Bond with a specify name, which must be a
//<String> type such as "Gov"
Real getBondPortion(String Name);

// Test whether the portfolio is leveraging or not, i.e., whether the Cash
//amount is less than zero or not.
Boolean isLeverage();

// Optimize the portfolio composition (only if there are more than 1 different
//kind of correlated stocks).
Void Optimize();

// Perform a dynamic simulation of the portfolio. The Stocks will follow the
//lognormal stochastic process. The plot parameter indicates whether a
//graphic output is desired or not.
Void Simulation(Real timestep, Int nstep, boolean plot);

3.9 Functions

3.9.1 Defining

PSL uses the following syntactic notation for function definitions.
When a function needs to terminate it must invoke the return statement.

function *return-type identifier* (*expression-list* $_{opt}$)
{
    ***statement-list*;**
}

3.9.2 Invocation

The following syntax is used in order to invoke a function, where *identifier* represents the name of the function.

identifier (*expression-list* $_{opt}$);

4 Project Plan

4.1 Team Responsibilities

Alexander Besidski	Front-end, Lexer/Parser
Xin Li	Front-end, Tree walker, interface between front & back-end
Jian Huang	Back-end
Wei-Chen Lee	Testing and Documentation

4.2 Programming Style

The general rules that we followed are listed below. The same programming style was used for both Java and ANTLR code.

4.2.1 Commenting

Under two considerations we use comments to explain our codes more clearly. First, if we cannot understand what the class or function-method does by looking at its name. Second, team members have different opinions about certain code and it needs to be modified iteratively, we use comments to mark and explain our thought in programs.

4.2.2 Identification Format

Code structure need to be unify for well-understanding and debugging. For the structure block we mention here including class, functional, conditional, looping blocks. We use first format in the front-end codes and second format in the back-end codes.

Function
{
……
}
	Function{
	……
	}

4.3 Software Project Environment

In our project, we combine ECLIPSE with ANTLR as programming environment. The ECLIPSE can automatically generate all the necessary files from antlr .g grammars and provide fully integrated support for .g files as if they were .java files. It totally removes any need for make files and makes using antlr a breeze. It also support for CVS repositories assessed via SSH to check in/out

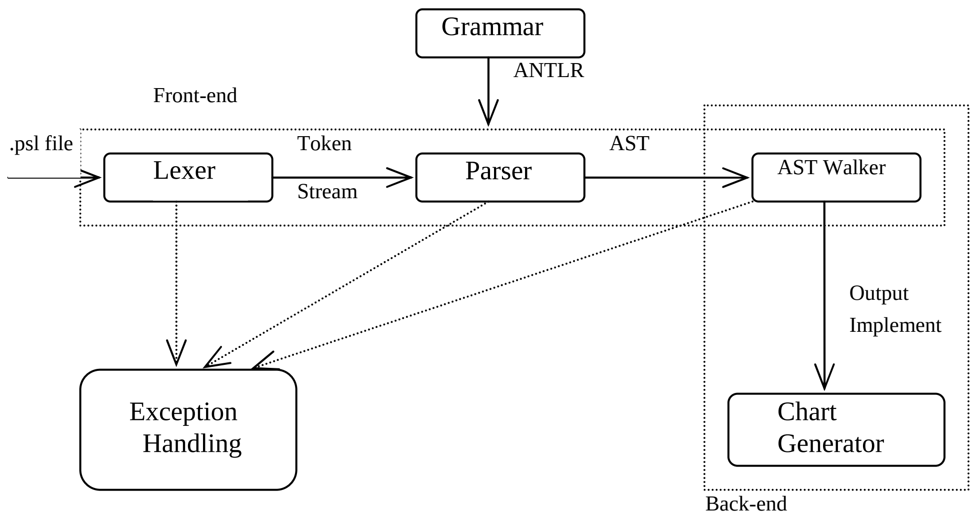
4.4 Project Timeline and Log

We had set several milestones with specific deadlines, set up weekly meetings with specific agendas to discuss any difficulties and provide the team members with regular updates. Here is the timeline that we used for the project.

Basic Idea about Project Language	Sep. 20,2004
Language White Paper	Sep. 27,2004
Language Grammar	Oct. 06,2004
Eclipse+ANTLR Set up	Oct. 14,2004
Language Reference Manual	Oct. 21,2004
Lexer/Parser	Nov. 10,2004
Tree Walker	Nov. 14,2004
Symbol Table	Nov. 19,2004
Front-end	Nov. 27,2004
Regression Test	Nov. 29,2004
Back-end	Dec. 11,2004
Testing and Error Recovery	Dec. 18,2004
Complete Project	Dec. 19,2004

5 Architecture Design

5.1 Block Diagram of PSL Interpreter

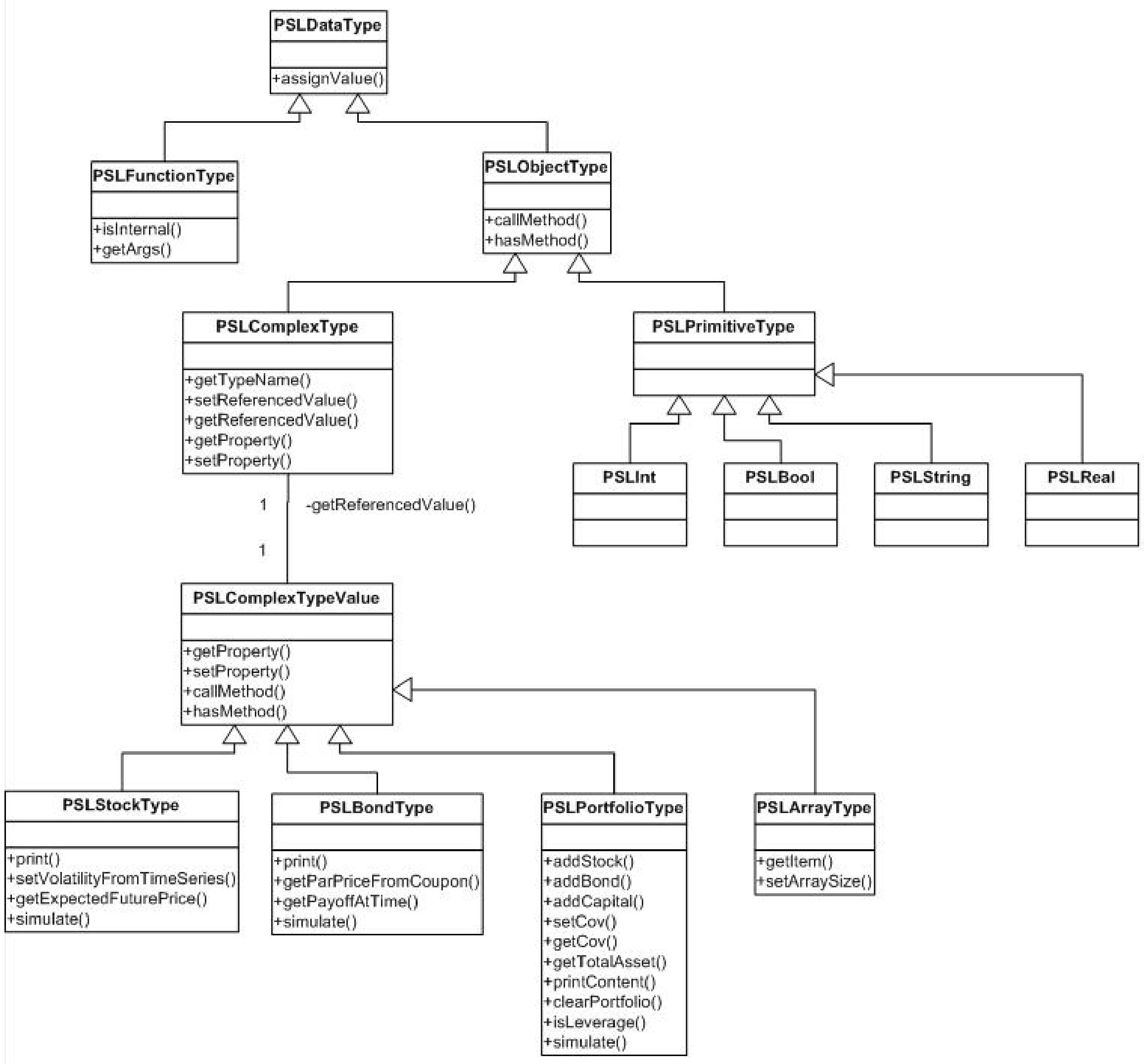


Front-end
The source .psl file is first passed through the lexer that it'll produce a stream of tokens. Then the tokens are parsed by the parser according to the PSL grammar rules. The parser constructs an AST(Abstract Syntax Tree) that it will sent to the AST walker at the backend. If it occurs a syntax error, it'll pass the codes to exception handling, showing type of error in the console screen.

Back-end
After everything goes right in the front-end, lots of statistics will be done in the back-end. For example, correlation between two financial instruments, present value of a cash flow are calculated according to the front-end data and back-end portfolio (stock, bond) history database. Lots of simulations are calculated and present as charts for execution. Charts represent the trend during a certain of time (which defined by users) of different kinds of portfolios.

5.2 Hierarchy of PSL Package

5.2.1 Front-end



5.2.2 Back-end

Chart	--- The Interface for general charts.
BaseChart	--- The abstract class for charts, basic fields and methods defined.
BaseScaleChart	--- Drawing common elements, such as scales, labels, legend.
LineChart	--- Drawing Line chart.
ChartItem	--- The Interface for items on a chart.
BaseChartItem	--- An abstract class, handling items' location.
TextChartItem	--- Handling text items, such as labels, description, and title.
ChartModel	--- An Interface.
BaseChartModel	--- A container for data sets presented on the chart.
ChartAttributeDefinition	--- A chart attribute.
ChartAttributes	--- Define operations on the chart attribute set.
ChartFactory	--- The Factory Pattern applied; Be able to render different types of chart.
ChartItemPostion	--- The chart item position information.
DataLoader	--- Loading time series data form a text file.
DataPlotter	--- A wrapper for Chart Drawing functions.
DataWriter	--- Writing time series data to a text file.
GeoBrownMove	--- Simulating the Geometric Brownian Motion.
PrintFormat	--- A wrapper for printf function.
Statistics	--- Some basic statistics functions.

6 Testing Plan

6.1 Goal

Since our language provides user-define variables and functions based on PSL-defined grammar structure. We need to consider our test examples from users' view. Our approach is to test PSL step by step. First, we divide PSL into different type structures, then sew them together to conduct concrete .psl file. In each level, if a test example passes compilation, we examine the correctness of Lexer/Parser through AST (Abstract Syntax Tree). If it fails the examination, we check the re-direction of exception handling.

6.2 Lexer/Parser Test

The test for Lexer/Parser needs to be done at very-beginning state. It reflects the legality of PSL grammar definition. For Lexer token generation, we need to check if we miss any possible token-generation rules. Fox example, is it legal to initialize a variable with a number. Reorder the sequences of the same file to make sure that there's no ambiguity in PSL syntax. We need to make sure the consistence of our PSL parser generation.

6.3 Regression Test

Regression Test is very important inside test plan, because it can save us lots of time when we adjust our PSL program in each state. We may occur a new error when we fixed a old error It is tiresome to check some trivial things iterately. In our regression test system, we examine every possible type checking error, scoping checking error (for same type), and memory allocation checking. Type checking error includes normal type (Integer, Real, String and Boolean), array type conversion and complex type (Bond, Stock and Portfolio) conversion. Scoping checking error include "{" "}" matching and illegal using variable in out-scope area. Memory allocation checking includes array space usage legality.

6.4 Special Case Test

There are still some test samples that we cannot use regression test. Therefore, we need to examine them separately case by case. The purpose of special case test is to make sure users may code their program accidentally or on purpose in a non-programmer point of view. We must make sure these illegal code will not crack the whole PSL structure and advancially point out where the error code is.

6.5 Integrated Test

Integrated test is the final test to make sure that everything goes the right track. Generally speaking, there cannot exit any big problem if PSL program pass the former tests. But scoping is an important issue to testify. During the integrated test, we need to make sure PSL does not confuse under the following scenario: re-declare the same variable name but with a different type, re-declare the same variable name and type but in a different scope.

7 Lesson Learned

- **Doing a project is the best way to learn this course.** Throughout the project, we learned the complexity of designing and implementing a new programming language, and we also got a better understanding for the knowledge that are taught in class. Now we have a clear view for different aspects of programming language and compiler.
- **Right tools make things easy.** In this project, we choose like ANTLR, Eclipse and CVS as our developing tools. They are proved as right choice. ANTLR, Eclipse and CVS can work together smoothly, saving us a lot of time. With ANTLR plug-in, we can compile and debug the ANTLR code just within Eclipse IDE, without switching back to command line.
- **Version Control is necessary for team work.** Using version control is once again proved to be greatly important in a team project. We choose CVS as it is commonly used and fully integrated into Eclipse. It happened several times that we went back to previous versions when a new functionality broke something that was working before. Without version control, there must be great delay for our project.
- **It would be a good idea to integrate codes as early as possible.** Our project is developed by four persons separately. Everything goes the right track when the codes are tested separately, but it fails when all components are integrated. It turned out to be some misunderstanding between developers. This cause some additional work for us.
- **Communication is the key to a successful team project.** Having efficient and effective communication, we know what is going on, who is doing what, and how things are going in general. It makes it easy to synchronize and make sure one's work is coherent with everything else.

A Appendix

A.1 Lexer.g

```
header
{
package frontend;
}

class PSLLexer extends Lexer;

options
{
  k = 2;
  charVocabulary = '\3'..'\377';
  importVocab = PSLAntlrParser;
  exportVocab = PSLAntlrLexer;
  testLiterals = false;
}

protected ALPHA :  'a'..'z' | 'A'..'Z' | '_';

protected DIGIT  :  '0'..'9';

WS  :  (' ' | '\t')+
    { $setType(Token.SKIP); };

NL  :  ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
    { $setType(Token.SKIP); newline(); } ;

COMMENT  :  ("/*"  (
            options {greedy=false;}  :  (NL)
                        |  ~( '\n' | '\r' )
          )* "*/"
    |  "//" (~( '\n' | '\r' ))* (NL | )) // support comments on the last line of
the file
        { $setType(Token.SKIP); };

LPAREN  :  '(';

RPAREN  :  ')';

LBRACE  :  '{';

RBRACE  :  '}';

LBRK  :  '[';

RBRK  :  ']';

SEMI  :  ';';

DOT    :  '.';

COMMA  :  ',';

// assignment operators
ASSIGN  :  '=';

PLUSEQ  :  "+=";

MINUSEQ  :  "-=";
```

```
DIVEQ  :  "/=";

MULEQ  :  "*=";

// unary operators
NOT  :  "!";

// comparison operators
EQ  :  "==";

NEQ  :  "!=";

LT  :  "<";

LTE  :  "<=";

GT  :  ">";

GTE  :  ">=";

// arithmetic operators
MINUS  :  '-';

PLUS  :  '+';

DIV    :  '/';

MUL    :  '*';

// logical operators
LOR    :  "||";

LAND  :  "&&";

// only this rule needs to be checked for tokens when parser rules use
// double quoted strings, therefore we only need to set testLiterals=true here
ID  options { testLiterals = true; }
  :  ALPHA (ALPHA | DIGIT)*;

// constants
protected EXPONENT  :  (('E'|'e') ('+'|'-')? (DIGIT)+);

protected INT  : (DIGIT)+;

protected REAL  : (DIGIT)+ '.' (DIGIT)+ (EXPONENT)?;

// resolve ambiguity betwee INT and REAL
INT_OR_REAL  :  ((DIGIT)+ '.') => REAL {$setType(REAL);}
      |  INT {$setType(INT);};

// C-style double quotation mark escaping
STRING  :  '"'! (('\\' '"') => '\\'! '"' | ~('"' | '\r' | '\n'))* '"'!;
```

A.2 Parser.g

```
header
{
package frontend;
}

class PSLParser extends Parser;

options
{
  k = 3;
  buildAST = true;
  exportVocab = PSLAntlrParser;
}

tokens
{
  NODE_FOR;
  FOR_INIT;
  FOR_COND;
  FOR_UPDT;
  UNARY_MINUS;
  PRIMITIVE_DECL;
  COMPLEX_DECL;
  COMPLEX_INITIALIZER;
  PRIMITIVE_INITIALIZER;
  COMPLEX_PROP_INITIALIZER;
  PROP_INVOKE;
  METHOD_INVOKE;
  FUNC_INVOKE;
  INVOKE_ARGS;
  ARRAY_ELEMENT;
  FUNC_DECL;
  ARGS_DECL;
  ARG_DECL;
  ARRAY_DECL;
  STMT_LIST;
  RET_TYPE;
}

statement_list  :  (statement | func_decl)+ EOF!
          { #statement_list = #([STMT_LIST, "statement_list"], statement_list); }
          ;

func_decl!  :  "function"^ id:ID
        LPAREN! (a:args_decl) RPAREN!
        s:compound_stmt
        {
            #func_decl = #([FUNC_DECL, "function"], id, a, s);
        };

statement  :  compound_stmt
      |  variable_declarator
      |  expression_stmt
      |  if_stmt
      |  while_stmt
      |  for_stmt
      |  break_stmt
      |  continue_stmt
      |  return_stmt;

compound_stmt  :  LBRACE! (statement)+ RBRACE!
          { #compound_stmt = #([STMT_LIST, "compound_stmt"], compound_stmt); };
```

```
args_decl  :  (arg_decl (COMMA! arg_decl)*)?
        { #args_decl=#([ARGS_DECL, "args_decl"], args_decl); };

arg_decl  :  (type) ID (LBRK (INT)? RBRK!)?
        { #arg_decl=#([ARG_DECL, "arg_decl"], arg_decl); };

// correct dangling else handling
if_stmt  :  "if"^ LPAREN! expression RPAREN! "then"!
      statement
      (options {greedy=true;} : ("else"! statement))?;

for_stmt!  :  "for"^
        LPAREN!
          (i:for_init) SEMI!  // initializer
          (c:for_cond) SEMI!  // condition test
          (u:for_updt)    // updater
        RPAREN!
        b:statement  // statement to loop over
        {
            #for_stmt = #([NODE_FOR, "for"], b,i,c,u);
        };

for_init:       (expression_or_declarator_list )?
                { #for_init=#([FOR_INIT, "for_init"], #for_init); };

for_cond:     (expression)?
          { #for_cond=#([FOR_COND, "for_cond"], for_cond); };

for_updt:       (expression_list)?
                { #for_updt=#([FOR_UPDT, "for_updt"], for_updt); };

// comma separated list of expressions
expression_list  :  expression (COMMA! expression)*
          { #expression_list = #([STMT_LIST, "expression_list"],
expression_list); };

expression_or_declarator  :  expression
              |  primitive_declarator
              |  complex_decalarator
              |  array_declarator;

expression_or_declarator_list  :  expression_or_declarator (COMMA!
expression_or_declarator)*
                  { #expression_or_declarator_list = #([STMT_LIST,
"expression_or_declarator_list"], expression_or_declarator_list); };

break_stmt  :  "break" SEMI!;

continue_stmt  :  "continue" SEMI!;

while_stmt  :  "while"^ LPAREN! expression RPAREN! statement;

return_stmt  :  "return"^ (expression)? SEMI!;

expression_stmt  :  expression SEMI!;

expression  :  assignment;

// assignment
assignment  :  logical_expr
        ( (ASSIGN^ | PLUSEQ^ | MINUSEQ^ | DIVEQ^ | MULEQ^) logical_expr )?;

// logical comparison
logical_expr  :  equality_expr ((LOR^ | LAND^) equality_expr)*;
```

```
// logical equality
equality_expr  :  rel_expr ((EQ^ | NEQ^) rel_expr)*;

// string and numeric relations (no recursion allowed: a<b<c is illegal)
rel_expr    :  additive_expr ((LT^ | LTE^ | GT^ | GTE^) additive_expr)?;

// addition/substraction
additive_expr  :  mult_expr ((MINUS^ | PLUS^) mult_expr)*;

// multiplication
mult_expr    :  unary_expr ((MUL^ | DIV^) unary_expr)*;

// logical negation
unary_expr    :  (NOT)? signed_expr;

// handle signed expressions by introducing new tokens
signed_expr    :  MINUS! primary_expr
          { #signed_expr=#([UNARY_MINUS, "unary_minus"], signed_expr); }
        |  PLUS! primary_expr // ignore the plus
        |  primary_expr;

// highest precedence level
// includes constants, identifiers, arrays,
// function, property and method calls
primary_expr!  :  c:constant
          { #primary_expr = #(#c); }
        |
          // function or identifier
          (
            (LPAREN! l:logical_expr RPAREN!)
            { #primary_expr = #l; }
            |
            id:ID
            { #primary_expr = #(#id); }
            |
            f:func_invoke
            { #primary_expr = #f; }
          )
          // array index
          (LBRK! a:additive_expr RBRK!
          { #primary_expr = #([ARRAY_ELEMENT, "array_element"], #primary_expr,
#a); } )?
          // recursive sequence of method and property calls
          (
            !DOT
            (
              p:ID
              { #primary_expr = #([PROP_INVOKE, "prop_invoke"], #primary_expr,
#p); }
              |
              m:ID LPAREN! (args:invoke_args)? RPAREN!
              { #primary_expr = #([METHOD_INVOKE, "method_invoke"], #primary_expr,
#m, #args); }
            )
            (
              LBRK! a2:additive_expr RBRK!
              { #primary_expr = #([ARRAY_ELEMENT, "array_element"], #primary_expr,
#a2); }
            )?
          )*;

func_invoke  :  ID LPAREN! (invoke_args)? RPAREN!
        {#func_invoke = #([FUNC_INVOKE,"func_invoke"], #func_invoke); };

invoke_args  :  expression (COMMA! expression)*
```

```
        { #invoke_args = #([INVOKE_ARGS,"invoke_args"], #invoke_args); };

variable_declarator  :
          (  primitive_declarator
          |  complex_decalarator
          |  array_declarator ) SEMI!;

primitive_declarator  :  primitive_type ID (ASSIGN! primitive_initializer)?
              { #primitive_declarator = #([PRIMITIVE_DECL,"primitive_decl"],
#primitive_declarator); };

complex_decalarator  :  complex_type ID (ASSIGN! (primitive_initializer |
complex_initializer))?
            { #complex_decalarator = #([COMPLEX_DECL,"complex_decl"],
#complex_decalarator); };

array_declarator!  :  t:type i:ID (LBRK! (a:additive_expr)? RBRK!)
            { #array_declarator = #([ARRAY_DECL,"array_decl"], #i, #t, #a); };

// primitive initializion is simply setting
// the declared variable to a value
primitive_initializer  :  logical_expr
              { #primitive_initializer =
#([PRIMITIVE_INITIALIZER,"primitive_init"], #primitive_initializer); };

// complex initializion allows assignment of
// values to complex type properties
complex_initializer  :  "new"! LPAREN! complex_property_initializer (COMMA!
complex_property_initializer)* RPAREN!
            { #complex_initializer = #([COMPLEX_INITIALIZER,"complex_init"],
#complex_initializer); };

// complex type property initializer
complex_property_initializer  :  ID ASSIGN! logical_expr
                  { #complex_property_initializer =
#([COMPLEX_PROP_INITIALIZER,"complex_init_prop"],
#complex_property_initializer); };

type  :  primitive_type
    |  complex_type;

primitive_type  :  "int"
        |  "real"
        |  "string"
        |  "boolean";

complex_type  :  "portfolio"
        |  "stock"
        |  "bond";

constant  :  "true"
      |  "false"
      |  INT
      |  REAL
      |  STRING;
```

A.3 Walker.g

```
header
{
package frontend;
}

{
  import java.io.*;
  import java.util.*;
}

class PSLWalker extends TreeParser;
options
{
  importVocab = PSLAntlrLexer;
  exportVocab = PSLAntlrWalker;
}

{
  PSLInterpreter interpreter = new PSLInterpreter();
}

expr returns [PSLDataType r]
{
  r = null;
  PSLDataType a,b,cond;
  PSLDataType[] init,update;
  String objname,classname;
  PSLDataType temp;
  if(interpreter.isAtReturn())
  {
    PSLDebugger.warn("Unreachable code: "+_t.getText()+"  (Already returned from
previous scope.). At line:"+_t.getLine());
    return r;
  }
}

  :  #(id:ID  { r = interpreter.getObject( id.getText() ); r.setIslValue(true);} )
  |  #(LOR   a=expr right_or:.)
    {
      if (a instanceof PSLBool)
              r = ( ((PSLBool)a).getValue() ? a : expr(#right_or));
      else
              r = a.or(expr(#right_or));
        }
  |  #(LAND   a=expr right_and:.)
    {
            if (a instanceof PSLBool)
              r = ( ((PSLBool)a).getValue() ? expr(#right_and) : a);
      else
              r = a.and(expr(#right_and));
        }
  |  #(PLUS  a=expr b=expr) { r = a.add(b);}
  |  #(MINUS a=expr b=expr) { r = a.sub(b);}
  |  #(MUL  a=expr b=expr) { r = a.mul(b);}
  |  #(DIV   a=expr b=expr) { r = a.div(b);}
  |  #(MOD a=expr b=expr) {   r = a.mod( b ); }
  |  #(UNARY_MINUS a=expr) { r = a.uminus(); }
  |   #(ASSIGN a=expr b=expr) { a.assignValue(b);  }
  |  #(PLUSEQ a=expr b=expr) { a.assignValue(a.add(b)); }
  |  #(MINUSEQ a=expr b=expr) { a.assignValue(a.sub(b)); }
  |  #(MULEQ a=expr b=expr) { a.assignValue(a.mul(b)); }
  |  #(DIVEQ a=expr b=expr) { a.assignValue(a.div(b)); }
```

```
   |  #(MODEQ a=expr b=expr) { a.assignValue(a.mod(b)); }
   |  #(EQ a=expr b=expr) { r = a.eq(b); }
   |  #(NEQ a=expr b=expr) { r = a.ne(b); }
   |  #(LT a=expr b=expr) { r = a.lt(b); }
   |  #(LTE a=expr b=expr) { r = a.le(b); }
   |  #(GT a=expr b=expr) { r = a.gt(b); }
   |  #(GTE a=expr b=expr) { r = a.ge(b); }
   |  i:INT { r = new PSLInt(i.getText());}
   |  f:REAL { r= new PSLReal(f.getText()); }
   |  s:STRING { r= new PSLString(s.getText());}
   |  "true"    { r = new PSLBool(true);}
   |  "false"    { r = new PSLBool(false);}
   |  #("if" cond=expr then_part:. (else_part:.)?)
     {
       assert(cond instanceof PSLBool);
       if(((PSLBool)cond).getValue())
         r = expr( #then_part);
       else if(else_part != null)
         r = expr( #else_part);
     }
   |  #(NODE_FOR for_body:. for_init:FOR_INIT for_cond:FOR_COND
for_update:FOR_UPDT )
     {
       interpreter.executeFor(this,#for_init, #for_cond,#for_update,#for_body);
     }
   |  #("while" wcond:. while_body:.)
     {
       if(!(expr(wcond) instanceof PSLBool))
       {
         PSLDebugger.error("Invalid conditional statement in <while>;");
       }
       interpreter.executeWhile(this,#wcond,#while_body);
     }
     |   #("break"   {interpreter.setBreak(); })
   |   #("continue" {interpreter.setContinue(); })
   |  #("return" (a=expr { r=a;})? {interpreter.setReturn(r);})
   |  #(FUNC_INVOKE fname:ID (args:.)?)
     {
        String function_name = fname.getText();
        assert(interpreter.functionDefined(function_name));
        // args might be null
        if(args != null)
           r=interpreter.InvokeFunction(this,function_name,expr_list(args));
         else
           r=interpreter.InvokeFunction(this,function_name,null);
       }

   |  #(PROP_INVOKE temp=expr prop_name:ID)
     {
       if (!(temp instanceof PSLComplexType))
         throw new PSLException("properties can only be invoked on complex types:
" + prop_name.getText());
       r = ((PSLComplexType)temp).getProperty(prop_name.getText());
       r.setIslValue(true);
     }
   |   #(METHOD_INVOKE temp=expr memb:ID (arglist:.)?)
     {
       if (!(temp instanceof PSLObjectType))
         throw new PSLException("methods can only be executed on complex and
primitive types: " + memb);
           String method = memb.getText();
       r = ((PSLObjectType)temp).callMethod(method, arglist == null ? null :
expr_list(arglist));
     }
   |  #(PRIMITIVE_DECL primitive_type:. primitive_name:ID
(primitive_initializer:.)?)
     {
       r = interpreter.registerPrimitiveType(primitive_type.getText(),
primitive_name.getText());
       r.setIslValue(true);
       if (primitive_initializer != null)
       {
         ((PSLPrimitiveType)r).assignValue(expr(primitive_initializer));
       }
     }
   |   #(FOR_INIT   (r=expr)?)
   |   #(FOR_COND   (r=expr)?)
   |   #(FOR_UPDT   (r=expr)?)
   |  #(PRIMITIVE_INITIALIZER r=expr)
   |  #(COMPLEX_DECL complex_type:. complex_name:ID
       {
         r = interpreter.registerComplexType(complex_type.getText(),
complex_name.getText());
         r.setIslValue(true);
       }
       (
         (
           primitive_initializer2:PRIMITIVE_INITIALIZER
           { ((PSLComplexType)r).assignValue(expr(primitive_initializer2)); }
         )
         |
         (
           complex_init:COMPLEX_INITIALIZER

{ ((PSLComplexType)r).assignProperyValues(complex_initializer(complex_init)); }
         )
         | /*nothing*/
       )
     )
   |  #(FUNC_DECL funname:ID args_decl:. stmt_body:.)
     {
       interpreter.registerFunction(funname.getText(), expr_list(args_decl),
#stmt_body);
     }
   |  #(ARG_DECL arg_type:. arg_name:ID (lbrk:LBRK (array_size:.)?)?)
     {
       r = new PSLFunctionArgument(arg_type.getText(),
           arg_name.getText(),
           lbrk == null ? false : true,
           array_size == null ? 0 : (new PSLInt(array_size.getText())).getValue());
     }
   |  #(STMT_LIST
       {
         boolean compoundStatementEntered = false;
         if (#STMT_LIST.getText().equals("compound_stmt"))
         {
           interpreter.EnterNewScope();
           compoundStatementEntered = true;
         }
       }
       (
         stmt:.
         {
           if(interpreter.canProceed())
           {
             r = expr(stmt);
           }
           else
             break;
         }
       )+
       {
         if (compoundStatementEntered)
           interpreter.ExitCurrentScope();
       }
         )
   |  #(ARRAY_DECL aname:. atype:. (asize:.)?)
     {
       r = interpreter.registerArray(aname.getText(), atype.getText(), asize ==
 null ? null : expr(asize));
       r.setIslValue(true);
     }
   |  #(ARRAY_ELEMENT array_ref:. array_index:.)
     {
       PSLDataType reference = expr(array_ref);
       PSLDataType index = expr(array_index);

       if (reference instanceof PSLComplexType)
       {
         if (((PSLComplexType)reference).getReferencedValue() instanceof
 PSLArrayType)
         {
           PSLArrayType array =
 (PSLArrayType)((PSLComplexType)reference).getReferencedValue();

           if (!(index instanceof PSLInt))
             throw new PSLException("array index must be an integer: " + index);
           r = array.getItem(((PSLInt)index).getValue());
           r.setIslValue(true);
         }
         else
           throw new PSLException("left hand value is not an array.");
       }
       else
         throw new PSLException("left hand value is not an array.");
     }
   ;

 complex_initializer returns [propValuePair[] r]
 {
   Vector v;
   r = null;
 }
   :  #(COMPLEX_INITIALIZER
       {v = new Vector();}
       (
         init:. {v.add(complex_property_initializer(init));}
       )*
       {
         r = (propValuePair[]) v.toArray(new propValuePair[0]);
       }
     );

 complex_property_initializer returns [propValuePair r]
 {
   r = null;
 }
   :  #(COMPLEX_PROP_INITIALIZER prop_name:ID prop_value:.)
     {
       r = new propValuePair(prop_name.getText(),expr(prop_value));
     };

 expr_list returns [PSLDataType[] r]
 {
   PSLDataType exp;
   Vector v;
   r = null;
 }
   : #(INVOKE_ARGS      {v = new Vector();}
       (
       exp = expr      {v.add(exp); }
     )*
   )  { r = interpreter.vecToArray(v);}
   | #(ARGS_DECL  {v = new Vector();}
     (arg_dec:. { v.add(expr(arg_dec)); }
     )*
     { r = interpreter.vecToArray(v); }
   )
   ;
```

A.4 NoSuchObjectError.java

```java
package frontend;

public class NoSuchObjectError extends PSLException
{
  NoSuchObjectError( String objname )
  {
    super("");
    System.err.println(objname+" is not a valid PSL object reference");
  }
}
```

A.5 propValuePair.java

```java
package frontend;

public class propValuePair
{
  private String propName;
  private PSLDataType propValue;

  public void print()
  {
    System.out.print(propName + " = " + propValue);
  }

  public propValuePair(String pname, PSLDataType pvalue)
  {

    this.propName = pname;
    this.propValue = (PSLDataType)pvalue.copy();
  }

  public String getProperty()
  {
    return propName;
  }

  public PSLDataType getValue()
  {
    return propValue;
  }
}
```

A.6 PSLArrayType.java

```
package frontend;

import java.util.*;

public class PSLArrayType extends PSLComplexTypeValue
{
  private String containedTypeName = "";
  private ArrayList items = new ArrayList();

  public void print()
  {

  }

  public String getTypeName()
  {
    return "array";
  }

  public String getContainedTypeName()
  {
    return this.containedTypeName;
  }

  public void setContainedTypeName(String type)
  {
    if (PSLPrimitiveType.isValidPrimitiveType(type) ||
PSLComplexType.isValidComplexType(type))
      this.containedTypeName = type;
    else
      throw new PSLException("Unrecognized type used for array declaration: " +
type);
  }

  public int getArraySize()
  {
    return items.size();
  }

  public PSLDataType getItem(int index)
  {
    if (this.getArraySize() == 0)
      throw new PSLException("Array uninitialized.");
    else if (index < 0 || index+1 > this.getArraySize())
      throw new PSLException("Index " + index + " outside of array boundaries 0.."
+ this.getArraySize());
    else
      return (PSLDataType)this.items.get(index);
  }

  public void setItem(int index, PSLDataType element)
  {
    if (this.getArraySize() == 0)
      throw new PSLException("Array uninitialized.");
    else if (index < 0 || index+1 > this.getArraySize())
      throw new PSLException("Index " + index + " outside of array boundaries 0.."
+ this.getArraySize());
    else
    {
      PSLDataType element_p = (PSLDataType)(element.copy());
      this.items.set(index,element_p);
    }
  }

  public void setArraySize(int size)
  {
    // if initialized with primitive types allocate storage right away
    this.items.clear();
    for(int i=0; i<size; i++)
      if (PSLPrimitiveType.isValidPrimitiveType(this.getContainedTypeName()))

this.items.add(PSLPrimitiveType.createPrimitive(this.getContainedTypeName()));
      else

this.items.add(PSLComplexType.createComplexType(this.getContainedTypeName(),
false));
  }

  public ArrayList getMethods()
  {
    ArrayList methods = new ArrayList();
    methods.add("getSize");

    return methods;
  }

  public PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    if (params != null && params.length > 0)
      throw new PSLException("array::getSize does not accept any arguments.");
    return new PSLInt(this.getArraySize());
  }
}
```

A.7 PSLBondType.java

```java
/**
 * @author xinli
 *
 *  Wrapper class for bond
 */

package frontend;

import java.util.*;

import backend.DataPlotter;
import backend.PrintFormat;
import backend.Statistics;

public class PSLBondType extends PSLComplexTypeValue
{

  public PSLBondType()
  {
    this.defineProperty("Name", new PSLString(""));
    //this.defineProperty("Price", new PSLReal(0));
    this.defineProperty("InterestRate", new PSLReal(0));
    this.defineProperty("Maturity", new PSLInt(0));
    this.defineProperty("Coupon", new PSLReal(0));
  }

  public String getTypeName()
  {
    return "bond";
  }

  public ArrayList getMethods()
  {
    ArrayList list = new ArrayList();
    list.add("print");
    list.add("getParPriceFromCoupon");
    list.add("getPayoffAtTime");
    list.add("simulate");
    return list;
  }

  public double calculatePrice()
  {
    double coupon;

    coupon = ((PSLReal)getProperty("Coupon")).getValue();
    int Maturity = ((PSLInt)getProperty("Maturity")).getValue();
    double yield = ((PSLReal)getProperty("InterestRate")).getValue();
    double data[] = new double[Maturity+1];
    data[0] = 0.0;
    for(int i=1; i<Maturity; i++)
    {
      data[i] = coupon;
    }
    data[Maturity] = 100 + coupon;
    double pv = backend.Statistics.calPresentValue( data, yield/100.0);
    return pv;
  }
  public PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    if (methodName.equals("print"))
    {
      if (params != null && params.length > 0)
        throw new PSLException("bond::print does not accept any arguments.");
      super.print();
    }
    else if(methodName.equals("getParPriceFromCoupon"))
    {
      double coupon;
      if(params == null)
      {
        coupon = ((PSLReal)getProperty("Coupon")).getValue();
      }
      else
      {
        if(params.length != 1)
        {
          throw new PSLException("bond::getParPriceFromCoupon(real): 1 argument
expected.");
        }

        if(!(params[0] instanceof PSLReal))
        {
          throw new PSLException("bond::getParPriceFromCoupon(real): argument
must be of real type.");
        }

        coupon = ((PSLReal)(params[0])).getValue();
        if(coupon <= 0)
        {
          throw new PSLException("bond::getParPriceFromCoupon(real): argument
must be positive.");
        }
      }
      int Maturity = ((PSLInt)getProperty("Maturity")).getValue();
      double yield = ((PSLReal)getProperty("InterestRate")).getValue();
      double data[] = new double[Maturity+1];
      data[0] = 0.0;
      for(int i=1; i<Maturity; i++)
      {
        data[i] = coupon;
      }
      data[Maturity] = 100 + coupon;
      double pv = backend.Statistics.calPresentValue( data, yield/100.0);

      return new PSLReal(pv);
    }

    else if(methodName.equals("getPayoffAtTime"))
    {
      if(params.length != 1)
      {
          throw new PSLException("bond::getPayoffAtTime(real): 1 argument
expected.");
      }
      if(!(params[0] instanceof PSLReal))
      {
        throw new PSLException("bond::getPayoffAtTime(real): argument must be
real.");
      }
      double time = ((PSLReal)(params[0])).getValue();
      if(time <= 0)
      {
        throw new PSLException("bond::getPayoffAtTime(real):: argument must be
positive.");
      }
      double payoff;
```

```
        int Maturity = ((PSLInt)getProperty("Maturity")).getValue();
        double coupon = ((PSLReal)getProperty("Coupon")).getValue();

        if(time >= Maturity)
        {
          payoff = ((double)Maturity)*coupon + 100.0;
        }
        else if(time < 1.0)
        {
          payoff = 0;
        }
        else
        {
          int int_time = (int)time;
          payoff = coupon*int_time;
        }

      }

      else if(methodName.equals("simulate"))
      {
        if(params.length != 1)
        {
            throw new PSLException("bond::simulate(int): 1 argument expected.");
        }
        if(!(params[0] instanceof PSLInt))
        {
          throw new PSLException("bond::simulate(int): 1 argument must be of real
type.");
        }

        int steps = ((PSLInt)params[0]).getValue();
        if(steps <= 0)
        {
          throw new PSLException("bond::simulate(int): 1 argument must be
positive.");
        }
        int Maturity = ((PSLInt)getProperty("Maturity")).getValue();
        double coupon = ((PSLReal)getProperty("Coupon")).getValue();
        double[] simbondData = new double[steps];

        for(int i=0; i<steps; i++)
        {
          if(i < Maturity)
          {
            // Before maturity, coupon only
            simbondData[i] = i*coupon;
          }
          else
          {
            // At maturity, coupon + face value
            // After maturity, no payoff
            simbondData[i] = Maturity*coupon + 100.0;
          }

        }

            PSLArrayType simbondDataArray = new PSLArrayType();
            simbondDataArray.setContainedTypeName("real");
            simbondDataArray.setArraySize(simbondData.length);

System.out.println("==============================================");
            System.out.println("bond Simulation");
            System.out.println("Name:
```

```
"+((PSLString)getProperty("Name")).getValue());
          System.out.println("Time Steps: "+simbondData.length);
          System.out.println("     Step      Payoff($)");

System.out.println("----------------------------------------------");
          //double totalPayoff = 0.0;
          for(int i=0; i<simbondData.length; i++)
          {
            //totalPayoff += simbondData[i];
            System.out.print("     ");
            PrintFormat.Print_Format(i+1,20);
            PrintFormat.Print_Format(simbondData[i],15);
            System.out.println();
            //System.out.println(""+i+"    "+simbondData[i]);
            simbondDataArray.setItem(i,new PSLReal(simbondData[i]));
          }

          List Data = new ArrayList();
          List ValueName = new ArrayList();

          Data.add(simbondData);
          ValueName.add(((PSLString)getProperty("Name")).getValue());

          DataPlotter.drawPlot(Data, ValueName, "Bond Simulation") ;

System.out.println("==============================================");
          PSLComplexType simbondRef = new PSLComplexType("array");
          simbondRef.setReferencedValue(simbondDataArray);
          return simbondRef;
    }
    else
      throw new PSLException("bond." + methodName + " not implemented.");

    return null;
  }
}
```

A.8 PSLBool.java

```
package frontend;

import java.io.*;

public class PSLBool extends PSLPrimitiveType
{
   boolean value;

   PSLBool( boolean var )
   {
     this.value = var;
   }

   public String getTypeName() {
        return "bool";
  }

   public boolean getValue()
   {
     return value;
   }

   public String toString()
    {
        return new String(""+value);
    }

   public void print() {
     System.out.println( value ? "true" : "false" );
   }

   public PSLDataType add( PSLDataType b)
   {
        if (b instanceof PSLString)
          return new PSLString(this.getValue() + ((PSLString)b).getValue());
        return error( b, "add" );
   }

   public PSLDataType and( PSLDataType b ) {
     if ( b instanceof PSLBool )
       return new PSLBool( value && ((PSLBool)b).value );
      return error( b, "and" );
   }

   public PSLDataType or( PSLDataType b ) {
     if ( b instanceof PSLBool )
       return new PSLBool( value || ((PSLBool)b).value );
      return error( b, "or" );
   }

   public PSLDataType not() {
     return new PSLBool( !value );
   }

   public PSLDataType eq( PSLDataType b ) {
     // not exclusive or
     if ( b instanceof PSLBool )
       return new PSLBool( ( value && ((PSLBool)b).value )
           || ( !value && !((PSLBool)b).value ) );
      return error( b, "==" );
   }
   public PSLDataType ne( PSLDataType b ) {
     // exclusive or
     if ( b instanceof PSLBool )
       return new PSLBool( ( value && !((PSLBool)b).value )
           || ( !value && ((PSLBool)b).value ) );
          return error( b, "!=" );
    }

   // convert other values to bool: only allow Int
   public PSLDataType convert(PSLDataType b)
   {
     if(b instanceof PSLInt)
       return convert(((PSLInt)b).getValue());
     return error(b, "conversion");
   }

   public PSLDataType convert(int b)
   {
     return new PSLBool(b!=0);
   }

   protected void onAssignValue(PSLDataType value)
   {
     if(!(value instanceof PSLBool))
       this.error(value, "incompatible types");
      this.value = ((PSLBool)value).value;
   }

   public PSLDataType copy()
   {
     return new PSLBool(this.value);
   }
}
```

A.9 PSLComplexType.java

```java
/*
 * Created on Nov 20, 2004
 *
 */

package frontend;

import java.util.*;

/**
 * @author Alex Besidski
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class PSLComplexType extends PSLObjectType
{
  private PSLComplexTypeValue value;
  public String typeName;

  /**
   *
   * @param referencedType name of the type referred to by this reference.
   */
  public PSLComplexType(String referencedType)
  {
    this.typeName = referencedType;
  }

  public String getTypeName()
  {
    return this.typeName;
  }

  public PSLComplexTypeValue getReferencedValue()
  {
    return this.value;
  }

  public void setReferencedValue(PSLComplexTypeValue value)
  {
    this.value = value;
  }

  protected void onAssignValue(PSLDataType value)
  {
    if (value == null)
      throw new PSLException("Reference cannot be set to null");

    if (!value.getTypeName().equals(this.typeName))
      throw new PSLException(value.getTypeName() + " cannot be cast to " +
this.typeName);
    else if (this.getTypeName().equals(("array")))
    {
      PSLArrayType left = (PSLArrayType)this.getReferencedValue();
      PSLArrayType right =
(PSLArrayType)((PSLComplexType)value).getReferencedValue();

      if (!left.getContainedTypeName().equals(right.getContainedTypeName()))
        throw new PSLException("Assignment not allowed on arrays containing
mismatched types.");
```

```
      if (left.getArraySize() > 0)
        throw new PSLException("Assignment to an already initialized array not 
allowed.");
    }

    this.value = ((PSLComplexType)value).getReferencedValue();
  }

  public void assignProperyValues(propValuePair[] values)
  {
    if (value != null)
    {
      for(int i=0; i<values.length; i++)
        this.setProperty(values[i].getProperty(), values[i].getValue());
    }
  }

  public ArrayList getMethods()
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");

    return this.value.getMethods();
  }

  protected PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");

    return this.value.callMethod(methodName, params);
  }

  public PSLDataType getProperty(String prop)
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");

    return this.value.getProperty(prop);
  }

  public void setProperty(String prop, PSLDataType value)
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");
    this.value.setProperty(prop,value);
  }

  public void print()
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");

    this.value.print();
  }

  public static boolean isValidComplexType(String typeName)
  {
    if(typeName.equals("stock") || typeName.equals("bond") || 
typeName.equals("portfolio") || typeName.equals("array"))
      return true;
    else
      return false;
  }
  public static PSLComplexType createComplexType(String typeName, boolean
initializeValue)
  {
    PSLComplexType reference = new PSLComplexType(typeName);

    if (!PSLComplexType.isValidComplexType(typeName))
      throw new PSLException("Unknown complex type:" + typeName);

    if (initializeValue)
    {
      PSLComplexTypeValue complex;

      if(typeName.equals("stock"))
        complex = new PSLStockType();
      else if(typeName.equals("bond"))
        complex = new PSLBondType();
      else if(typeName.equals("portfolio"))
        complex = new PSLPortfolioType();
      else
        complex = new PSLArrayType();

      reference.setReferencedValue(complex);
    }

    return reference;
  }

  public PSLDataType copy()
  {
    if (this.value == null)
      throw new PSLException("Reference to " + this.typeName + " is not set.");

    PSLComplexType reference = new PSLComplexType(this.typeName);
    reference.setReferencedValue(this.value);

    return reference;
  }
}
```

A.10 PSLComplexTypeValue.java

```
package frontend;

import java.util.*;

public abstract class PSLComplexTypeValue
{
  private Hashtable Properties = new Hashtable();

  protected void defineProperty(String prop, PSLDataType value)
  {
    this.Properties.put(prop, value);
  }

  public void print()
  {
    Enumeration keys = this.Properties.keys();

    while (keys.hasMoreElements())
    {
      String key = (String) keys.nextElement();

      System.out.print(key + " = ");
      this.getProperty(key).print();
    }
  }

  public boolean hasProperty(String prop)
  {
    return this.Properties.containsKey(prop);
  }

  public PSLDataType getProperty(String prop)
  {
    if (!this.hasProperty(prop))
      throw new PSLException("Property " + prop + " undefined for " + " complex
type " + this.getTypeName());

    return (PSLDataType)this.Properties.get(prop);
  }

  public void setProperty(String prop, PSLDataType value)
  {
    assert(this.Properties.containsKey(prop));
    this.Properties.put(prop,value);
  }

  public abstract String getTypeName();
  public abstract ArrayList getMethods();

    public boolean hasMethod(String meth)
  {
    return this.getMethods().contains(meth);
  }

    public PSLDataType callMethod(String methodName, PSLDataType[] params)
  {
    if(!hasMethod(methodName))
      throw new PSLException(methodName+" is not a valid method name for type "+
this.getTypeName());
      return executeMethod(methodName,params);
  }
  protected abstract PSLDataType executeMethod(String methodName, PSLDataType[]
params);

  public static boolean isValidComplexType(String typeName)
  {
    if(typeName.equals("stock") || typeName.equals("bond") ||
typeName.equals("portfolio"))
      return true;
    else
      return false;
  }
}
```

A.11 PSLDataType.java

```
package frontend;

import java.io.*;
import java.util.ArrayList;

public abstract class PSLDataType
{
  String name;   // used in hash table
  private boolean islValue = false;

    public PSLDataType()
    {
    }

    public PSLDataType(String name)
    {
    this.name = name;
    }

    public void setName( String name )
    {
        this.name = name;
    }

    public boolean getIslValue()
  {
      return this.islValue;
    }

    public void setIslValue(boolean islValue)
  {
      this.islValue = islValue;
    }

    public abstract String getTypeName();

  public abstract PSLDataType copy();

    // By default, all the operations are illegal unless explictly defined in
individual definitions
    public PSLDataType error( String msg ) {
        throw new PSLException( "illegal operation: " + msg
                               + "( <" + this.getTypeName() + "> "
                               + ( name != null ? name : "<?>" )
                               + " )" );
    }

    public PSLDataType error( PSLDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new PSLException(
            "illegal operation: " + msg
            + "( <" + this.typeToString(this) + "> "
            + ( name != null ? name : "<?>" )
            + " and "
            + "<" + this.typeToString(b) + "> "
            + ( name != null ? name : "<?>" )
            + " )" );
    }

    private String typeToString(PSLDataType value)
  {
      if (value instanceof PSLFunctionArgument)
      {
        PSLFunctionArgument arg = (PSLFunctionArgument)value;
        if (arg.getIsArray())
        {
          return arg.getArgumentType() + "[" +
          (arg.getArraySize() == 0 ?
          "" :
          new Integer(arg.getArraySize()).toString())+ "]";
        }
      }

      if (value.getTypeName().equals("array"))
      {
        PSLArrayType array =
(PSLArrayType)((PSLComplexType)value).getReferencedValue();
        return array.getContainedTypeName() + "[" +
          (array.getArraySize() == 0 ?
            "" :
            new Integer(array.getArraySize()).toString())+ "]";
      }
      else
        return value.getTypeName();
    }

    public abstract void print();

    public PSLDataType assign( PSLDataType b ) {
        return error( b, "=" );
    }

    public PSLDataType uminus() {
        return error( "-" );
    }

    public PSLDataType add( PSLDataType b )
  {
        return error( b, "+" );
    }

    public PSLDataType sub( PSLDataType b )
  {
        return error( b, "-" );
    }

    public PSLDataType mul( PSLDataType b )
  {
        return error( b, "*" );
    }

    public PSLDataType div( PSLDataType b )
  {
        return error( b, "/" );
    }

    public PSLDataType mod( PSLDataType b )
  {
        return error( b, "%" );
    }


    public PSLDataType gt( PSLDataType b ) {
        return error( b, ">" );
    }
    public PSLDataType ge( PSLDataType b ) {
        return error( b, ">=" );
    }

    public PSLDataType lt( PSLDataType b ) {
        return error( b, "<" );
    }

    public PSLDataType le( PSLDataType b ) {
        return error( b, "<=" );
    }

    public PSLDataType eq( PSLDataType b ) {
        return error( b, "==" );
    }

    public PSLDataType ne( PSLDataType b ) {
        return error( b, "!=" );
    }

    public PSLDataType and( PSLDataType b ) {
        return error( b, "and" );
    }

    public PSLDataType or( PSLDataType b ) {
        return error( b, "or" );
    }

    public PSLDataType not() {
        return error( "not" );
    }

    public String toString()
    {
      error("toString() method not implemented for this class.");
      return null;
    }

    protected abstract void onAssignValue(PSLDataType value);

    public final void assignValue(PSLDataType value)
  {
      if (!this.getIslValue())
        throw new PSLException("left hand operand is not an l-value.");

      this.onAssignValue(value);
  }

}
```

A.12 PSLDebugger.java

```
package frontend;

public class PSLDebugger
{
  static boolean debug_tag = false;
  static boolean warn_tag = false;
  public static void info(String s)
  {
    if(debug_tag)
      System.out.println("[PSL DEBUG INFO]"+s);
  }
  public static void warn(String s)
  {
    if(warn_tag)
      System.err.println("[PSL WARNING MESSAGE]"+s);
  }

  public static void error(String s)
  {
    System.err.println("[PSL ERROR MESSAGE]"+s);
    throw new PSLException("PSL exited due to compilation errors.");
  }
}
```

A.13 PSLException.java

```java
package frontend;

class PSLException extends RuntimeException {
    PSLException( String msg ) {
        System.err.println( "PSL Interpreter Error: " + msg );
    }
}
```

A.14 PSLFunctionArgument.java

```java
package frontend;

public class PSLFunctionArgument extends PSLDataType
{
  private String arg_type;
  private String arg_name;
  private boolean isArray;
  private int arraySize;

  public PSLFunctionArgument(String type, String name, boolean isArray, int
arraySize)
  {
    this.arg_type = type;
    this.arg_name = name;
    this.isArray = isArray;
    this.arraySize = arraySize;
  }

  public boolean getIsArray()
  {
    return this.isArray;
  }

  public int getArraySize()
  {
    return this.arraySize;
  }

  public void verifyArgumentType(PSLDataType arg)
  {
    if (arg.getTypeName().equals("array"))
    {
      PSLArrayType array = (PSLArrayType)
((PSLComplexType)arg).getReferencedValue();
      if (array != null)
      {
        if (!this.isArray)
          this.error(arg, "incompatible function argument types.");
        else if ((!this.getTypeName().equals(array.getContainedTypeName())) ||
            (this.arraySize != 0 && this.arraySize != array.getArraySize()))
          this.error(arg, "array type does not match function declaration.");
      }
    }
    else if (!arg.getTypeName().equals(this.getArgumentType()) || this.isArray)
      this.error(arg, "incompatible function argument types.");
  }

  public String getTypeName()
  {
    return this.getArgumentType();
  }

  public String getArgumentType()
  {
    return arg_type;
  }

  public String getArgumentName()
  {
    return arg_name;
  }
  public PSLDataType copy()
  {
    return new PSLFunctionArgument(this.arg_type, this.arg_name, this.isArray,
this.arraySize);
  }

  public void print() {
    System.out.println("Argument Type:  "+arg_type);
    System.out.println("Argument Name:  "+arg_name);
  }

  protected void onAssignValue(PSLDataType value)
  {
    throw new PSLException("function arguments do not support onAssignValue");
  }
}
```

A.15 PSLFunctionType.java

```java
package frontend;

import antlr.collections.AST;
import java.util.ArrayList;

public class PSLFunctionType extends PSLDataType
{
  private PSLFunctionArgument[] args = new PSLFunctionArgument[0];
  private AST body;            // body = null means an internal function.
  private PSLSymbolTable pst;   // the symbol table of static parent
  private int id;              // for internal functions only

  public PSLFunctionType( String name, PSLDataType[] argss,
                 AST body, PSLSymbolTable pst)
  {
    super( name );

    this.args = new PSLFunctionArgument[argss.length];

    for(int i=0; i<argss.length; i++)
    {
      if(!(argss[i] instanceof PSLFunctionArgument))
      {
        throw new PSLException("Invalid argument definition in
function:"+argss[i]);
      }
      else
      {
        PSLFunctionArgument pf = (PSLFunctionArgument)argss[i];
        this.args[i] = (PSLFunctionArgument)pf.copy();
      }
    }
    this.body = body;
    this.pst = pst;
  }

  public PSLFunctionType( String name, int id ) {
    super( name );
    this.args = null;
    this.id = id;
    pst = null;
    body = null;
  }

  public final boolean isInternal() {
    return body == null;
  }

  public final int getInternalId()
  {
    return id;
  }

  public final void setInternalId(int id)
  {
    this.id = id;
  }

  public String getTypeName() {
    return "function";
  }
  public ArrayList getMethods()
  {
    throw new PSLException("functions do not support methods");
  }

  protected PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    throw new PSLException("cannot execute methods on functions");
  }

  public PSLDataType copy() {
    return new PSLFunctionType( name, args, body, pst);
  }

  public void print() {
    if ( body == null )
    {
      System.out.println( name + " = <internal-function> #" + id );
    }
    else
    {
      if ( name != null )
        System.out.print( name + " = " );
      System.out.print( "<function>(" );
      for ( int i=0; i<this.args.length; i++ )
      {
        args[i].print();
        System.out.print( "," );
      }
      System.out.println( ")" );
    }
  }
  public PSLFunctionArgument[] getArgs() {
    return this.args;
  }

  public PSLSymbolTable getParentSymbolTable() {
    return pst;
  }

  public AST getBody() {
    return body;
  }

  protected void onAssignValue(PSLDataType value)
  {
    throw new PSLException("functins do not support onAssignValue");
  }
}
```

A.16 PSLInt.java

```
package frontend;

import java.io.*;

class PSLInt extends PSLPrimitiveType
{
    int value;

    public PSLInt( int x )
    {
        value = x;
    }

    public PSLInt(String x)
    {
      this(Integer.parseInt(x));
    }

    public int getValue()
    {
      return value;
    }

    public String getTypeName()
    {
        return "int";
    }

    public String toString()
    {
      return new String(""+value);
    }
    public static int intValue( PSLDataType b ) {
        if ( b instanceof PSLReal )
            return (int) ((PSLReal)b).getValue();
        if ( b instanceof PSLInt )
            return ((PSLInt)b).value;
        b.error( "cast to int" );
        return 0;
    }

    public void print() {
        System.out.println( Integer.toString( value ) );
    }

    public PSLDataType uminus() {
        return new PSLInt( -value );
    }

    public PSLDataType add( PSLDataType b )
  {
        if ( b instanceof PSLInt )
            return new PSLInt( value + ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLReal(this.getValue() + ((PSLReal)b).getValue());
        else if (b instanceof PSLString)
          return new PSLString(this.getValue() + ((PSLString)b).getValue());
        else
          return this.error(b, "+");
    }
  public PSLDataType sub( PSLDataType b )
{
      if ( b instanceof PSLInt )
          return new PSLInt( value - ((PSLInt)b).getValue());
      else if (b instanceof PSLReal)
        return new PSLReal(this.getValue() - ((PSLReal)b).getValue());
      else
        return this.error(b, "-");
  }


  public PSLDataType mul( PSLDataType b )
{
      if ( b instanceof PSLInt )
          return new PSLInt( value * ((PSLInt)b).getValue());
      else if (b instanceof PSLReal)
        return new PSLReal(this.getValue() * ((PSLReal)b).getValue());
      else
        return this.error(b, "*");
  }


  public PSLDataType div( PSLDataType b )
{
      if ( b instanceof PSLInt )
          return new PSLInt( value / ((PSLInt)b).getValue());
      else if (b instanceof PSLReal)
        return new PSLReal(this.getValue() / ((PSLReal)b).getValue());
      else
        return this.error(b, "/");
  }

  public PSLDataType gt( PSLDataType b )
{
      if ( b instanceof PSLInt )
          return new PSLBool( value > intValue(b) );
      return b.lt( this );
  }

  public PSLDataType ge( PSLDataType b ) {
      if ( b instanceof PSLInt )
          return new PSLBool( value >= intValue(b) );
      return b.le( this );
  }

  public PSLDataType lt( PSLDataType b ) {
      if ( b instanceof PSLInt )
          return new PSLBool( value < intValue(b) );
      return b.gt( this );
  }

  public PSLDataType le( PSLDataType b ) {
      if ( b instanceof PSLInt )
          return new PSLBool( value <= intValue(b) );
      return b.ge( this );
  }

  public PSLDataType eq( PSLDataType b ) {
      if ( b instanceof PSLInt )
          return new PSLBool( value == intValue(b) );
      return b.eq( this );
  }

  public PSLDataType ne( PSLDataType b ) {
        if ( b instanceof PSLInt )
            return new PSLBool( value != intValue(b) );
        return b.ne( this );
    }

    protected void onAssignValue(PSLDataType value)
    {
      if(!(value instanceof PSLInt))
        this.error(value, "incompatible types");

      this.value = ((PSLInt)value).value;
    }

   public PSLDataType copy()
   {
     return new PSLInt(this.value);
   }
}
```

A.17 PSLInterpreter.java

```
package frontend;

import java.util.*;

import antlr.collections.AST;

public class PSLInterpreter
{
  private PSLSymbolTable symbolTable;
  final static int at_none = 0;
  final static int at_break = 1;
  final static int at_continue = 2;
  final static int at_return = 3;
  String label;
  public static Random rand  = new Random(); // Gaussian random variable output
  private int control = at_none;

  public PSLInterpreter()
  {
    this.symbolTable = new PSLSymbolTable(null);
  }
  public PSLDataType[] vecToArray(Vector v)
  {
    PSLDataType[]  vlist = new PSLDataType[v.size()];

    for( int i=0; i<v.size(); ++i )
    {
      vlist[i] = (PSLDataType)v.elementAt(i);
    }
    return vlist;
  }

  public void EnterNewScope()
  {
    this.symbolTable = new PSLSymbolTable(this.symbolTable);
  }

  public void ExitCurrentScope()
  {
    this.symbolTable = this.symbolTable.parent();
  }

  public void setReturn(PSLDataType return_value)
  {
    symbolTable.setReturnValue(return_value);
    control = at_return;
  }

  public void setBreak()
  {
    control = at_break;
  }

  public void setContinue()
  {
    control = at_continue;
  }

  public void reset()
  {
    control = at_none;
  }
  public PSLPrimitiveType registerPrimitiveType(String typeName, String varName)
  {
    if(!PSLPrimitiveType.isValidPrimitiveType(typeName))
      throw new PSLException("Invalid Primitive Type: "+typeName);

    PSLPrimitiveType pt =  PSLPrimitiveType.createPrimitive(typeName);
    if(symbolTable.containsVar(varName))
      {
      PSLDebugger.error("PSL interpreter error: variable "+varName+" has already
been defined.");
      throw new PSLException("");
    }
    symbolTable.setValue(varName, pt);

    return pt;
  }

  public PSLComplexType registerComplexType(String typeName, String varName)
  {
    PSLComplexType ct = PSLComplexType.createComplexType(typeName, true);
    symbolTable.setValue(varName, ct);

    return ct;
  }

  public PSLComplexType registerArray(String arrayName, String arrayType,
PSLDataType size)
  {
    PSLComplexType ct = PSLComplexType.createComplexType("array", true);
    PSLArrayType array = (PSLArrayType) ct.getReferencedValue();
    array.setContainedTypeName(arrayType);

    if (size instanceof PSLInt)
      array.setArraySize(((PSLInt)size).getValue());
    else if (size != null)
      throw new PSLException("Array size must be an integer.");

    symbolTable.setValue(arrayName, ct);
    return ct;
  }

  public PSLObjectType getObject(String objname)
  {
    return (PSLObjectType)(symbolTable.getValue(objname));
  }

  public boolean isInternal(String function_name)   // Xin
  {
    if(function_name.equals("random"))  // random number generator
    {
      return true;
    }
    else if(function_name.equals("loadData")) // load data into memory
    {
      return true;
    }
    else if(function_name.equals("saveData")) // write data to disk
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  public boolean functionDefined(String function_name)  // Xin
  {
    // Internal functions
    if(isInternal(function_name))
    {
      return true;
    }
    else
    {
      //     should look up the symble table
      return symbolTable.functionDefined(function_name);
    }
    //return false;
  }

  public PSLDataType execInternal(String function_name, PSLDataType[] params)
  {
//     Internal functions
    if(function_name.equals("random"))  // random number generator
    {
      if(params != null)
      {
        throw new PSLException("Internal function random() does not accept any
argument.");
      }
      double value = rand.nextDouble();
      return new PSLReal(value);
    }
    else if(function_name.equals("loadData")) // load data into memory
    {
      if(params.length != 1)
      {
        throw new PSLException("Internal function loadData(string) only accept 1
argument.");
      }
      if(!(params[0] instanceof PSLString))
      {
        throw new PSLException("Internal function laodData(string): file name must
be a string.");
      }
      String filename = ((PSLString)params[0]).getValue();
      double[] data = backend.DataLoader.load(filename);

      PSLArrayType dataArray = new PSLArrayType();
          dataArray.setContainedTypeName("real");
          dataArray.setArraySize(data.length);

          for(int i=0; i<data.length; i++)
          {
            System.out.println("data:  "+data[i]);
            dataArray.setItem(i,new PSLReal(data[i]));
          }

          PSLComplexType arrayRef = new PSLComplexType("array");
          arrayRef.setReferencedValue(dataArray);
      return arrayRef;
    }
    else if(function_name.equals("saveData")) // write data to disk
    {
      if(params.length != 2)
      {
        throw new PSLException("Internal function saveData(real[],string) accept
2 arguments.");
      }
      if(!(params[0] instanceof PSLComplexType))
      {
        throw new PSLException("Internal function saveData(real[],string): first
arguement must be an array.");
      }

      if(!(params[1] instanceof PSLString))
      {
        throw new PSLException("Internal function laodData(string): file name must
be a string.");
      }
      String filename = ((PSLString)params[1]).getValue();

      PSLComplexType ref = (PSLComplexType)(params[0]);
      PSLArrayType S = (PSLArrayType)ref.getReferencedValue();
      double[] data = new double[S.getArraySize()];

        for(int i=0; i<data.length; i++)
        {
          data[i] = ((PSLReal)(S.getItem(i))).getValue();
        }

      backend.DataWriter.write(data,filename);
      return null;

    }
    else
    {
      throw new PSLException("Unexpected internal function name:"+function_name);
    }
  }

  public void registerFunction(String fname, PSLDataType[] args_decl, AST
stmt_body)
  {
    PSLDebugger.info("Register Function: "+fname);
    if(functionDefined(fname))   // Xin added: prevent function redeclaration
    {
      throw new PSLException("Function:"+fname+" has already been defined");
    }
    PSLSymbolTable pst = new PSLSymbolTable(symbolTable);
    PSLFunctionType function = new PSLFunctionType(fname, args_decl,
              stmt_body, pst);

    symbolTable.setValue(fname, function);
  }
  public PSLDataType InvokeFunction(PSLWalker walker, String function_name,
PSLDataType[] params) throws antlr.RecognitionException
  {
    //PSLDebugger.info("Invoking function: "+function_name);
    //     func must be an existing function
    if(!functionDefined(function_name))
    {
      throw new PSLException(function_name+" is not a function!");
    }

    // Is this function an internal function?
      // Names of formal args are not necessary for internal functions.
    if (isInternal(function_name))
        {
            PSLDataType r = execInternal( function_name,
                                 params);
            return r;
        }
    PSLFunctionType func = (PSLFunctionType)(symbolTable.get(function_name));
```

```
        // check numbers of actual and formal arguments
        PSLFunctionArgument[] args = ((PSLFunctionType)func).getArgs();

        if ( args.length != params.length )
            throw new PSLException( "Unmatched number of parameters in " +
func.name + " call." );


        PSLDebugger.info("External function");

        // Create a new symbol table
        // And make it top level to avoid seeing the scope in the main program
        PSLSymbolTable oldSymbolTable = symbolTable;
        symbolTable = new PSLSymbolTable(null);

        // assign actual parameters to formal arguments
        for ( int i=0; i<args.length; i++ )
        {
            PSLDataType d = params[i];
            args[i].verifyArgumentType(d);
            d.setName( args[i].getArgumentName() );
            symbolTable.setValue( args[i].getArgumentName(),d);
        }

        // call the function body
        PSLDebugger.info("Executing function body:");
        //PSLDataType[] exprlist =
walker.expr_list( ((PSLFunctionType)func).getBody() );

        // set return value to the last expression of the function
        PSLDataType r = null;
        walker.expr( ((PSLFunctionType)func).getBody() );

        //PSLDataType r = exprlist[exprlist.length-1];


        // no break or continue can go through the function
        if ( control == at_break || control == at_continue )
            throw new PSLException( "nowhere to break or continue" );

        // if a return was called
        if ( control == at_return )
        {
            tryResetFlowControl( ((PSLFunctionType)func).name );
            r = symbolTable.getReturnValue();
      //        PSLDebugger.info("return value type:"+r.getTypeName());
        }
        // remove this symbol table and recover old symbol table
        //symbolTable = symbolTable.parent();
        symbolTable = oldSymbolTable;

        return r;
  }
  public void tryResetFlowControl(String loop_label)
  {
     if ( null == label || label.equals( loop_label ) )
            control = at_none;

  }

  public void executeFor(PSLWalker walker,AST init, AST condition, AST update, AST
for_body) throws antlr.RecognitionException
  {
        //  create a new symbol table
```

```
    this.EnterNewScope();
        PSLDebugger.info("Executing for statements ...");

        walker.expr(init);

        PSLDataType conditionBool = walker.expr(condition);

        if(conditionBool == null)
        {
          while(true)
          {
            walker.expr(for_body);
            if(isAtBreak())  break;
            reset();
            walker.expr(update);
          }
        }

        else
        {
          if (!(conditionBool instanceof PSLBool))
            throw new PSLException("Invalid conditional statement in <for>:");

          while(((PSLBool)(conditionBool)).getValue())
          {
            walker.expr(for_body);
            if(isAtBreak())  break;
            reset();
            walker.expr(update);

            conditionBool = walker.expr(condition);
          }
        }
    reset();
    this.ExitCurrentScope();
  }

  public void executeWhile(PSLWalker walker, AST condition, AST body) throws
antlr.RecognitionException
  {
    this.EnterNewScope();
        PSLDebugger.info("Executing while statements ...");
    while(((PSLBool)(walker.expr(condition))).getValue())
    {
      //PSLDebugger.info(body.getText());
      walker.expr(body);
      if(isAtBreak())  break;
      reset();
    }
    reset();
    this.ExitCurrentScope();
  }

  public boolean forCanProceed(PSLDataType[] init, PSLDataType cond, PSLInt[]
values)
  {
    return false;
  }

  public boolean isAtBreak()
  {
    return control==at_break;
  }

  public boolean isAtContinue()
  {
    return control==at_continue;
  }
  public boolean isAtReturn()
  {
    return control==at_return;
  }

  public boolean canProceed()
  {
    return control == at_none;
  }
}
```

A.18 PSLMain.java

```java
package frontend;

import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

class PSLMain
{
  public static void execFile( String filename )
    {
        try
        {
            InputStream input = ( null != filename ) ?
                (InputStream) new FileInputStream( filename ) :
                (InputStream) System.in;

            PSLLexer lexer = new PSLLexer( input );
            PSLParser parser = new PSLParser( lexer );

            // Parse the input program
            parser.statement_list();

            CommonAST tree = (CommonAST)parser.getAST();

            PSLWalker walker = new PSLWalker();
            // Traverse the tree created by the parser

            walker.expr( tree );
        }
        catch( IOException e )
    {
            System.err.println( "Error: I/O: " + e );
        }
        catch( RecognitionException e )
    {
            System.err.println( "Error: Recognition: " + e );
        }
        catch( TokenStreamException e )
    {
            System.err.println( "Error: Token stream: " + e );
        }
        catch( Exception e )
    {
            System.err.println( "Error: " + e );
        }
    }

    public static void main( String[] args )
    {
        if ( args.length == 1)
            execFile( args[args.length-1] );
        else
          execFile(null);

        System.exit( 0 );
    }
}
```

A.19 PSLObjectType.java

```
package frontend;

import java.util.*;

public abstract class PSLObjectType extends PSLDataType
{
  public abstract ArrayList getMethods();

    public boolean hasMethod(String meth)
  {
    return this.getMethods().contains(meth);
  }

    public PSLDataType callMethod(String methodName, PSLDataType[] params)
  {
    if(!hasMethod(methodName))
      throw new PSLException(methodName+" is not a valid method name for type "+
this.getTypeName());
      return executeMethod(methodName,params);
  }

  protected abstract PSLDataType executeMethod(String methodName, PSLDataType[]
params);
}
```

A.20 PSLPortfolio.java

```java
/**
 * @author xinli
 *
 *  wrapper class for portfolio
 */
package frontend;

import java.util.*;

import backend.DataPlotter;
import backend.PrintFormat;

public class PSLPortfolioType extends PSLComplexTypeValue
{
  public static final int optimize = 1;
  public static final int maxStock = 10;
  public static final int maxBond = 10;

  public double[][] varianceCovarianceMatrix = new double[maxStock][maxStock];
  private Vector numStock = new Vector();
  private Vector numBond = new Vector();


  Vector stockSet = new Vector();
  Vector bondSet = new Vector();

  public PSLPortfolioType()
  {
    this.defineProperty("Name", new PSLString(""));
    this.defineProperty("Capital", new PSLReal(0));
  }

  public String getTypeName()
  {
    return "portfolio";
  }

  public ArrayList getMethods()
  {
    ArrayList list = new ArrayList();

    list.add("addStock");
    list.add("addBond");
    list.add("addCapital");
    list.add("setCov");
    list.add("getCov");
    list.add("getTotalAsset");
    list.add("printContent");
    list.add("clearPortfolio");
    list.add("isLeverage");
    list.add("simulate");

    return list;
  }

  protected PSLDataType executeMethod(String methodName,PSLDataType[] params)
  {
    if(methodName.equals("addStock"))   // add stocks to the portfolio
    {
      if(params.length != 2)
      {
          throw new PSLException("portfolio.addStock(stock,int): 2 arguments
expected.");
      }
      if(!(params[0] instanceof PSLComplexType))
      {
        throw new PSLException("portfolio.addStock(stock,int): first argument
must be a stock.");
      }
      if(!(params[1] instanceof PSLInt))
      {
        throw new PSLException("portfolio.addStock(stock,int): second argument
must be an integer.");
      }

      int nStock = ((PSLInt)(params[1])).getValue();
      if(nStock <=0)
      {
        throw new PSLException("portfolio.addStock(stock,int): second argument
must be positive.");
      }
      PSLComplexType ref = (PSLComplexType)(params[0]);
      if(!(ref.typeName.equals("stock")))
        {
        throw new PSLException("portfolio.addStock(stock): argument must be a
stock.");
        }
      PSLStockType S = (PSLStockType)(ref.getReferencedValue());

      boolean already = false;
      for(int i=0; i<stockSet.size(); i++)
      {
        PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
        if(st.getProperty("Name").equals(S.getProperty("Name")))
        {
          // this stock is already contained in the portfolio
          // so only needs to increment its quantity
          already = true;
          int num = ((Integer)(numStock.elementAt(i))).intValue();
          num += nStock;
          numStock.setElementAt(new Integer(num), i);
          break;
        }
      }
      if(!already)   // new stock
      {
        if(stockSet.size() == maxStock)
        {
          throw new PSLException("portfolio already has too many stock types.");
        }

        stockSet.addElement(S);
        numStock.addElement(new Integer(nStock));
      }
//       substract the corresponding cash amount of the stock price
      double cash = ((PSLReal)getProperty("Capital")).getValue();
      cash -= nStock*((PSLReal)(S.getProperty("Price"))).getValue();
      this.setProperty("Capital", new PSLReal(cash));
      return null;
    }

    else if(methodName.equals("addBond"))
    {
      if(params.length != 2)
      {
          throw new PSLException("portfolio.addBond(bond,int): 2 argument
expected.");
```

```
      }
      if(!(params[0] instanceof PSLComplexType))
      {
        throw new PSLException("portfolio.addBond(bond,int): first argument must
be a bond.");
      }

      if(!(params[1] instanceof PSLInt))
      {
        throw new PSLException("portfolio.addBond(bond,int): second argument must
be an integer.");
      }

      int nBond = ((PSLInt)(params[1])).getValue();
      if(nBond <=0)
      {
        throw new PSLException("portfolio.addBond(bond,int): second argument must
be positive.");
      }
      PSLComplexType ref = (PSLComplexType)(params[0]);
      if(!(ref.typeName.equals("bond")))
        {
        throw new PSLException("portfolio.addStock(stock): argument must be a
bond.");
        }
      PSLBondType B = (PSLBondType)(ref.getReferencedValue());

      boolean already = false;
      for(int i=0; i<bondSet.size(); i++)
      {
        PSLBondType bd= (PSLBondType)(bondSet.elementAt(i));
        if(bd.getProperty("Name").equals(B.getProperty("Name")))
        {
          // this bond is already contained in the portfolio
          // so only needs to increment its quantity
          already = true;
          int num = ((Integer)(numBond.elementAt(i))).intValue();
          num += nBond;
          numBond.setElementAt(new Integer(num), i);
          break;
        }
      }
      if(!already)   // new bond
      {
        if(bondSet.size() == maxBond)
        {
          throw new PSLException("portfolio already has too many bond types.");
        }

        bondSet.addElement(B);
        numBond.addElement(new Integer(nBond));
      }
//        substract the corresponding cash amount of the stock price
      double cash = ((PSLReal)getProperty("Capital")).getValue();
      cash -= nBond*(B.calculatePrice());
      this.setProperty("Capital", new PSLReal(cash));
      return null;
    }

    // Add cash to the total capital
    else if(methodName.equals("addCapital"))
    {
      if(params.length != 1)
      {
          throw new PSLException("portfolio.addCapital(real): 1 argument
expected.");
      }
      if(!(params[0] instanceof PSLReal))
      {
        throw new PSLException("portfolio.addCapital(real): argument must be a
real.");
      }

      double additional_cash = ((PSLReal)params[0]).getValue();
      double cash = ((PSLReal)getProperty("Capital")).getValue();
      cash += additional_cash;

      this.setProperty("Capital", new PSLReal(cash));
      return null;
    }

    else if(methodName.equals("setCov"))
    {
      if(params.length != 3)
      {
          throw new PSLException("portfolio.setCov(string,string,real): 3
arguments expected.");
      }
      if(!(params[0] instanceof PSLString))
      {
        throw new PSLException("portfolio.setCov(string,string,real): first
argument must be a string.");
      }
      if(!(params[1] instanceof PSLString))
      {
        throw new PSLException("portfolio.setCov(string,string,real): second
argument must be a string.");
      }
      if(!(params[2] instanceof PSLReal))
      {
        throw new PSLException("portfolio.setCov(string,string,real): third
argument must be real.");
      }

      boolean found;
      int   stock_index1=-1,stock_index2=-1;

      String stock1 = ((PSLString)params[0]).getValue();
      found = false;
      for(int i=0; i<stockSet.size(); i++)
      {
        PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
        if(stock1.equals(((PSLString)(st.getProperty("Name"))).getValue()))
        {
          found = true;
          stock_index1=i;
          break;
        }
      }
      if(!found)
      {
        throw new PSLException("portfolio.setCov(string,string,real): stock name
"+stock1+" not found.");
      }

      String stock2 = ((PSLString)params[1]).getValue();
      found = false;
      for(int i=0; i<stockSet.size(); i++)
      {
        PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
        if(stock2.equals(((PSLString)(st.getProperty("Name"))).getValue()))
        {
          found = true;
          stock_index2=i;
          break;
        }
      }
      if(!found)
      {
        throw new PSLException("portfolio.setCov(string,string,real): stock name
"+stock1+" not found.");
      }
      double covariance = ((PSLReal)params[2]).getValue();
      if(covariance<0 || covariance>1)
      {
        throw new PSLException("portfolio.setCov(string,string,real): third
argument(covariance) must be between 0 and 1.");
      }
      varianceCovarianceMatrix[stock_index1][stock_index2]=covariance;
      varianceCovarianceMatrix[stock_index2][stock_index1]=covariance;
      return null;
    }

    else if(methodName.equals("getCov"))
    {
      if(params.length != 2)
      {
          throw new PSLException("portfolio.getCov(string,string): 2 arguments
expected.");
      }
      if(!(params[0] instanceof PSLString))
      {
        throw new PSLException("portfolio.getCov(string,string): first argument
must be a string.");
      }
      if(!(params[1] instanceof PSLString))
      {
        throw new PSLException("portfolio.getCov(string,string): second argument
must be a string.");
      }


      boolean found;
      int   stock_index1=-1,stock_index2=-1;

      String stock1 = ((PSLString)params[0]).getValue();
      found = false;
      for(int i=0; i<stockSet.size(); i++)
      {
        PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
        if(stock1.equals(((PSLString)(st.getProperty("Name"))).getValue()))
        {
          found = true;
          stock_index1=i;
          break;
        }
      }
      if(!found)
      {
        throw new PSLException("portfolio.setCov(string,string,real): stock name
"+stock1+" not found.");
      }

      String stock2 = ((PSLString)params[1]).getValue();
      found = false;
```

```
        for(int i=0; i<stockSet.size(); i++)
        {
          PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
          if(stock2.equals(((PSLString)(st.getProperty("Name"))).getValue()))
          {
            found = true;
            stock_index2=i;
            break;
          }
        }
        if(!found)
        {
          throw new PSLException("portfolio.setCov(string,string,real): stock name
"+stock1+" not found.");
        }
        double covariance = varianceCovarianceMatrix[stock_index1][stock_index2];

        return new PSLReal(covariance);
      }


      else if(methodName.equals("printContent"))
      {
        if(params != null)
        {
            throw new PSLException("portfolio.printContent() does not accept any
argument.");
        }
        System.out.println("===============================================");
        System.out.println("    Content of Portfolio:
"+((PSLString)(getProperty("Name"))).getValue());
        System.out.println(" ---------------------------------------------");
        double totalAsset = 0.0;
        for(int i=0; i<stockSet.size(); i++)
        {
          PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
          String st_name = ((PSLString)(st.getProperty("Name"))).getValue();
          int st_num = ((Integer)numStock.elementAt(i)).intValue();
          double price = ((PSLReal)st.getProperty("Price")).getValue();
          totalAsset += price*st_num;
               System.out.print("  STOCK      ");
               backend.PrintFormat.Print_Format(st_name,18);
               backend.PrintFormat.Print_Format(st_num,18);
               System.out.println();
        }
        for(int i=0; i<bondSet.size(); i++)
        {
          PSLBondType bd= (PSLBondType)(bondSet.elementAt(i));
          String bd_name = ((PSLString)(bd.getProperty("Name"))).getValue();
          int bd_num = ((Integer)numBond.elementAt(i)).intValue();
          double price = bd.calculatePrice();
          totalAsset += price*bd_num;
          System.out.print("  BOND       ");
          backend.PrintFormat.Print_Format(bd_name,18);
               backend.PrintFormat.Print_Format(bd_num,18);
               System.out.println();
        }
        double cash = ((PSLReal)getProperty("Capital")).getValue();
        totalAsset += cash;
        System.out.print("  CASH       ");
        backend.PrintFormat.Print_Format(" ",18);
        backend.PrintFormat.Print_Format(cash,18);
        if(cash<0)
        {
          System.out.println("(DEBT)");
```

```
    }
    else
    {
      System.out.println();
    }

    System.out.println(" ---------------------------------------------");

    System.out.print("  TOTAL ASSET                  ");
    backend.PrintFormat.Print_Format(totalAsset, 18);
    System.out.println();
    System.out.println("===============================================");
    return null;
  }
  else if(methodName.equals("getTotalAsset"))
  {
    double totalAsset = 0.0;
    for(int i=0; i<stockSet.size(); i++)
    {
      PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
      double price = ((PSLReal)st.getProperty("Price")).getValue();
      int st_num = ((Integer)numStock.elementAt(i)).intValue();
           totalAsset += price*st_num;
    }
    for(int i=0; i<bondSet.size(); i++)
    {
      PSLBondType bd= (PSLBondType)(bondSet.elementAt(i));
      double price = bd.calculatePrice();
      int bd_num = ((Integer)numBond.elementAt(i)).intValue();
           totalAsset += price*bd_num;
    }
    double cash = ((PSLReal)getProperty("Capital")).getValue();
    totalAsset += cash;
    return new PSLReal(totalAsset);
  }

  // Clear the portfolio up, i.e., sell all the non-cash assets
  else if(methodName.equals("clearPortfolio"))
  {
    double totalAsset = 0.0;
    for(int i=0; i<stockSet.size(); i++)
    {
      PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
      double price = ((PSLReal)st.getProperty("Price")).getValue();
      int st_num = ((Integer)numStock.elementAt(i)).intValue();
           totalAsset += price*st_num;
    }
    for(int i=0; i<bondSet.size(); i++)
    {
      PSLBondType bd= (PSLBondType)(bondSet.elementAt(i));
      double price = bd.calculatePrice();
      int bd_num = ((Integer)numBond.elementAt(i)).intValue();
           totalAsset += price*bd_num;
    }
    double cash = ((PSLReal)getProperty("Capital")).getValue();
    totalAsset += cash;

    this.setProperty("Capital",new PSLReal(totalAsset));
      numStock.clear();
      numBond.clear();
      stockSet.clear();
      bondSet.clear();
  }
  else if(methodName.equals("isLeverage"))
  {
      double cash = ((PSLReal)getProperty("Capital")).getValue();
      return new PSLBool(cash<0);
    }

      else if(methodName.equals("simulate"))
      {
        if(params.length != 1)
      {
          throw new PSLException("portfolio::simulate(int): 1 argument
expected.");
      }
      if(!(params[0] instanceof PSLInt))
      {
        throw new PSLException("portfolio::simulate(int): 1 argument must be of
real type.");
      }

      int steps = ((PSLInt)params[0]).getValue();
      if(steps <= 0)
      {
        throw new PSLException("stock::simulate(int): 1 argument must be
positive.");
      }

          List Data = new ArrayList();
          List ValueName = new ArrayList();

          double cash = ((PSLReal)getProperty("Capital")).getValue();

          double[] assetValue = new double[steps];

          for(int i=0; i<steps; i++)
          {
            assetValue[i] = cash;
          }

          Data.add(assetValue);
      ValueName.add("Portflio: "+((PSLString)getProperty("Name")).getValue());

      for(int i=0; i<stockSet.size(); i++)
      {
        PSLStockType st= (PSLStockType)(stockSet.elementAt(i));
        double price = ((PSLReal)(st.getProperty("Price"))).getValue();
        int st_num = ((Integer)numStock.elementAt(i)).intValue();
        double mu = ((PSLReal)(st.getProperty("Return"))).getValue()/100.0;
        double sigma = ((PSLReal)st.getProperty("Volatility")).getValue()/100.0;
        double[] simStockData = backend.GeoBrownMove.simulate(price, steps, mu,
sigma);

        for(int j=0; j<steps; j++)
        {
          simStockData[j] *= (double)st_num;
          assetValue[j] += simStockData[j];
        }

        Data.add(simStockData);
        ValueName.add("
Stock---"+((PSLString)st.getProperty("Name")).getValue());
      }
      for(int i=0; i< bondSet.size(); i++)
      {
        PSLBondType bt= (PSLBondType)(bondSet.elementAt(i));
        int Maturity = ((PSLInt)bt.getProperty("Maturity")).getValue();
        int bt_num = ((Integer)numBond.elementAt(i)).intValue();
        double coupon = ((PSLReal)bt.getProperty("Coupon")).getValue();
        double[] simbondData = new double[steps];

        for(int j=0; j<steps; j++)
        {
          if(j < Maturity)
          {
            simbondData[j] = j*coupon;
          }
          else
          {
            simbondData[j] = Maturity*coupon + 100.0;
          }

        }
        double price = bt.calculatePrice();
        for(int j=0; j<steps; j++)
        {
          simbondData[j] *= (double)bt_num;
          //assetValue[j] += simbondData[j];  // wrong
          assetValue[j] += price*(double)bt_num;
        }

        Data.add(simbondData);

        System.out.println("==============================================");
            System.out.println("Portfolio Simulation");
            System.out.println("Name:
"+((PSLString)getProperty("Name")).getValue());

        System.out.println("Time Steps: "+simbondData.length);
            System.out.println("     Step      Capital Gain($)");

System.out.println("----------------------------------------------");

            for(int k=0; k<assetValue.length; k++)
            {
              System.out.print("     ");
              PrintFormat.Print_Format(k+1,20);
              PrintFormat.Print_Format(assetValue[k],15);
              System.out.println();
            }
        ValueName.add("
Bond---"+((PSLString)bt.getProperty("Name")).getValue());
      }

          DataPlotter.drawPlot(Data, ValueName, "Portfolio Simulation") ;

System.out.println("==============================================");

          return null;
      }
      else
      {
        throw new PSLException("portfolio." + methodName + " not implemented.");
      }
    return null;
  }
}
```

A.21 PSLPrimtiveType.java

```java
package frontend;

import java.util.ArrayList;

public abstract class PSLPrimitiveType extends PSLObjectType
{
  public ArrayList getMethods()
  {
    ArrayList list = new ArrayList();
    list.add("print");

    return list;
  }

  protected PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    if (params == null || params.length == 0)
    {
      this.print();
      return null;
    }
    else
      throw new PSLException("print does not accept any arguments for type:"+
this.getTypeName());
  }

  public static boolean isValidPrimitiveType(String typeName)
  {
    if(typeName.equals("int") || typeName.equals("real") ||
typeName.equals("string") || typeName.equals("boolean"))
      return true;
    else
      return false;
  }

  public static PSLPrimitiveType createPrimitive(String typeName)
  {
    if(typeName.equals("int"))
      return new PSLInt(0);
    else if(typeName.equals("real"))
      return new PSLReal(0.0);
    else if(typeName.equals("string"))
      return new PSLString("");
    else if(typeName.equals("boolean"))
      return new PSLBool(false);
    else
      throw new PSLException("No such primitive type:");
  }
}
```

A.22 PSLReal.java

```
package frontend;

import java.io.*;

class PSLReal extends PSLPrimitiveType {
    double value;
    //boolean doublePrecision = true;

    public PSLReal( double x )
    {
        value = x;
    }

    public PSLReal( String x )
    {
      this(Double.parseDouble(x));
    }

    public String getTypeName()
    {
        return "real";
    }

    public PSLDataType copy()
    {
        return new PSLReal( value );
    }

    public double getValue()
    {
      return value;
    }

    public static double doubleValue( PSLDataType b ) {
        if ( b instanceof PSLReal )
            return ((PSLReal)b).value;
        if ( b instanceof PSLInt )
            return (double) ((PSLInt)b).value;
        b.error( "cast to real" );
        return 0;
    }

    public void print( ) {
        System.out.println( Double.toString( value ) );
    }

    public PSLDataType uminus() {
        return new PSLReal( -value );
    }

    public PSLDataType add( PSLDataType b )
  {
        if ( b instanceof PSLInt )
            return new PSLReal(this.getValue() + ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLReal(this.getValue() + ((PSLReal)b).getValue());
        else if (b instanceof PSLString)
          return new PSLString(this.getValue() + ((PSLString)b).getValue());
        else
          return this.error(b, "+");
    }
    public PSLDataType sub( PSLDataType b )
  {
      if ( b instanceof PSLInt )
            return new PSLReal(this.getValue() - ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLReal(this.getValue() - ((PSLReal)b).getValue());
        else
          return this.error(b, "-");
    }


    public PSLDataType mul( PSLDataType b )
  {
      if ( b instanceof PSLInt )
            return new PSLReal(this.getValue() * ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLReal(this.getValue() * ((PSLReal)b).getValue());
        else
          return this.error(b, "*");
    }


    public PSLDataType div( PSLDataType b )
  {
      if ( b instanceof PSLInt )
            return new PSLReal(this.getValue() / ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLReal(this.getValue() / ((PSLReal)b).getValue());
        else
          return this.error(b, "/");
    }

    public PSLDataType gt( PSLDataType b ) {
        return new PSLBool( value > doubleValue(b) );
    }

    public PSLDataType ge( PSLDataType b ) {
        return new PSLBool( value >= doubleValue(b) );
    }

    public PSLDataType lt( PSLDataType b ) {
        return new PSLBool( value < doubleValue(b) );
    }

    public PSLDataType le( PSLDataType b ) {
        return new PSLBool( value <= doubleValue(b) );
    }

    public PSLDataType eq( PSLDataType b ) {
        return new PSLBool( value == doubleValue(b) );
    }

    public PSLDataType ne( PSLDataType b ) {
        return new PSLBool( value != doubleValue(b) );
    }

    protected void onAssignValue(PSLDataType value)
    {
      if (value instanceof PSLReal)
      this.value = ((PSLReal)value).getValue();
    else if (value instanceof PSLInt)
      this.value = ((PSLInt)value).getValue();
    else
      this.error(value, "incompatible types");
    }

    public String toString()
    {
      return new String(""+value);
    }
}
```

A.23 PSLStockType.java

```
/**
 * @author xinli
 *
 *  wrapper class for stock
 */

package frontend;

import java.util.*;

import backend.DataPlotter;
import backend.GeoBrownMove;
import backend.PrintFormat;


public class PSLStockType extends PSLComplexTypeValue
{
  public PSLStockType()
  {
    this.defineProperty("Name", new PSLString(""));
    this.defineProperty("Price", new PSLReal(0));
    this.defineProperty("Return", new PSLReal(0));
    this.defineProperty("Volatility", new PSLReal(0));
    this.defineProperty("Dividend", new PSLReal(0));
  }

  public String getTypeName()
  {
    return "stock";
  }

  public ArrayList getMethods()
  {
    ArrayList list = new ArrayList();

    list.add("print");
    list.add("setVolatilityFromTimeSeries");
    list.add("getExpectedFuturePrice");
    list.add("simulate");

    return list;
  }

  public PSLDataType executeMethod(String methodName, PSLDataType[] params)
  {
    if (methodName.equals("print"))
    {
      if (params != null && params.length > 0)
        throw new PSLException("stock::print does not accept any arguments.");
      super.print();
    }
    else if(methodName.equals("getExpectedFuturePrice"))
    {
      if(params.length != 1)
      {
          throw new PSLException("stock::getExpectedFuturePrice(real): 1
argument expected.");
      }
      if(!(params[0] instanceof PSLReal))
      {
        throw new PSLException("stock::getExpectedFuturePrice(real): argument
must be of real type.");
```

```
      }
      double time = ((PSLReal)(params[0])).getValue();
      if(time <= 0)
      {
        throw new PSLException("stock::getExpectedFuturePrice(real): argument
must be positive.");
      }

      double price = ((PSLReal)getProperty("Price")).getValue();
      double ret = ((PSLReal)getProperty("Return")).getValue()/100.0;
      double futurePrice = price * Math.exp(time*ret/100.0);
      return new PSLReal(futurePrice);
    }
    else if(methodName.equals("setVolatilityFromTimeSeries"))
    {
      if(params.length != 1)
      {
          throw new PSLException("stock::setVolatilityFromTimeSeries(real[]): 1
argument expected.");
      }
      if(!(params[0] instanceof PSLComplexType))
      {
        throw new PSLException("stock::setVolatilityFromTimeSeries(real[]):
argument must be an array reference.");
      }
      PSLComplexType ref = (PSLComplexType)(params[0]);
      PSLArrayType S = (PSLArrayType)ref.getReferencedValue();
      double[] u = new double[S.getArraySize()-1];

      // get logarithmic return
      for(int i=0; i<S.getArraySize()-1; i++)
      {
        //Define ui as (Si-Si-1)/Si-1
        double Si = ((PSLReal)(S.getItem(i+1))).getValue();
        double Si1 = ((PSLReal)(S.getItem(i))).getValue();
        if(Si<0 || Si1<0)
        {
          throw new PSLException("Error: Negative stock price found in
stock::setVolatilityFromTimeSeries(real[])");
        }

        u[i] = Math.log(Si/Si1);
      }
      double sigma = backend.Statistics.calStdDev(u);
      this.setProperty("Volatility", new PSLReal(sigma*100.0));
    }

    else if(methodName.equals("simulate"))
    {
      if(params.length != 1)
      {
          throw new PSLException("stock::simulate(int): 1 argument expected.");
      }
      if(!(params[0] instanceof PSLInt))
      {
        throw new PSLException("stock::simulate(int): 1 argument must be of real
type.");
      }

      int steps = ((PSLInt)params[0]).getValue();
      if(steps <= 0)
      {
        throw new PSLException("stock::simulate(int): 1 argument must be
positive.");
      }
      double price = ((PSLReal)(getProperty("Price"))).getValue();
      double mu = ((PSLReal)(getProperty("Return"))).getValue()/100.0;
      double sigma = ((PSLReal)getProperty("Volatility")).getValue()/100.0;
      double[] simStockData = backend.GeoBrownMove.simulate(price, steps, mu,
sigma);
          PSLArrayType simStockDataArray = new PSLArrayType();
          simStockDataArray.setContainedTypeName("real");
          simStockDataArray.setArraySize(simStockData.length);

System.out.println("==============================================");
          System.out.println("Stock Monte Carlo Simulation Using Geometric
Brownian Motion");
          System.out.println("Name:
"+((PSLString)getProperty("Name")).getValue());
          System.out.println("Time Steps: "+simStockData.length);

          System.out.println("     Step      Price($)");

System.out.println("----------------------------------------------");

          for(int i=0; i<simStockData.length; i++)
          {
            System.out.print("     ");
            PrintFormat.Print_Format(i+1,20);
            PrintFormat.Print_Format(simStockData[i],15);
            System.out.println();
            //System.out.println(""+i+"    "+simStockData[i]);
            simStockDataArray.setItem(i,new PSLReal(simStockData[i]));
          }

          List Data = new ArrayList();
          List ValueName = new ArrayList();

          Data.add(simStockData);
          ValueName.add(((PSLString)getProperty("Name")).getValue());

          DataPlotter.drawPlot(Data, ValueName, "Stock Simulation") ;

System.out.println("==============================================");
          PSLComplexType simStockRef = new PSLComplexType("array");
          simStockRef.setReferencedValue(simStockDataArray);

          return simStockRef;
    }
    else
      throw new PSLException("stock." + methodName + " not implemented.");
    return null;
  }
}
```

A.24 PSLStringType.java

```
package frontend;

import java.io.*;
import java.util.*;

public class PSLString extends PSLPrimitiveType
{
    String value;

    public PSLString( String str ) {
        this.value = str;
    }

    public String getTypeName() {
        return "string";
    }

    public String getValue()
    {
      return value;
    }
    public PSLDataType copy()
    {
        return new PSLString( value );
    }

    public void print() {
        System.out.println( value );
    }

    public PSLDataType add( PSLDataType b )
  {
        if ( b instanceof PSLInt )
            return new PSLString( this.getValue() + ((PSLInt)b).getValue());
        else if (b instanceof PSLReal)
          return new PSLString(this.getValue() + ((PSLReal)b).getValue());
        else if (b instanceof PSLBool)
          return new PSLString(this.getValue() + ((PSLBool)b).getValue());
        else if (b instanceof PSLString)
          return new PSLString(this.getValue() + ((PSLString)b).getValue());
        else
          return this.error(b, "+");
  }

    public PSLDataType eq( PSLDataType b )
    {
      if (b instanceof PSLString)
        return new PSLBool(this.getValue().equals(((PSLString)b).getValue()));
        return error( b, "==" );
    }

    public PSLDataType ne( PSLDataType b )
    {
      if (b instanceof PSLString)
        return new PSLBool(!this.getValue().equals(((PSLString)b).getValue()));
        return error( b, "!=" );
    }

    protected void onAssignValue(PSLDataType value)
    {
      if(!(value instanceof PSLString))
        this.error(value, "incompatible types");
        this.value = ((PSLString)value).value;
    }
}
```

A.25 PSLSymbolTable.java

```
package frontend;

import java.util.*;

class PSLSymbolTable extends HashMap {

    PSLSymbolTable parent;
    private static PSLDataType returnValue = null;

    public PSLSymbolTable( PSLSymbolTable parent ) {
       this.parent = parent;
    }


    public final PSLSymbolTable parent() {
        return parent;
    }

    public final boolean containsVar( String name ) {

      PSLSymbolTable st = this;

      do{
        if(st.containsKey(name))
            return true;
        st = st.parent();
      }
      while(null != st);

        return false;
    }

    public final PSLDataType getValue( String name ) {

        Object x = this.get( name );

        PSLSymbolTable st = this;

        while ( null == x && null != st.parent() )
        {
            st = st.parent();
            x = st.get( name );
        }

        if(x==null)
        {
          throw new PSLException("Symbol "+"\""+name+"\""+" not defined in current
context.");
        }
        return (PSLDataType) x;
    }

    public final void setValue( String name, PSLDataType data)
  {
      if (this.containsKey(name))
        throw new PSLException("variable " + name + " already exists in the current
scope.");
        this.put( name, data );
    }

    public boolean functionDefined(String function_name)
    {
      if(!containsVar(function_name))
      {
        PSLDebugger.info("no such varaible");
        return false;
      }
      PSLDataType pt = getValue(function_name);
      if(!(pt instanceof PSLFunctionType))
      {
        PSLDebugger.info("this is not a function");
        return false;
      }
      return true;
    }

    public void setReturnValue(PSLDataType return_value)
    {
      returnValue = return_value.copy();
    }

    public PSLDataType getReturnValue()
    {
      return returnValue;
    }

}
```

A.26 BaseChart.java

```java
/* file name  : BaseChart.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/12/2004 04:40:26
 * copyright  :
 *
 * modifications:
 *
 */

package backend;

import java.awt.Color;
import java.awt.Font;
import java.awt.Rectangle;
import java.awt.Stroke;
import java.awt.Graphics2D;
import java.awt.BasicStroke;
import java.awt.RenderingHints;
import java.awt.geom.Rectangle2D;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;


public abstract class BaseChart implements Chart {

  /** The default border width. */
  public static final float DEFAULT_BORDER_WIDTH = 0.0F;
  /** The default padding. */
  public static final float DEFAULT_PADDING = 2.0F;
  /** The default title font. */
  public static final Font DEFAULT_TITLE_FONT =
          new Font("Helvetica", Font.BOLD, 20);
  /** The default description line width. */
  public static final float DEFAULT_DESCRIPTION_LINE_WIDTH = 1.0F;
  /** The default description color. */
  public static final Color DEFAULT_DESCRIPTION_COLOR = Color.black;
  /** The default description pattern. */
  public static final String DEFAULT_DESCRIPTION_PATTERN = "%v";
  /** The default grid color. */
  public static final Color DEFAULT_SHADOW_COLOR = Color.darkGray;
  /** The default shadow offset. */
  public static final float DEFAULT_SHADOW_OFFSET = 4.0F;
  /** The default description font. */
  public static final Font DEFAULT_DESCRIPTION_FONT =
          new Font("Helvetica", 0, 14);
  /** The default value format. */
  public static final NumberFormat DEFAULT_VALUE_FORMAT =
          NumberFormat.getNumberInstance();
  /** The default percent format. */
  public static final NumberFormat DEFAULT_PERCENT_FORMAT =
          NumberFormat.getNumberInstance();

  /** The chart model. */
  protected ChartModel model;
  /** The chart attributes. */
  protected ChartAttributes attributes = null;

  /**
   * Gets a list of supported attributes for the chart.
   * @return The list of supported attributes for the chart.
   */
```

```
public List getSupportedAttributes() {
  List attributes = new ArrayList();
  fillSupportedAttributes(attributes);
  return attributes;
}

/**
 * Adds an attribute definition to the
 * list of supported attributes for the chart.
 */
protected void addAttributeDefinition(
      List attributeDefinitions,
      String name,
      int type,
      boolean list,
      boolean optional) {
  attributeDefinitions.add(
    new ChartAttributeDefinition(
      name,
      type,
      list,
      optional
    )
  );
}

/**
 * Fills the list of supported attributes for the chart.
 * @param attrDefs The list of supported attributes for the chart to fill.
 */
protected void fillSupportedAttributes(List attrDefs) {
  addAttributeDefinition(attrDefs,
      ATTR_BACKGROUND_COLOR,
      ChartAttributeDefinition.TYPE_COLOR,
      false,
      true);
  addAttributeDefinition(attrDefs,
      ATTR_BORDER_WIDTH,
      ChartAttributeDefinition.TYPE_FLOAT,
      false, true);
  addAttributeDefinition(attrDefs,
      ATTR_TITLE_FONT,
      ChartAttributeDefinition.TYPE_FONT,
      false, true);
  addAttributeDefinition(attrDefs,
      ATTR_SEGMENT_LABELS,
      ChartAttributeDefinition.TYPE_STRING,
      true, true);
  addAttributeDefinition(attrDefs,
      ATTR_AREA_LABELS,
      ChartAttributeDefinition.TYPE_STRING,
      true, true);
  addAttributeDefinition(attrDefs,
      ATTR_DESCRIPTION_FONT,
      ChartAttributeDefinition.TYPE_FONT,
      false, true);
  addAttributeDefinition(attrDefs,
      ATTR_DESCRIPTION_PATTERN,
      ChartAttributeDefinition.TYPE_STRING,
      false, true);
  addAttributeDefinition(attrDefs,
      ATTR_DESCRIPTION_PATTERN,
      ChartAttributeDefinition.TYPE_STRING,
      false, true);
  addAttributeDefinition(attrDefs,
        ATTR_DESCRIPTION_LINE_WIDTH,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_DESCRIPTION_COLOR,
        ChartAttributeDefinition.TYPE_COLOR,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_VALUE_FORMAT,
        ChartAttributeDefinition.TYPE_FORMAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_PERCENT_FORMAT,
        ChartAttributeDefinition.TYPE_FORMAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SEGMENT_COLORS,
        ChartAttributeDefinition.TYPE_COLOR,
        true, true);
    addAttributeDefinition(attrDefs,
        ATTR_LINE_WIDTH,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SCALE_MARK_LENGTH,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SHADOW_COLOR,
        ChartAttributeDefinition.TYPE_COLOR,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SHADOW_OFFSET,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_VIEW_LEGEND,
        ChartAttributeDefinition.TYPE_BOOLEAN,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_VIEW_GRID,
        ChartAttributeDefinition.TYPE_BOOLEAN,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_GRID_COLOR,
        ChartAttributeDefinition.TYPE_COLOR,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_GRID_BACKGROUND_COLOR,
        ChartAttributeDefinition.TYPE_COLOR,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_FILL_AREA,
        ChartAttributeDefinition.TYPE_BOOLEAN,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_VIEW_SCALE_ARROWS,
        ChartAttributeDefinition.TYPE_BOOLEAN,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SCALE_ARROWHEAD_LENGTH,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SCALE_ARROWHEAD_WIDTH,
        ChartAttributeDefinition.TYPE_FLOAT,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_SCALE_ARROWHEAD_STYLE,
        ChartAttributeDefinition.TYPE_STRING,
        false, true);
    addAttributeDefinition(attrDefs,
        ATTR_VALUE_STEP_SIZE,
        ChartAttributeDefinition.TYPE_DOUBLE,
        false, true);
  }

  /**
   * Gets the chart attributes.
   */
  public ChartAttributes getAttributes() {
    return attributes;
  }
  /**
   * Sets the chart attributes.
   */
  public void setAttributes(ChartAttributes attributes) {
    this.attributes = attributes;
  }

  /**
   * Gets the background color.
   */
  public Color getBackgroundColor() {
    if (attributes.isDefined(ATTR_BACKGROUND_COLOR)) {
      return attributes.getColor(ATTR_BACKGROUND_COLOR);
    }
    return null;
  }

  /**
   * Gets the border width.
   */
  public float getBorderWidth() {
    if (attributes.isDefined(ATTR_BORDER_WIDTH)) {
      return attributes.getFloat(ATTR_BORDER_WIDTH);
    }
    return DEFAULT_BORDER_WIDTH;
  }

  /**
   * Gets the description line width.
   */
  public float getDescriptionLineWidth() {
    if (attributes.isDefined(ATTR_DESCRIPTION_LINE_WIDTH)) {
      return attributes.getFloat(ATTR_DESCRIPTION_LINE_WIDTH);
    }
    return DEFAULT_DESCRIPTION_LINE_WIDTH;
  }

  /**
   * Gets the description line color.
   */
  public Color getDescriptionColor() {
    if (attributes.isDefined(ATTR_DESCRIPTION_COLOR)) {
      return attributes.getColor(ATTR_DESCRIPTION_COLOR);
    }
    return DEFAULT_DESCRIPTION_COLOR;
  }
```

```
/**
 * Gets the description pattern.
 */
public String getDescriptionPattern() {
  String descriptionPattern = attributes.getString(ATTR_DESCRIPTION_PATTERN);
  if (descriptionPattern != null) {
    return descriptionPattern;
  }

  return DEFAULT_DESCRIPTION_PATTERN;
}

/**
 * Computes the fill color for value area.
 */
public Color getShadowColor()  {
  Color color = DEFAULT_SHADOW_COLOR;
  if (attributes.isDefined(ATTR_SHADOW_COLOR)) {
    color = attributes.getColor(ATTR_SHADOW_COLOR);
  }
  int r = color.getRed();
  int g = color.getGreen();
  int b = color.getBlue();
  return new Color(r, g, b, 192);
}

/**
 * Gets the shadow length.
 */
public float getShadowLength() {
  if (attributes.isDefined(ATTR_SHADOW_OFFSET)) {
    return attributes.getFloat(ATTR_SHADOW_OFFSET);
  }
  return DEFAULT_SHADOW_OFFSET;
}

/**
 * Gets the number format for value output.
 */
public NumberFormat getValueFormat() {
  if (attributes.isDefined(ATTR_VALUE_FORMAT)) {
    return (NumberFormat) attributes.getObject(ATTR_VALUE_FORMAT);
  }
  return DEFAULT_VALUE_FORMAT;
}

/**
 * Gets the number format for percent output.
 */
public NumberFormat getPercentFormat() {
  if (attributes.isDefined(ATTR_PERCENT_FORMAT)) {
    return (NumberFormat) attributes.getObject(ATTR_PERCENT_FORMAT);
  }
  return DEFAULT_PERCENT_FORMAT;
}

/**
 * Gets the left padding.
 */
public float getLeftPadding() {
  if (attributes.isDefined(ATTR_PADDING_LEFT)) {
    return attributes.getFloat(ATTR_PADDING_LEFT);
  } else if (attributes.isDefined(ATTR_PADDING)) {
    return attributes.getFloat(ATTR_PADDING);
  }
  return DEFAULT_PADDING;
}

/**
 * Gets the right padding.
 */
public float getRightPadding() {
  if (attributes.isDefined(ATTR_PADDING_RIGHT)) {
    return attributes.getFloat(ATTR_PADDING_RIGHT);
  } else if (attributes.isDefined(ATTR_PADDING)) {
    return attributes.getFloat(ATTR_PADDING);
  }
  return DEFAULT_PADDING;
}

/**
 * Gets the top padding.
 * @return The current top padding.
 */
public float getTopPadding() {
  if (attributes.isDefined(ATTR_PADDING_TOP)) {
    return attributes.getFloat(ATTR_PADDING_TOP);
  } else if (attributes.isDefined(ATTR_PADDING)) {
    return attributes.getFloat(ATTR_PADDING);
  }
  return DEFAULT_PADDING;
}
/**
 * Gets the top padding.
 */
public float getBottomPadding() {
  if (attributes.isDefined(ATTR_PADDING_BOTTOM)) {
    return attributes.getFloat(ATTR_PADDING_BOTTOM);
  } else if (attributes.isDefined(ATTR_PADDING)) {
    return attributes.getFloat(ATTR_PADDING);
  }
  return DEFAULT_PADDING;
}

/**
 * Gets the title font.
 */
public Font getTitleFont() {
  if (attributes.isDefined(ATTR_TITLE_FONT)) {
    return attributes.getFont(ATTR_TITLE_FONT);
  }
  return DEFAULT_TITLE_FONT;
}

/**
 * Gets the description font.
 */
public Font getDescriptionFont() {
  if (attributes.isDefined(ATTR_DESCRIPTION_FONT)) {
    return attributes.getFont(ATTR_DESCRIPTION_FONT);
  }
  return DEFAULT_DESCRIPTION_FONT;
}

/**
 * Gets the segment labels.
 */
public String[] getSegmentLabels() {
  if (attributes.isDefined(ATTR_SEGMENT_LABELS)) {
    return attributes.getStringArray(ATTR_SEGMENT_LABELS);
  }
  return null;
}

/**
 * Gets the area labels.
 */
public String[] getAreaLabels() {
  if (attributes.isDefined(ATTR_AREA_LABELS)) {
    return attributes.getStringArray(ATTR_AREA_LABELS);
  }
  return null;
}

/**
 * Gets the chart model.
 */
public ChartModel getModel() {
  return model;
}
/**
 * Sets the chart model.
 */
public void setModel(ChartModel model) throws ChartException {
  this.model = model;
  if (!supportsNegativeValues()
      && (model != null)
      && (model.getMinValue() < 0.0)) {
    throw new ChartException("This chart type doesn't support negative
values.");
  }
}

/**
 * Tells, whether the chart supports negative values or not.
 */
public boolean supportsNegativeValues() {
  return true;
}

/**
 * Gets the minimum value from the chart data.
 */
protected double getMinValue() {
  return model.getMinValue();
}

/**
 * Gets the maximum value from the chart data.
 */
protected double getMaxValue() {
  return model.getMaxValue();
}

/**
 * Draws the chart to the given chart output.
 */
public Rectangle2D draw(Rectangle2D viewport, Graphics2D graphics)
  throws ChartException {
    graphics.setRenderingHint(
        RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON
        );
  float borderWidth = getBorderWidth();
  if (borderWidth > 0.0) {
    Stroke oldStroke = graphics.getStroke();
    graphics.setStroke(new BasicStroke(borderWidth));
    graphics.drawRect(
          (int) viewport.getX(),
          (int) viewport.getY(),
          (int) viewport.getWidth(),
          (int) viewport.getHeight()
          );
    graphics.setStroke(oldStroke);
  }

  Color backgroundColor = getBackgroundColor();
  if (backgroundColor != null) {
    Color c = graphics.getColor();
    graphics.setColor(backgroundColor);
    graphics.fillRect(
        (int) viewport.getMinX(),
        (int) viewport.getMinY(),
        (int) viewport.getWidth(),
        (int) viewport.getHeight()
    );
    graphics.setColor(c);
  }

  viewport = new Rectangle(
    (int) (viewport.getMinX() + getLeftPadding()),
    (int) (viewport.getMinY() + getTopPadding()),
    (int) (viewport.getWidth() - getLeftPadding() - getRightPadding()),
    (int) (viewport.getHeight() - getTopPadding() - getBottomPadding())
  );
  TextChartItem title = model.getTitle();
  if (title != null) {
    // drawing the title
    title.resetMetrics();
    title.draw(viewport, graphics);
  }
  return viewport;
}
}
```

A.27 BaseChartItem.java

```
/* file name  : BaseChartItem.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/12/2004 05:52:34
 */

package backend;

import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;


public abstract class BaseChartItem implements ChartItem {

  /** The position of the chart item. */
  protected ChartItemPosition position = null;
  /** The output area. */
  protected Rectangle2D outputArea = null;

  /**
   * Constructs a BaseChartItem instance with specific attributes.
   */
  public BaseChartItem(ChartItemPosition position) {
    this.position = position;
  }

  /**
   * Gets the position of the chart item.
   * @return The current position.
   */
  public ChartItemPosition getPosition() {
    return position;
  }

  /**
   * Gets the output area ot the chart item.
   */
  public Rectangle2D getOutputArea(Rectangle2D viewport, Graphics2D graphics) {
    if (outputArea == null) {
      outputArea = calculateOutputArea(viewport, graphics);
    }
    return outputArea;
  }

  /**
   * Resets the calculated metrics to enforce re-calculation of the
   * item's output area.
   */
  public void resetMetrics() {
    this.outputArea = null;
  }

  /**
   * Calculates the position of the item.
   */
  protected abstract Rectangle2D calculateOutputArea(Rectangle2D viewport,
Graphics2D graphics);


  /**
   * Gets the width of the item box.
   */
  public double getWidth(Rectangle2D viewport, Graphics2D graphics) {
    return getOutputArea(viewport, graphics).getWidth();
  }

  /**
   * Gets the height of the item box.
   */
  public double getHeight(Rectangle2D viewport, Graphics2D graphics) {
    return getOutputArea(viewport, graphics).getHeight();
  }

 /**
   * Moves the item position.
   */
  public void move(double x, double y) {
     double curX = 0.0;
    double curY = 0.0;
    double curWidth = 0.0;
    double curHeight = 0.0;

    if (outputArea != null) {
      curX = outputArea.getX();
      curY = outputArea.getY();
      curWidth = outputArea.getWidth();
      curHeight = outputArea.getHeight();
    }

    if (x != 0.0) {
      if (position.getHorizontalPositionMode() == ChartItemPosition.ABSOLUTE) {
        position.setX(position.getX() + x);
        curX = position.getX();
      }
    }

    if (y != 0.0) {
      if (position.getVerticalPositionMode() == ChartItemPosition.ABSOLUTE) {
        position.setY(position.getY() + y);
        curY = position.getY();
      }
    }

    if (outputArea != null) {
      outputArea.setFrame(curX, curY, curWidth, curHeight);
    }
  }

  /**
   * Moves the item to the desired position.
   */
  public void moveTo(double x, double y) {
    double curX = 0.0;
    double curY = 0.0;
    double curWidth = 0.0;
    double curHeight = 0.0;

    if (outputArea != null) {
      curX = outputArea.getX();
      curY = outputArea.getY();
      curWidth = outputArea.getWidth();
      curHeight = outputArea.getHeight();
    }

    if (x != 0.0) {
      if (position.getHorizontalPositionMode() == ChartItemPosition.ABSOLUTE) {
        position.setX(x);
        curX = position.getX();
      }
    }

    if (y != 0.0) {
      if (position.getVerticalPositionMode() == ChartItemPosition.ABSOLUTE) {
        position.setY(y);
        curY = position.getY();
      }
    }

    if (outputArea != null) {
      outputArea.setFrame(curX, curY, curWidth, curHeight);
    }
  }
}
```

A.28 BaseChartModel.java

```java
/* file name  : BaseChartModel.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/18/2004 05:13:01
 */

package backend;

import java.util.List;

public class BaseChartModel implements ChartModel {

  /** The chart's title item. */
  protected TextChartItem title = null;
  /** The chart data. */
  protected List chartData = null;
  /** The chart value descriptions. */
  protected List valueDescriptions = null;
  /** The minimum data value. */
  protected double minValue = 0.0;
  /** The maximum data value. */
  protected double maxValue = 0.0;

  /**
   * Constructs a DefaultChartModel instance with no specific attributes.
   */
  public BaseChartModel() {
  }

  public TextChartItem getTitle() {
    return title;
  }

  public void setTitle(TextChartItem title) {
    this.title = title;

  }

  public List getChartData() {
    return chartData;
  }

  public void setChartData(List chartData) {
    this.chartData = chartData;
    if (chartData == null) {
      return;
    }
    int numSegments = chartData.size();
    double min = minValue;
    double max = maxValue;
    // set the minValue and maxValue
    for (int i = 0; i < numSegments; i++) {
      double[] row = (double[]) chartData.get(i);
      for (int j = 0; j < row.length; j++) {
        double val = row[j];
        min = Math.min(min, val);
        max = Math.max(max, val);
      }
    }
    this.minValue = Math.min(minValue, min);
    this.maxValue = Math.max(maxValue, max);
  }
  /**
   * Gets the chart value descriptions as a list of Strings.
   */
  public List getValueDescriptions() {
    return valueDescriptions;
  }
  /**
   * Sets the chart value descriptions as a list of Strings.
   */
  public void setValueDescriptions(List valueDescriptions) {
    this.valueDescriptions = valueDescriptions;
  }

  /**
   * Gets the minimum value from the chartData.
   */
  public double getMinValue() {
    return minValue;
  }

  /**
   * Sets the minimum value from the chart data.
   */
  public void setMinValue(double minValue) {
    this.minValue = minValue;
  }

  /**
   * Gets the maximum value from the chartData.
   */
  public double getMaxValue() {
    return maxValue;
  }
  /**
   * Sets the maximum value from the chart data.
   */
  public void setMaxValue(double maxValue) {
    this.maxValue = maxValue;
  }

  /**
   * Gets the number of segments.
   */
  public int getNumSegments() {
    return chartData.size();
  }

  /**
   * Gets the number of areas.
   */
  public int getNumAreas() {
    int numAreas = 0;
    int numSegments = chartData.size();
    for (int i = 0; i < numSegments; i++) {
      double[] row = (double[]) chartData.get(i);
      numAreas = Math.max(numAreas, row.length);
    }
    return numAreas;
  }
}
```

A.29 BastScaleChart.java

```
/* file name  : Base2DScaleChart.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/26/2004 04:34:22
 */

package backend;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.Rectangle;
import java.awt.Stroke;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.text.NumberFormat;
import java.util.List;

public abstract class BaseScaleChart extends BaseChart {

  /** The segment mark style for centered lines. */
  public static final int SEGMENT_MARK_STYLE_CENTER = 1;
  /** The segment mark style for border lines. */
  public static final int SEGMENT_MARK_STYLE_BORDER = 2;
  /** The default segment mark style. */
  public static final int DEFAULT_SEGMENT_MARK_STYLE =
            SEGMENT_MARK_STYLE_BORDER;

  /** The arrowhead style for open arrows. */
  public static final int ARROWHEAD_STYLE_OPEN = 1;
  /** The arrowhead style for filled arrows. */
  public static final int ARROWHEAD_STYLE_FILLED = 2;
  /** The default arrowhead style. */
  public static final int DEFAULT_ARROWHEAD_STYLE =
            ARROWHEAD_STYLE_OPEN;

  /** The default line width. */
  public static final float DEFAULT_LINE_WIDTH = 1.0F;
  /** The default line width. */
  public static final float DEFAULT_SCALE_MARK_LENGTH = 6.0F;
  /** The default scale arrowhead length. */
  public static final float DEFAULT_SCALE_ARROWHEAD_LENGTH = 10.0F;
  /** The default scale arrowhead style. */
  public static final String DEFAULT_SCALE_ARROWHEAD_STYLE = "open";
  /** The default grid color. */
  public static final Color DEFAULT_GRID_COLOR = Color.gray;
  /** The default grid background color. */
  public static final Color DEFAULT_GRID_BACKGROUND_COLOR = Color.lightGray;

  /** The Font to use for the grid. */
  protected Font scaleFont = null;
  /** The effective viewport (is determined whenever draw scale is called). */
  protected Rectangle effectiveViewport = null;
  /** The vertical anchor from which to draw the chart. */
  protected int verticalAnchor = 0;
  /** The minimum vertical position. */
  protected int verticalMinimum = 0;
  /** The maximum vertical position. */
  protected int verticalMaximum = 0;
  /** The vertical position of the zero coordinate. */
```

```
protected int verticalZero = 0;

/** The maximum value. */
protected double max;
/** The minimum value. */
protected double min;
/** The value range. */
protected double range;

/** The horizontal offset for the chart. */
protected double horizontalOffset = 0.0;
/** The horizontal scale for the chart. */
protected double horizontalScale = 0.0;
/** The vertical scale for the chart values. */
protected double verticalScale = 0.0;
/** The horizontal positions for the values. */
protected int[] horizontalPositions = null;

/** The maximum segment label height, calculated at method calcViewport. */
private double maxSegmentLabelHeight = 0.0;
/** The segment metrics, calculated at method calcViewport. */
private Rectangle2D[] segmentMetrics = null;

/** The maximum area label height, calculated at method calcViewport. */
private double maxAreaLabelHeight = 0.0;
/** The legend metrics, calculated at method calcViewport. */
private Rectangle2D legendMetrics = null;

/**
 * Gets the line width.
 */
public float getLineWidth() {
  if (attributes.isDefined(ATTR_LINE_WIDTH)) {
    return attributes.getFloat(ATTR_LINE_WIDTH);
  }
  return DEFAULT_LINE_WIDTH;
}

/**
 * Gets the scale mark length.
 */
public float getScaleMarkLength() {
  if (attributes.isDefined(ATTR_SCALE_MARK_LENGTH)) {
    return attributes.getFloat(ATTR_SCALE_MARK_LENGTH);
  }
  return DEFAULT_SCALE_MARK_LENGTH;
}

/**
 * Gets the value step size.
 */
public float getValueStepSize() {
  if (attributes.isDefined(ATTR_VALUE_STEP_SIZE)) {
    return attributes.getFloat(ATTR_VALUE_STEP_SIZE);
  }
  return 0.0F;
}

/**
 * Gets the 'view-legend' flag.
 */
public boolean isViewLegend() {
  if (!attributes.isDefined(ATTR_VIEW_LEGEND)) {
    return true;
  }
```

```
    return attributes.getBoolean(ATTR_VIEW_LEGEND);
  }

  /**
   * Gets the 'view-scale-arrows' flag.
   */
  public boolean isViewScaleArrows() {
    if (!attributes.isDefined(ATTR_VIEW_SCALE_ARROWS)) {
      return true;
    }
    return attributes.getBoolean(ATTR_VIEW_GRID);
  }

  /**
   * Gets the scale arrowhead length.
   */
  public float getScaleArrowheadLength() {
    if (attributes.isDefined(ATTR_SCALE_ARROWHEAD_LENGTH)) {
      return attributes.getFloat(ATTR_SCALE_ARROWHEAD_LENGTH);
    }
    return DEFAULT_SCALE_ARROWHEAD_LENGTH;
  }

  /**
   * Gets the scale arrowhead width.
   */
  public float getScaleArrowheadWidth() {
    if (attributes.isDefined(ATTR_SCALE_ARROWHEAD_WIDTH)) {
      return attributes.getFloat(ATTR_SCALE_ARROWHEAD_WIDTH);
    }
    return getScaleArrowheadLength() * 0.4F;
  }

  /**
   * Gets the scale arrowhead style.
   */
  public int getScaleArrowheadStyle() {
    if (attributes.isDefined(ATTR_SCALE_ARROWHEAD_STYLE)) {
      String style = attributes.getString(ATTR_SCALE_ARROWHEAD_STYLE);
      if (style.equals("filled")) {
        return ARROWHEAD_STYLE_FILLED;
      }
    }
    return DEFAULT_ARROWHEAD_STYLE;
  }

  /**
   * Gets the segment mark style.
   */
  public int getSegmentMarkStyle() {
    if (attributes.isDefined(ATTR_SEGMENT_MARK_STYLE)) {
      String style = attributes.getString(ATTR_SEGMENT_MARK_STYLE);
      if (style.equals("border")) {
        return SEGMENT_MARK_STYLE_BORDER;
      }
      return SEGMENT_MARK_STYLE_CENTER;
    }
    return DEFAULT_SEGMENT_MARK_STYLE;
  }
  /**
   * Gets the 'view-grid' flag.
   */
  public boolean isViewGrid() {
    if (!attributes.isDefined(ATTR_VIEW_GRID)) {
      return true;
    }
    return attributes.getBoolean(ATTR_VIEW_GRID);
  }

  /**
   * Gets the grid color.
   */
  public Color getGridColor() {
    if (attributes.isDefined(ATTR_GRID_COLOR)) {
      return attributes.getColor(ATTR_GRID_COLOR);
    }
    return DEFAULT_GRID_COLOR;
  }

  /**
   * Gets the grid background color.
   */
  public Color getGridBackgroundColor() {
    if (attributes.isDefined(ATTR_GRID_BACKGROUND_COLOR)) {
      return attributes.getColor(ATTR_GRID_BACKGROUND_COLOR);
    }
    return DEFAULT_GRID_BACKGROUND_COLOR;
  }

  /**
   * Gets the inner padding.
   */
  protected int getInnerPadding() {
    return 10;
  }

  /**
   * Draws the scale.
   */
  protected void drawScale(Rectangle2D viewport,
                Rectangle2D effectiveViewport,
                Graphics2D graphics) {

    Color scaleColor = getDescriptionColor();
    Color backgroundColor = getGridBackgroundColor();
    Color gridColor = getGridColor();

    int halfScaleLength = (int) (getScaleMarkLength() / 2.0F);
    int x = (int) effectiveViewport.getX();
    int width = (int) effectiveViewport.getWidth();

    int arrowHeadLength = (int) getScaleArrowheadLength();
    int arrowHeadWidth = (int) getScaleArrowheadWidth();
    int arrowHeadStyle = getScaleArrowheadStyle();
    int segmentMarkStyle = getSegmentMarkStyle();

    List chartData = model.getChartData();
    int numSegments = chartData.size();

    Font oldFont = graphics.getFont();
    Stroke oldStroke = graphics.getStroke();
    Stroke gridStroke = new BasicStroke(getDescriptionLineWidth() / 2);
    Stroke scaleStroke = new BasicStroke(getDescriptionLineWidth(),
              BasicStroke.CAP_ROUND,
              BasicStroke.JOIN_ROUND);
    graphics.setStroke(scaleStroke);

    String[] segmentLabels = getSegmentLabels();

    Font segmentDescriptionFont = getDescriptionFont();
```

```
segmentDescriptionFont = segmentDescriptionFont.deriveFont(
                AffineTransform.getRotateInstance(Math.PI * 1.5));
graphics.setFont(segmentDescriptionFont);

Rectangle segmentDescriptionArea = new Rectangle(x,
            verticalMinimum + halfScaleLength,
            width,
            (int) maxSegmentLabelHeight);

boolean viewGrid = isViewGrid();
if (viewGrid) {
  // drawing grid background
  graphics.setColor(backgroundColor);
  graphics.fillRect(x, verticalMaximum,
                width, verticalMinimum - verticalMaximum);
}

graphics.setColor(scaleColor);
// drawing the horizontal scale
int vsMaxX = (int) (viewport.getX() + viewport.getWidth());
int maxGridX = x + width;
if (isViewScaleArrows()) {
  if (arrowHeadStyle == ARROWHEAD_STYLE_FILLED) {
    Polygon arrowPoly = new Polygon();
    arrowPoly.addPoint(vsMaxX - arrowHeadLength,
              verticalMinimum - arrowHeadWidth);
    arrowPoly.addPoint(vsMaxX, verticalMinimum);
    arrowPoly.addPoint(vsMaxX - arrowHeadLength,
        verticalMinimum + arrowHeadWidth);
    graphics.fill(arrowPoly);
    graphics.draw(arrowPoly);
  } else {
    graphics.drawLine(vsMaxX - arrowHeadLength,
                  verticalMinimum - arrowHeadWidth,
                  vsMaxX, verticalMinimum);
    graphics.drawLine(vsMaxX - arrowHeadLength,
                  verticalMinimum + arrowHeadWidth,
                  vsMaxX, verticalMinimum);
  }
  graphics.drawLine(x, verticalMinimum,
                vsMaxX, verticalMinimum);
} else {
  graphics.drawLine(x, verticalMinimum, maxGridX, verticalMinimum);
}

// drawing the vertical scale
int vsMinY = (int) viewport.getY();
if (isViewScaleArrows()) {
  graphics.drawLine(x, vsMinY, x, verticalMinimum);
  if (arrowHeadStyle == ARROWHEAD_STYLE_FILLED) {
    Polygon arrowPoly = new Polygon();
    arrowPoly.addPoint(x - arrowHeadWidth, vsMinY + arrowHeadLength);
    arrowPoly.addPoint(x, vsMinY);
    arrowPoly.addPoint(x + arrowHeadWidth, vsMinY + arrowHeadLength);
    graphics.fill(arrowPoly);
    graphics.draw(arrowPoly);
  } else {
    graphics.drawLine(x, vsMinY, x - arrowHeadWidth,
                    vsMinY + arrowHeadLength);
    graphics.drawLine(x, vsMinY, x + arrowHeadWidth,
                    vsMinY + arrowHeadLength);
  }
} else {
  graphics.drawLine(x, verticalMaximum, x, verticalMinimum);
}
```

```
// drawing the horizontal descriptions
this.horizontalPositions = new int[numSegments];
int hScaleYfrom = verticalMinimum - halfScaleLength;
int hScaleYto = verticalMinimum + halfScaleLength;
double halfHorizontalScale = horizontalScale / 2.0;
double lastPos = horizontalOffset + halfHorizontalScale;

double segXPos = 0.0;
switch (segmentMarkStyle) {
  case SEGMENT_MARK_STYLE_BORDER:
    segXPos = (int) effectiveViewport.getMinX();
    break;

  case SEGMENT_MARK_STYLE_CENTER:
    segXPos = (int) lastPos;
    break;
}

// To determine the internal of horizontal descriptions
int horiSteps;
if (numSegments > 500) {
    horiSteps = 100;
} else if ( numSegments > 300) {
    horiSteps = 50;
} else if ( numSegments > 50) {
    horiSteps = 10;
} else if ( numSegments > 15) {
    horiSteps = 5;
} else {
    horiSteps = 1;
}

graphics.setFont(segmentDescriptionFont);
for (int i = 0; i < numSegments; i++) {
  int xPos = (int) lastPos;
  int xSegPos = (int) segXPos;
  horizontalPositions[i] = xPos;
  if (viewGrid && ( i % horiSteps == 0) &&
      ((i > 0) || (segmentMarkStyle != SEGMENT_MARK_STYLE_BORDER))) {
    graphics.setStroke(gridStroke);
    graphics.setColor(gridColor);
    graphics.drawLine(xSegPos, verticalMinimum, xSegPos, verticalMaximum);
  }
  if ( i % horiSteps == 0) {
    graphics.setStroke(scaleStroke);
    graphics.setColor(scaleColor);
    graphics.drawLine(xSegPos, hScaleYfrom, xSegPos, hScaleYto);
  }
  lastPos += horizontalScale;
  segXPos += horizontalScale;
  Rectangle2D segmentMetric = segmentMetrics[i];
  if ((segmentLabels != null) && (i < segmentLabels.length) &&
      ( i % horiSteps == 0) ) {
    graphics.drawString(
      segmentLabels[i],
      (float) (xPos + segmentMetric.getHeight() / 3.0),
      (float) (segmentDescriptionArea.getMinY() +
               segmentMetric.getWidth() + halfScaleLength * 2)
    );
  }
}

if (segmentMarkStyle == SEGMENT_MARK_STYLE_BORDER) {
  int xSegPos = (int) segXPos;
  if (viewGrid) {
    graphics.setStroke(gridStroke);
    graphics.setColor(gridColor);
    graphics.drawLine(xSegPos, verticalMinimum, xSegPos, verticalMaximum);
  }
  graphics.setStroke(scaleStroke);
  graphics.setColor(scaleColor);
  graphics.drawLine(xSegPos, hScaleYfrom, xSegPos, hScaleYto);
}

// drawing the vertical descriptions
segmentDescriptionFont = getDescriptionFont();
graphics.setFont(segmentDescriptionFont);
FontMetrics fm = graphics.getFontMetrics();
NumberFormat valueFormat = getValueFormat();

float stepSize = getValueStepSize();
double maxValue = getMaxValue();
double minValue = getMinValue();
double valueRange = maxValue - minValue;

if (stepSize == 0.0F) {
  if (valueRange > 500000) {
    stepSize = 100000;
  } else if (valueRange > 50000) {
    stepSize = 10000;
  } else if (valueRange > 5000) {
    stepSize = 1000;
  } else if (valueRange > 500) {
    stepSize = 100;
  } else if (valueRange > 50) {
    stepSize = 10;
  } else {
    stepSize = 1;
  }
}

float base = (float) minValue;
if (base < 0.0F) {
  base = 0.0F;
}

// drawing the zero position
int xPos = (int) effectiveViewport.getMinX();
int yPos = calculateVerticalPosition(0.0);
int vScaleXfrom = xPos - halfScaleLength;
int vScaleXto = xPos + halfScaleLength;

String strValue = null;
Rectangle2D textMetrics = null;

for (float value = base; value <= maxValue; value += stepSize) {
  yPos = calculateVerticalPosition(value);
  if (viewGrid) {
    graphics.setStroke(gridStroke);
    graphics.setColor(gridColor);
    if (value != base) {
      graphics.drawLine(x, yPos, maxGridX, yPos);
    }
  }

  graphics.setStroke(scaleStroke);
  graphics.setColor(scaleColor);
  graphics.drawLine(vScaleXfrom, yPos, vScaleXto, yPos);
```

```
      strValue = valueFormat.format(value);
      textMetrics = fm.getStringBounds(strValue, graphics);

      graphics.drawString(
        strValue,
        (float) (xPos - halfScaleLength * 2 - textMetrics.getWidth()),
        (float) (yPos + textMetrics.getHeight() / 3)
      );
    }

    for (float value = -stepSize; value >= minValue; value -= stepSize) {
      yPos = calculateVerticalPosition(value);
      if (viewGrid) {
        graphics.setStroke(gridStroke);
        graphics.setColor(gridColor);
        graphics.drawLine(x, yPos, maxGridX, yPos);
      }

      graphics.setStroke(scaleStroke);
      graphics.setColor(scaleColor);
      graphics.drawLine(vScaleXfrom, yPos, vScaleXto, yPos);

      strValue = valueFormat.format(value);
      textMetrics = fm.getStringBounds(strValue, graphics);

      graphics.drawString(
        strValue,
        (float) (xPos - halfScaleLength * 2 - textMetrics.getWidth()),
        (float) (yPos + textMetrics.getHeight() / 3)
      );
    }

    graphics.setStroke(oldStroke);
    graphics.setFont(oldFont);
  }

  /**
   * Draws the legend.
   */
  protected void drawLegend(Rectangle2D viewport, Graphics2D graphics) {

    int padding = getInnerPadding();

    if (isViewLegend()) {
      Font oldFont = graphics.getFont();
      Stroke oldStroke = graphics.getStroke();
      graphics.setStroke(new BasicStroke(getDescriptionLineWidth(),
                     BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));

      double labelBoxSize = maxAreaLabelHeight * 0.75;
      double labelBoxXPos = legendMetrics.getMinX() + padding;
      double labelXPos = labelBoxXPos + labelBoxSize * 2;
      double labelYPos = legendMetrics.getMinY();

      Color[] segmentColors = attributes.getColorArray(ATTR_SEGMENT_COLORS);

      Font segmentDescriptionFont = getDescriptionFont();
      graphics.setFont(segmentDescriptionFont);

      String[] areaLabels = getAreaLabels();
      for (int i = 0; i < areaLabels.length; i++) {
        String label = areaLabels[i];
        labelYPos += maxAreaLabelHeight * 1.5;

        // setting fill color
        Color oldColor = graphics.getColor();
        graphics.setColor(segmentColors[i]);

        graphics.fillRect(
          (int) labelBoxXPos,
          (int) (labelYPos - labelBoxSize),
          (int) (labelBoxSize),
          (int) (labelBoxSize)
        );

        // restoring old color
        graphics.setColor(oldColor);
        graphics.drawRect(
          (int) labelBoxXPos,
          (int) (labelYPos - labelBoxSize),
          (int) (labelBoxSize),
          (int) (labelBoxSize)
        );
        graphics.drawString(label, (float) labelXPos, (float) labelYPos);
      }

      graphics.draw(legendMetrics);

      // setting old font again
      graphics.setFont(oldFont);
      graphics.setStroke(oldStroke);
    }
  }

  /**
   * Calculates the viewport.
   */
  protected Rectangle calcViewport(Rectangle2D viewport, Graphics2D graphics) {

    int padding = getInnerPadding();
    int x = (int) (viewport.getX() + padding);
    int y = (int) (viewport.getY() + padding);
    int width = (int) viewport.getWidth() - padding * 2;
    int height = (int) viewport.getHeight() - padding * 2;

    if ((model == null) || (model.getChartData() == null)) {
      effectiveViewport = new Rectangle(x, y, width, height);
      return effectiveViewport;
    }

    TextChartItem title = model.getTitle();
    if (title != null) {
      Rectangle2D titleArea = title.getOutputArea(viewport, graphics);
      height -= (int) (titleArea.getMaxY() - y);
      y = (int) titleArea.getMaxY();
    }
    effectiveViewport = new Rectangle(x, y, width, height);

    List chartData = model.getChartData();
    int numSegments = chartData.size();
    this.min = getMinValue();
    if (min < 0.0) {
      min -= 5;
    }
    this.max = getMaxValue() * 1.05;
    this.range = max - min;

    Font oldFont = graphics.getFont();

    Font segmentDescriptionFont = getDescriptionFont();
    graphics.setFont(segmentDescriptionFont);
    FontMetrics fm = graphics.getFontMetrics();

    /* calculating value description metrics. */
    String maxValue = Long.toString((long) getMaxValue());
    String minValue = Long.toString((long) getMinValue());
    Rectangle2D maxValueMetrics = null;
    if (maxValue.length() > minValue.length()) {
      maxValueMetrics = fm.getStringBounds(maxValue, graphics);
    } else {
      maxValueMetrics = fm.getStringBounds(minValue, graphics);
    }

    width -= (int) (maxValueMetrics.getMaxX() + padding);
    x += (int) (maxValueMetrics.getMaxX() + padding);

    if (isViewLegend()) {
      /* calculating legend metrics. */
      maxAreaLabelHeight = 0.0;
      double areaLabelWidth = 0.0;
      String[] areaLabels = getAreaLabels();
      for (int i = 0; i < areaLabels.length; i++) {
        String label = areaLabels[i];
        Rectangle2D textMetrics = fm.getStringBounds(label, graphics);
        maxAreaLabelHeight = Math.max(maxAreaLabelHeight,
textMetrics.getHeight());
        areaLabelWidth = Math.max(areaLabelWidth, textMetrics.getWidth());
      }

      double legendHeight = areaLabels.length * (maxAreaLabelHeight * 1.5) +
padding;
      areaLabelWidth += (maxAreaLabelHeight * 1.5) + (padding) * 2;

      // reducing effective viewport
      width -= (int) (areaLabelWidth + padding * 2);

      // setting legend metrics
      this.legendMetrics = new Rectangle(
        x + width + padding,
        y + padding,
        (int) areaLabelWidth,
        (int) legendHeight
      );
    }

    effectiveViewport = new Rectangle(x, y, width, height);

    /* calculating segment description metrics. */
    String[] segmentLabels = getSegmentLabels();
    this.maxSegmentLabelHeight = 0.0;
    this.segmentMetrics = new Rectangle2D[numSegments];

    this.horizontalScale = effectiveViewport.getWidth() /
          (model.getChartData().size());
    this.horizontalOffset = effectiveViewport.getX();
    this.verticalAnchor = (int) (effectiveViewport.getHeight()
                + viewport.getY() + padding);

    if (segmentLabels != null) {
      String label = null;
      for (int i = 0; i < numSegments; i++) {
        if (i < segmentLabels.length) {
          label = segmentLabels[i];
        } else {
          label = "null";
        }
        Rectangle2D textMetrics = fm.getStringBounds(label, graphics);
        segmentMetrics[i] = textMetrics;
        maxSegmentLabelHeight = Math.max(maxSegmentLabelHeight,
textMetrics.getWidth());
      }
    }

    int vertDescriptionDelta = (int) maxSegmentLabelHeight;
    this.verticalScale = (effectiveViewport.getHeight() - vertDescriptionDelta)
/ range;

    this.verticalZero = calculateVerticalPosition(0.0);
    if (vertDescriptionDelta > 0) {
      verticalAnchor -= vertDescriptionDelta;
      verticalZero = calculateVerticalPosition(0.0);
    }
    this.verticalScale = (verticalZero - effectiveViewport.getY() - padding) /
max;

    this.horizontalPositions = new int[numSegments];
    double lastPos = horizontalOffset + (horizontalScale / 2.0);

    graphics.setFont(segmentDescriptionFont);
    for (int i = 0; i < numSegments; i++) {
      int xPos = (int) lastPos;
      horizontalPositions[i] = xPos;
      lastPos += horizontalScale;
    }

    this.verticalMinimum = verticalAnchor;
    this.verticalMaximum = calculateVerticalPosition(max);
    if (min < 0.0) {
      verticalMinimum = calculateVerticalPosition(min);
    }
    this.verticalZero = calculateVerticalPosition(0.0);

    graphics.setFont(oldFont);

    return effectiveViewport;
  }

  /**
   * Calculates the vertical position for the given value.
   */
  protected int calculateVerticalPosition(double value) {
    return verticalAnchor - (int) ((value - min) * verticalScale);
  }

  /**
   * Calculates the base scale line position for the chart.
   */
  protected int calculateBaseLinePosition() {
    double nn = 0.0;
    if (model.getMinValue() > 0.0) {
      nn = model.getMinValue();
    }
    return calculateVerticalPosition(nn);
  }

}
```

A.30 Chart.java

```
/* file name  : Chart.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/26/2004 04:48:33
 * copyright  :
 *
 * modifications:
 *
 */


package backend;

import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.util.List;

public interface Chart {

  /** Constant for the attribute 'background-color'. */
  String ATTR_BACKGROUND_COLOR = "background-color";
  /** Constant for the attribute 'border-width'. */
  String ATTR_BORDER_WIDTH = "border-width";
  /** Constant for the attribute 'title-font'. */
  String ATTR_TITLE_FONT = "title-font";
  /** Constant for the attribute 'area-labels'. */
  String ATTR_AREA_LABELS = "area-labels";
  /** Constant for the attribute 'segment-labels'. */
  String ATTR_SEGMENT_LABELS = "segment-labels";
  /** Constant for the attribute 'description-font'. */
  String ATTR_DESCRIPTION_FONT = "description-font";
  /** Constant for the attribute name 'description-pattern'. */
  String ATTR_DESCRIPTION_PATTERN = "description-pattern";
  /** Constant for the attribute 'description-line-width'. */
  String ATTR_DESCRIPTION_LINE_WIDTH = "description-line-width";
  /** Constant for the attribute 'description-color'. */
  String ATTR_DESCRIPTION_COLOR = "description-color";
  /** Constant for the attribute 'value-format'. */
  String ATTR_VALUE_FORMAT = "value-format";
  /** Constant for the attribute 'percent-format'. */
  String ATTR_PERCENT_FORMAT = "percent-format";
  /** Constant for the attribute 'segment-colors'. */
  String ATTR_SEGMENT_COLORS = "segment-colors";
  /** Constant for the attribute 'padding'. */
  String ATTR_PADDING = "padding";
  /** Constant for the attribute 'padding-left'. */
  String ATTR_PADDING_LEFT = "padding-left";
  /** Constant for the attribute 'padding-right'. */
  String ATTR_PADDING_RIGHT = "padding-right";
  /** Constant for the attribute 'padding-top'. */
  String ATTR_PADDING_TOP = "padding-top";
  /** Constant for the attribute 'padding-bottom'. */
  String ATTR_PADDING_BOTTOM = "padding-bottom";
  /** Constant for the attribute 'shadow-color'. */
  String ATTR_SHADOW_COLOR = "shadow-color";
  /** Constant for the attribute 'shadow-offset'. */
  String ATTR_SHADOW_OFFSET = "shadow-offset";
 /** Constant for the attribute 'line-width' */
  String ATTR_LINE_WIDTH = "line-width";
  /** Constant for the attribute 'scale-mark-length' */
  String ATTR_SCALE_MARK_LENGTH = "scale-mark-length";
  /** Constant for the attribute 'view-legend' */
  String ATTR_VIEW_LEGEND = "view-legend";
  /** Constant for the attribute 'value-step-size' */
  String ATTR_VALUE_STEP_SIZE = "value-step-size";
  /** Constant for the attribute 'view-scale-arrows' */
  String ATTR_VIEW_SCALE_ARROWS = "view-scale-arrows";
  /** Constant for the attribute 'scale-arrowhead-length' */
  String ATTR_SCALE_ARROWHEAD_LENGTH = "scale-arrowhead-length";
  /** Constant for the attribute 'scale-arrowhead-length' */
  String ATTR_SCALE_ARROWHEAD_WIDTH = "scale-arrowhead-width";
  /** Constant for the attribute 'scale-arrowhead-style' */
  String ATTR_SCALE_ARROWHEAD_STYLE = "scale-arrowhead-style";
  /** Constant for the attribute 'segment-mark-style' */
  String ATTR_SEGMENT_MARK_STYLE = "segment-mark-style";
  /** Constant for the attribute 'view-grid' */
  String ATTR_VIEW_GRID = "view-grid";
  /** Constant for the attribute 'grid-color'*/
  String ATTR_GRID_COLOR = "grid-color";
  /** Constant for the attribute 'grid-bgcolor' */
  String ATTR_GRID_BACKGROUND_COLOR = "grid-bgcolor";
  /** Constant for the attribute 'fill-area'*/
  String ATTR_FILL_AREA = "fill-area";

  /**
   * Gets the chart model.
   */
  ChartModel getModel();
  /**
   * Sets the chart model.
   * @exception ChartException In case the model contained
   * invalid data for the chart.
   */
  void setModel(ChartModel model) throws ChartException;

  /**
   * Tells, whether the chart supports negative values or not.
   * @return true, in case the chart supports negative values else false.
   */
  boolean supportsNegativeValues();

  /**
   * Gets a list of supported attributes for the chart.
   * @return The map of supported attributes for the chart.
   */
  List getSupportedAttributes();

  /**
   * Gets the chart attributes.
   * @return The attribute set.
   */
  ChartAttributes getAttributes();
  /**
   * Sets the chart attributes.
   */
  void setAttributes(ChartAttributes attributes);

  /**
   * Draws the chart to the given chart output.
   */
  Rectangle2D draw(Rectangle2D viewport, Graphics2D graphics)
    throws ChartException;
}
```

A.31 ChartAttributeDefinition.java

```
/* file name  : ChartAttributeDefinition.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/26/2004 05:33:37
 * copyright  :
 */


package backend;


public class ChartAttributeDefinition {

  /** Constant for boolean type. */
  public static final int TYPE_BOOLEAN = 1;
  /** Constant for integer type. */
  public static final int TYPE_INT = 2;
  /** Constant for double type. */
  public static final int TYPE_DOUBLE = 3;
  /** Constant for float type. */
  public static final int TYPE_FLOAT = 4;
  /** Constant for String type. */
  public static final int TYPE_STRING = 5;
  /** Constant for Date type. */
  public static final int TYPE_DATE = 6;
  /** Constant for Color type. */
  public static final int TYPE_COLOR = 7;
  /** Constant for font type. */
  public static final int TYPE_FONT = 8;
  /** Constant for Format type. */
  public static final int TYPE_FORMAT = 9;
  /** Constant for generic Object type. */
  public static final int TYPE_OBJECT = 10;

  /** The name attribute. */
  private String name = null;
  /** The parameter type. */
  private int type = TYPE_STRING;
  /** The list flag. */
  private boolean list = false;
  /** The optional flag. */
  private boolean optional = false;

  /**
   * Constructs a ChartAttributeDefinition instance with specific attributes.
   */
  public ChartAttributeDefinition(String name, int type, boolean list, boolean
optional) {
    this.name = name;
    this.type = type;
    this.list = list;
    this.optional = optional;
  }

  /**
   * Gets the parameter name.
   * @return The parameter name.
   */
  public String getName() {
    return this.name;
  }

  /**
```

```
   * Gets the parameter type.
   * @return The parameter type.
   */
  public int getType() {
    return this.type;
  }

  /**
   * Gets the list flag
   * @return The list flag.
   */
  public boolean isList() {
    return this.list;
  }

  /**
   * Gets the optional flag
   * @return The optional flag.
   */
  public boolean isOptional() {
    return this.optional;
  }

}
```

A.32 ChartAttributes.java

```java
/* file name  : ChartAttributes.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/16/2004 06:00:29
 */


package backend;

import java.awt.Color;
import java.awt.Font;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ChartAttributes {

  /** The chart attributes. */
  private Map attributes = null;

  /**
   * Constructs a ChartAttributes instance
   */
  public ChartAttributes() {
    this.attributes = new HashMap();
  }

  /**
   * Constructs a ChartChartAttributes instance with specific attributes.
   * @param attributes The chart attributes.
   */
  public ChartAttributes(Map attributes) {
    this.attributes = attributes;
  }

  /**
   * Check whether an attribute is defined.
   */
  public boolean isDefined(String name) {
    return attributes.get(name) != null;
  }

  /**
   * Gets a boolean attribute.
   */
  public boolean getBoolean(String name) {
    Boolean value = (Boolean) attributes.get(name);
    if (value == null) {
      return false;
    }
    return value.booleanValue();
  }
  /**
   * Sets a boolean attribute.
   */
  public void setBoolean(String name, boolean value) {
    attributes.put(name, new Boolean(value));
  }

  /**
   * Gets a integer attribute.
   */
  public int getInt(String name) {
```

```
  Number value = (Number) attributes.get(name);
  if (value == null) {
    return 0;
  }
  return value.intValue();
}
/**
 * Sets a integer attribute.
 */
public void setInt(String name, int value) {
  attributes.put(name, new Integer(value));
}

/**
 * Gets a long attribute.
 */
public double getLong(String name) {
  Number value = (Number) attributes.get(name);
  if (value == null) {
    return 0L;
  }
  return value.longValue();
}
/**
 * Sets a long attribute.
 */
public void setLong(String name, long value) {
  attributes.put(name, new Long(value));
}

/**
 * Gets a float attribute.
 */
public float getFloat(String name) {
  Number value = (Number) attributes.get(name);
  if (value == null) {
    return 0.0F;
  }
  return value.floatValue();
}
/**
 * Sets a float attribute.
 */
public void setFloat(String name, float value) {
  attributes.put(name, new Float(value));
}

/**
 * Gets a double attribute.
 */
public double getDouble(String name) {
  Number value = (Number) attributes.get(name);
  if (value == null) {
    return 0.0;
  }
  return value.doubleValue();
}
/**
 * Sets a double attribute.
 */
public void setDouble(String name, double value) {
  attributes.put(name, new Double(value));
}

/**
```

```
 * Gets a String attribute.
 */
public String getString(String name) {
  Object value = attributes.get(name);
  if (value == null) {
    return null;
  }
  return value.toString();
}
/**
 * Sets a String attribute.
 */
public void setString(String name, String value) {
  attributes.put(name, value);
}

/**
 * Gets a String array attribute.
 */
public String[] getStringArray(String name) {
  List values = getList(name);
  int numValues = values.size();
  String[] strings = new String[numValues];
  for (int i = 0; i < numValues; i++) {
    Object value = values.get(i);
    if (value != null) {
      strings[i] = value.toString();
    }
  }
  return strings;
}

/**
 * Gets a Font attribute.
 */
public Font getFont(String name) {
  Font value = (Font) attributes.get(name);
  return value;
}
/**
 * Sets a Font attribute.
 */
public void setFont(String name, Font value) {
  attributes.put(name, value);
}

/**
 * Gets a Color attribute.
 */
public Color getColor(String name) {
  Color value = (Color) attributes.get(name);
  return value;
}
/**
 * Sets a Color attribute.
 */
public void setColor(String name, Color value) {
  attributes.put(name, value);
}

/**
 * Gets a Color array attribute.
 */
public Color[] getColorArray(String name) {
  List values = getList(name);
```

```
    int numValues = values.size();
    Color[] colors = new Color[numValues];
    for (int i = 0; i < numValues; i++) {
      colors[i] = (Color) values.get(i);
    }
    return colors;
  }

  /**
   * Gets a Object attribute.
   */
  public Object getObject(String name) {
    Object value = attributes.get(name);
    return value;
  }
  /**
   * Sets a Object attribute.
   */
  public void setObject(String name, Object value) {
    attributes.put(name, value);
  }

  /**
   * Gets a list attribute.
   */
  public List getList(String name) {
    List value = (List) attributes.get(name);
    return value;
  }
  /**
   * Sets a list attribute.
   */
  public void setList(String name, List value) {
    attributes.put(name, value);
  }
}
```

A.33 ChartException.java

```java
/* file name  : ChartException.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/15/2004 05:28:04
 * copyright  :
 */

package backend;

public class ChartException extends Exception {

  /** The original cause. */
  private Throwable cause = null;

  /**
   * Constructs a ChartException instance with specific attributes.
   * @param message The error message.
   */
  public ChartException(String message) {
    super(message);
  }

  /**
   * Constructs a ChartException instance with specific attributes.
   * @param message The error message.
   * @param cause The original exception.
   */
  public ChartException(String message, Throwable cause) {
    super(message);
    this.cause = cause;
  }

  /**
   * Constructs a ChartException instance with specific attributes.
   * @param cause The original exception.
   */
  public ChartException(Throwable cause) {
    this.cause = cause;
  }

  /**
   * Gets the original cause of the exception.
   * @return The current value of property cause.
   */
  public Throwable getCause() {
    return cause;
  }

}
```

A.34 ChartFactory.java

```java
/* file name  : ChartFactory.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/20/2004 06:03:28
 */

package backend;

import java.util.HashMap;
import java.util.Map;

public class ChartFactory {

  /** The chart type registry. */
  private static final Map CHART_TYPES = new HashMap();

  /**
   * Registers the given chart class.
   */
  public static final void registerChart(String name, Class classToRegister) {
    synchronized (CHART_TYPES) {
      CHART_TYPES.put(name, classToRegister);
    }
  }

  static {
    // registering the chart types
    // We may register other types at here.
    registerChart("timeline", LineChart.class);
  }

  /**
   * Creates a chart instance.
   */
  public static Chart createChart(String type)
    throws ChartException {
    return createChart(type, null, null);
  }

  /**
   * Creates a chart instance.
   * @param type The name of the chart type to instanciate.
   * @param model The chart model.
   * @param attributes The chart attributes.
   * @return The created chart, or null if the chart type is not registered.
   * @exception ChartException In case the chart construction or configuration
failed.
   */
  public static Chart createChart(String type, ChartModel model,
                                  ChartAttributes attributes)
    throws ChartException {
    try {
      Class clazz = (Class) CHART_TYPES.get(type);
      // creating chart instance
      Chart chart = (Chart) clazz.newInstance();
      // configuring chart
      chart.setModel(model);
      chart.setAttributes(attributes);
      return chart;
    } catch (IllegalAccessException ex) {
      throw new ChartException("The chart type '"
                + type + "' is not allowed to be accessed: "
                + ex.getMessage(), ex);
    } catch (InstantiationException ex) {
      throw new ChartException("The chart type '"
                + type + "' cannot get instanciated: "
                + ex.getMessage(), ex);
    }
  }

}
```

A.35 ChartItem.java

```java
/* file name  : ChartItem.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/14/2004 05:25:02
 */


package backend;

import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;

public interface ChartItem {

  /**
   * Gets the position of the chart item.
   */
  ChartItemPosition getPosition();

  /**
   * Gets the output area ot the chart item.
   */
  Rectangle2D getOutputArea(Rectangle2D viewport, Graphics2D graphics);

  /**
   * Resets the calculated metrics to enforce re-calculation of the
   * item's output area.
   */
  void resetMetrics();

  /**
   * Draws the item to the given graphic instance.
   */
  void draw(Rectangle2D viewport, Graphics2D graphics);

  /**
   * Gets the width of the item box.
   */
  double getWidth(Rectangle2D viewport, Graphics2D graphics);

  /**
   * Gets the height of the item box.
   */
  double getHeight(Rectangle2D viewport, Graphics2D graphics);

  /**
   * Moves the item position.
   */
  void move(double x, double y);

  /**
   * Moves the item to the desired position.
   */
  void moveTo(double x, double y);
}
```

A.36 ChartItemPosition.java

```
/* file name  : ChartItemPosition.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/16/2004 06:08:52
 */


package backend;

public class ChartItemPosition {

  /** Constant telling that the item should be
   * placed at an absolutely defined position.
   */
  public static final short ABSOLUTE = 0;

  /** Constant telling that the item should be placed at the left side. */
  public static final short LEFT = 1;
  /** Constant telling that the item should be centered horizontally. */
  public static final short CENTER = 2;
  /** Constant telling that the item should be placed at the right side. */
  public static final short RIGHT = 3;

  /** Constant telling that the item should be placed at the top side. */
  public static final short TOP = 4;
  /** Constant telling that the item should be centered vertically. */
  public static final short MIDDLE = 5;
  /** Constant telling that the item should be placed at the bottom side. */
  public static final short BOTTOM = 6;

  /** The hozizontal position mode. */
  private short horizontalPositionMode = ABSOLUTE;
  /** The vertical position mode. */
  private short verticalPositionMode = ABSOLUTE;

  /** The absolute hozizontal position. */
  private double x = 0.0;
  /** The absolute vertical position. */
  private double y = 0.0;

  /**
   * Constructs a ChartItemPosition instance with specific attributes.
   */
  public ChartItemPosition(short horizontalPositionMode, short
verticalPositionMode) {
    this.horizontalPositionMode = horizontalPositionMode;
    this.verticalPositionMode = verticalPositionMode;
  }

  /**
   * Constructs a ChartItemPosition instance with specific attributes.
   */
  public ChartItemPosition(double x, short verticalPositionMode) {
    this.x = x;
    this.verticalPositionMode = verticalPositionMode;
  }

  /**
   * Constructs a ChartItemPosition instance with specific attributes.
   */
  public ChartItemPosition(short horizontalPositionMode, double y) {
    this.horizontalPositionMode = horizontalPositionMode;
    this.y = y;
  }

  /**
   * Constructs a ChartItemPosition instance with specific attributes.
   */
  public ChartItemPosition(double x, double y) {
    this.x = x;
    this.y = y;
  }

  /**
   * Gets the horizontal position mode.
   */
  public short getHorizontalPositionMode() {
    return horizontalPositionMode;
  }

  /**
   * Gets the vertical position mode.
   */
  public short getVerticalPositionMode() {
    return verticalPositionMode;
  }

  /**
   * Gets the absolute horizontal position.
   */
  public double getX() {
    return x;
  }
  /**
   * Sets the absolute horizontal position.
   */
  public void setX(double x) {
    this.x = x;
  }

  /**
   * Gets the absolute vertical position mode.
   */
  public double getY() {
    return y;
  }
  /**
   * Sets the absolute horizontal position.
   */
  public void setY(double y) {
    this.y = y;
  }
}
```

A.37 ChartModel.java

```java
/* file name  : ChartModel.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/17/2004 05:11:33
 */


package backend;

import java.util.List;

public interface ChartModel {

  /**
   * Gets the chart title.
   */
  TextChartItem getTitle();
  /**
   * Sets the chart title.
   */
  void setTitle(TextChartItem title);

  /**
   * Gets the chart data as a list of double arrays.
   */
  List getChartData();
  /**
   * Sets the chart data as a list of double arrays.
   */
  void setChartData(List chartData);

  /**
   * Gets the chart value descriptions as a list of Strings.
   */
  List getValueDescriptions();
  /**
   * Sets the chart value descriptions as a list of Strings.
   */
  void setValueDescriptions(List valueDescriptions);

  /**
   * Gets the minimum value from the chart data.
   */
  double getMinValue();
  /**
   * Sets the minimum value from the chart data.
   */
  void setMinValue(double minValue);

  /**
   * Gets the maximum value from the chart data.
   */
  double getMaxValue();
  /**
   * Sets the maximum value from the chart data.
   */
  void setMaxValue(double maxValue);

  /**
   * Gets the number of segments.
   */
  int getNumSegments();
  /**
   * Gets the number of areas.
   */
  int getNumAreas();

}
```

A.38 DataLoader.java

```
/* file name  : DataLoader.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 12/03/2004 08:20:12
 */

package backend;

import java.io.*;
import java.util.* ;

public class DataLoader {

  /**
   * A very simple function to read data from a file
   * into a double array.
   * We just assume that the data files are well formated.
   * Otherwise, we should do more checking.
   *
   * @param filename, the name/path of data file.
   */
  public static double[] load(String filename){

    ArrayList list = new ArrayList();

    try {
      File inputFile = new File(filename);
      BufferedReader in = new BufferedReader(new FileReader(filename));

      String str;

      while ((str= in.readLine()) != null){
        list.add(str);
      }

      in.close();
    }
    catch (IOException ioe)
    {
      System.out.println(ioe);
    }

    double[] data = new double[ list.size() ];

    for(int i=0; i < list.size(); i++) {
      data[i] = Double.valueOf(list.get(i).toString()).doubleValue();
    }

    return data;
  }
}
```

A.39 DataPlotter.java

```
/* file name  : ChartTester.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 12/05/2004 11:56:21
 */

package backend;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Rectangle;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.*;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class DataPlotter {

   /**
   * To draw time series in line chart.
   * This a wrapper for underlying chart drawing functions.
   *
   * @param data  , the time series datas
   * @param ValueName , the data descriptions
   * @param Title , the chart title
   */

  // Added by Xin Li to control when the plot will close itself and return
  public static boolean get_back = false;

  public static void drawPlot(List data,
                  List ValueNames,
                  String title) {

    // change chartModel
    ChartModel cm = new BaseChartModel();


    cm.setTitle(new TextChartItem(
            new ChartItemPosition(ChartItemPosition.CENTER,
                ChartItemPosition.TOP),
            title,
            new Font("Helvetica",
                  Font.HANGING_BASELINE | Font.BOLD, 16))
        );

    // rearrange the data for chart plotting
    List chartData = new ArrayList();
    int datalength = Integer.MAX_VALUE;
    // we just show the data according to the smallest length
    double[] column;
    for( int i = 0; i < data.size(); i++) {
      column = (double[]) data.get(i);
      if (datalength > column.length )
      datalength = column.length;
    }
    // copy the data
```

```
for ( int i = 0; i < datalength; i++) {
  double[] dataseg = new double[ data.size() ];
  for (int j=0; j < data.size(); j++) {
    column = (double[]) data.get(j);
    dataseg[j] =  column[i];
  }
  chartData.add( dataseg );
}

cm.setChartData(chartData);

//set the chart attributes
ChartAttributes attrs = new ChartAttributes();

attrs.setList(Chart.ATTR_AREA_LABELS, ValueNames);

//set axix scale
List descriptions = new ArrayList();
for(int i = 0; i < datalength; i++) {
  descriptions.add(Integer.toString(i));
}
attrs.setList(Chart.ATTR_SEGMENT_LABELS, descriptions);

//add colors
List colors = new ArrayList();
colors.add(Color.blue);
colors.add(Color.green);
colors.add(Color.red);
colors.add(Color.orange);
colors.add(Color.yellow);
colors.add(Color.magenta);
colors.add(Color.cyan);
colors.add(Color.pink);
attrs.setList(Chart.ATTR_SEGMENT_COLORS, colors);

// set up other default attributes
attrs.setString(Chart.ATTR_DESCRIPTION_PATTERN, "%s (%v Parts, %p%%)");
attrs.setFloat(Chart.ATTR_BORDER_WIDTH, 1.0F);
attrs.setFloat(Chart.ATTR_DESCRIPTION_LINE_WIDTH, 2.0F);
attrs.setFloat(Chart.ATTR_LINE_WIDTH, 5.0F);
attrs.setBoolean(Chart.ATTR_VIEW_LEGEND, true);
attrs.setBoolean(Chart.ATTR_FILL_AREA, false);
attrs.setFloat(Chart.ATTR_VALUE_STEP_SIZE, 20);
attrs.setString(Chart.ATTR_SEGMENT_MARK_STYLE, "center");


// draw the chart
try {
  get_back = false;
  final Chart chart = ChartFactory.createChart("timeline", cm, attrs);

  JFrame frame = new JFrame(title);
  frame.getContentPane().setLayout(new GridBagLayout());
  GridBagConstraints gbc = new GridBagConstraints();
  gbc.gridx = 0;
  gbc.gridy = 0;
  gbc.fill = GridBagConstraints.BOTH;
  gbc.weightx = 1.0;
  gbc.weighty = 1.0;

  frame.getContentPane().add(new JPanel() {
    public void paint(Graphics graphics) {
      int width = this.getWidth();
      int height = this.getHeight();
      graphics.setColor(Color.white);
```

```
          graphics.fillRect(0, 0, width, height);
          graphics.setColor(Color.black);
          try {
            chart.draw(new Rectangle(30, 30, getWidth()-60, getHeight()-60),
                       (Graphics2D) graphics);
          } catch (ChartException e) {
            e.printStackTrace();
          }
        }
      }, gbc);

      frame.setSize(800, 600);
      frame.setLocation(100, 100);

      frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          get_back = true;
          //System.exit(0);
        }
      });

      while(!get_back)
      {
        frame.show();
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

A.40 DataWriter.java

```
/* file name  : DataWriter.java
 * authors    : Xin Li (xl74@columbia.edu)
 * created    : 12/15/2004 09:30:01
 */

package backend;

import java.io.*;
import java.util.* ;

public class DataWriter {

  /**
   * A very simple function to load data into a file
   * We just assume that the data files are well formated.
   * Otherwise, we should do more checking.
   *
   * @param filename, the name/path of data file.
   */
  public static void write(double[] data,String filename){

    ArrayList list = new ArrayList();

    try {
       BufferedWriter out = new BufferedWriter(new FileWriter(filename));
         for(int i=0; i<data.length; i++)
         {
           out.write(""+data[i]);
           out.newLine();
      out.flush();
         }
         out.close();
    }

    catch (IOException ioe)
    {
      System.out.println(ioe);
    }

  }
}
```

A.41 GeoBrownMove.java

```java
/* file name  : GeoBrownMove.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 12/04/2004 12:28:46
 */

package backend;

import java.util.*;

public class GeoBrownMove {

  /**
   * To simulate the geometric Brownian movtion.
   *
   * @param init  , the initial value of the time series
   * @param steps , how many steps should we simulate.
   * @param mu    , the "percentage drift",  if mu = 0.1 and
   *                the stochastic part was not present, the stock
   *                would rise by 10% in a year
   * @param sigma , the "percentage volatility", if sigma = 0.3 then
   *                roughly one standard deviation over a year would be 30%
   *                of the present price of the stock.
   * @return data , the simulated time series
   */
  public static double[] simulate(
              double init,
              int steps,
              double mu,
              double sigma) {
    double Z=0;
    Random rand  = new Random(); // Gaussian random variable output
    double[] data = new double[steps];

    data[0] = init;

    for(int i=1; i < steps; i++){
      Z = rand.nextGaussian();
      data[i] = data[i-1]*Math.exp(mu + 0.5 * sigma *Z);
    }

    return data;
  }

  public static double[] simulate(
              double init,
              int steps) {
     double[] data;
     data = backend.GeoBrownMove.simulate(init, steps, 0.01, 0.08);
     return data;
  }
}
```

A.42 LineChart.java

```
/* file name  : LineChart.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/20/2004 06:11:05
 */

package backend;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.GeneralPath;
import java.awt.geom.Rectangle2D;
import java.util.List;


public class LineChart extends BaseScaleChart {

  /**
   * Rules, whether the area below the line is filled or not.
   */
  public boolean getFillArea() {
    if (attributes.isDefined(ATTR_FILL_AREA)) {
      return attributes.getBoolean(ATTR_FILL_AREA);
    }
    return false;
  }

  public Rectangle2D draw(Rectangle2D viewport, Graphics2D graphics)
    throws ChartException {
    viewport = super.draw(viewport, graphics);
    if ((model == null) || (model.getChartData() == null)) {
      return viewport;
    }

    // calculating the effective viewport
    Rectangle effectiveViewport = calcViewport(viewport, graphics);
    // drawing the scale
    drawScale(viewport, effectiveViewport, graphics);
    // drawing the legend
    drawLegend(viewport, graphics);

    graphics.setStroke(new BasicStroke(getLineWidth(),
              BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));

    /** The segment colors. */
    Color[] segmentColors = attributes.getColorArray(ATTR_SEGMENT_COLORS);

    List chartData = model.getChartData();
    int numSegments = chartData.size();

    boolean fillArea = getFillArea();
    int baseLineY = calculateBaseLinePosition();


    GeneralPath[] segmentPaths = new GeneralPath[model.getNumAreas()];
    GeneralPath[] segmentAreas = new GeneralPath[model.getNumAreas()];

    for (int i = 0; i < numSegments; i++) {
      double[] row = (double[]) chartData.get(i);
      int curPos = horizontalPositions[i];
```

```
        for (int j = 0; j < row.length; j++) {
          if (i == 0) {
            int yPos = calculateVerticalPosition(row[j]);
            segmentPaths[j] = new GeneralPath();
            segmentPaths[j].moveTo(curPos, yPos);
            if (fillArea) {
              segmentAreas[j] = new GeneralPath();
              segmentAreas[j].moveTo(curPos, baseLineY);
              segmentAreas[j].lineTo(curPos, yPos);
            }
          } else {
            int yPos = calculateVerticalPosition(row[j]);
            segmentPaths[j].lineTo(curPos, yPos);
            if (fillArea) {
              segmentAreas[j].lineTo(curPos, yPos);
              if (i == numSegments - 1) {
                // closing area segment
                segmentAreas[j].lineTo(curPos, baseLineY);
              }
            }
          }
        }
      }

      for (int j = 0; j < segmentPaths.length; j++) {
        Color color = segmentColors[j];
        if (fillArea) {
          graphics.setColor(getFillColor(color));
          graphics.fill(segmentAreas[j]);
        }
        graphics.setColor(color);
        graphics.draw(segmentPaths[j]);
      }

      return viewport;
    }

    /**
     * Computes the fill color for value area.
     */
    private Color getFillColor(Color color)  {
      int r = color.getRed();
      int g = color.getGreen();
      int b = color.getBlue();
      return new Color(r, g, b, 128);
    }
  }
```

A.43 PrintFormat.java

```java
/*
 * Created on 2004-12-16
 *
 */


/**
 * @author xinli
 *
 *  library functions for formatted output
 */

package backend;

import java.text.*;
import java.util.*;

public class PrintFormat {

  public static void Print_Format(double a, int blankLength)
  {
    NumberFormat NFT = NumberFormat.getInstance();
    NFT.setMinimumFractionDigits(2);
    NFT.setMaximumFractionDigits(2);
    int n = Double.toString(a).length();
    System.out.print(NFT.format(a));
    for(int i = 0; i < blankLength- n; i++ )
      System.out.print(" ");
  }

  public static void Print_Format(String a, int blankLength)
  {
    int n = a.length();
    System.out.print(a);
    for(int i = 0; i < blankLength - n; i++)
       System.out.print(" ");

  }


  public static void Print_Format(int a, int blankLength)
  {

    int n = Integer.toString(a).length();
    System.out.print(a);
    for(int i = 0; i < blankLength/2 - n; i++ )
      System.out.print(" ");
  }
}
```

A.44 Statistics.java

```java
/* file name  : Statistics.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 12/05/2004 10:51:26
 */

package backend;

/**
 * A utility class that provides some simple statistical functions.
 */
public abstract class Statistics {

 /**
  * Returns the mean of an array of numbers.
  */
  public static double calMean(final double[] data) {
    double result = 0.0;
    if (data != null && data.length > 0) {
      double sum = 0.0;
      int counter = 0;
      for (; counter < data.length; counter++) {
        sum = sum +data[counter];
      }
      result = (sum / counter);
    }
    return result;
  }

  /**
   * Returns the standard deviation of a set of numbers.
   */
  public static double calStdDev(final double[] data) {
    final double avg = calMean(data);
    double sum = 0.0;

    for (int counter = 0; counter < data.length; counter++) {
      final double diff = data[counter] - avg;
      sum = sum + diff * diff;
    }

    return Math.sqrt(sum / (data.length - 1));
  }

  /**
   * Calculates the correlation between two datasets.
   * Both arrays should contain the same number
   * of items.  Null values are treated as zero.
   *
   * @param data1  the first dataset.
   * @param data2  the second dataset.
   *
   * @return The correlation.
   */
  public static double calCorrelation(final double[] data1, final double[] data2)
{
    //check the parameters at first
    if (data1.length != data2.length) {
      throw new IllegalArgumentException(
        "'data1' and 'data2' arrays must have same length."
      );
    }
    final int n = data1.length;
  double sumX = 0.0;
  double sumY = 0.0;
  double sumX2 = 0.0;
  double sumY2 = 0.0;
  double sumXY = 0.0;
  double x = 0.0;
  double y = 0.0;
  for (int i = 0; i < n; i++) {
    x = data1[i];
    y = data2[i];
    sumX = sumX + x;
    sumY = sumY + y;
    sumXY = sumXY + (x * y);
    sumX2 = sumX2 + (x * x);
    sumY2 = sumY2 + (y * y);
  }

  return (n * sumXY - sumX * sumY)
        / Math.pow((n * sumX2 - sumX * sumX)
          * (n * sumY2 - sumY * sumY), 0.5);
}

/**
 * Returns a data set for a moving average on the data set passed in.
 *
 * @param xData  an array of the x data.
 * @param yData  an array of the y data.
 * @param period  the number of data points to average
 *
 * @return a double[][] the length of the data set in the first dimension,
 *         with two doubles for x and y in the second dimension
 */
public static double[] calMovingAverage(final double[] Data,
                                        final int period) {
  // check arguments

  if (period > Data.length) {
    throw new IllegalArgumentException(
      "calMovingAverage(...): period can't be longer than dataset.");
  }

  final double[] result = new double[Data.length - period];
  for (int i = 0; i < result.length; i++) {
    double sum = 0.0;
    for (int j = 0; j < period; j++) {
      sum += Data[i + j];
    }
    sum = sum / period;
    result[i] = sum;
  }

  return result;

}

/**
 * Calculates the present value of a cash flow.
 *
 * @param data    , the cash flow
 * @param interest, the current interest.
 *
 * @return The present value.
 */
public static double calPresentValue( double[] data, double interest) {
  double pv = 0.0;
```

```
    if (data != null && data.length > 0) {
      for (int i= data.length-1; i >= 0; i--) {
        pv = pv / (1+interest) + data[i];
      }
    }
    return pv;
  }

}
```

A.45 TextChartItem.java

```java
/* file name  : TextChartItem.java
 * authors    : Jian Huang (jh2353@columbia.edu)
 * created    : 11/15/2004 06:24:01
 */
package backend;

import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Rectangle2D;
import java.util.List;
import java.util.ArrayList;


public class TextChartItem extends BaseChartItem {

  /**
   * A class holding informations about each single line of the text item.
   */
  private class TextSegment {
    /** The text of the line. */
    private String segmentText;
    /** The horizontzal offset. */
    private double xOffset;
    /** The vertical offset. */
    private double yOffset;
    /** The segment width. */
    private double width;
    /** The line height. */
    private double height;

    /**
     * Constructs a TextSegment instance with specific attributes.
     */
    public TextSegment(String segmentText,
                  double xOffset,
                  double yOffset,
                  double width,
                  double height) {
      this.segmentText = segmentText;
      this.xOffset = xOffset;
      this.yOffset = yOffset;
      this.width = width;
      this.height = height;
    }

    /**
     * Getter method for the the property height.
     */
    public double getHeight() {
      return height;
    }
    /**
     * Getter method for the the property segmentText.
     */
    public String getSegmentText() {
      return segmentText;
    }
    /**
     * Getter method for the the property width.
     */
    public double getWidth() {
```

```
    return width;
  }
  /**
   * Getter method for the the property xOffset.
   */
  public double getXOffset() {
    return xOffset;
  }
  /**
   * Getter method for the the property yOffset.
   */
  public double getYOffset() {
    return yOffset;
  }
}

/** The text of the chart item. */
private String text = null;
/** A list of text segments. */
private List textSegments = new ArrayList();
/** The maximum with, zero means no limit. */
private int maxWidth = 0;
/** The font for the chart item. */
private Font font = null;
/** The horizontal alignment of the text. */
private int hAlign = ChartItemPosition.LEFT;

/**
 * Constructs a ChartTextItem instance with specific attributes.
 */
public TextChartItem(ChartItemPosition position, String text, Font font) {
  super(position);
  this.text = text;
  this.font = font;
}

/**
 * Constructs a ChartTextItem instance with specific attributes.
 */
public TextChartItem(ChartItemPosition position, int maxWidth, int hAlign,
                     String text, Font font) {
  this(position, text, font);
  this.maxWidth = maxWidth;
  this.hAlign = hAlign;
}

/**
 * Gets the item's text.
 */
public String getText() {
  return text;
}

/**
 * Gets the maximum width of the item box.
 */
public int getMaxWidth() {
  return maxWidth;
}

/**
 * Gets the item's horizontal alignment.
 */
public int getHAlign() {
  return hAlign;
```

```java
}

/**
 * Gets the item's font.
 */
public Font getFont() {
  return font;
}

/**
 * Calculates the output area of the text item.
 */
public Rectangle2D calculateOutputArea(Rectangle2D viewport,
                      Graphics2D graphics) {
  int xPos = 0;
  int yPos = 0;

  Font oldFont = graphics.getFont();
  graphics.setFont(this.font);
  Rectangle2D textMetrics =
        graphics.getFontMetrics().getStringBounds(text, graphics);
  double width = 0.0;
  double height = 0.0;
  if ((maxWidth > 0) && (textMetrics.getWidth() > maxWidth)) {
    // splitting text segments into parts to fit into maximum space
    char[] textChars = text.toCharArray();
    int segmentOffset = 0;
    int lastWhitespace = 0;
    double yOffset = 0.0;
    for (int i = 0; i < textChars.length; i++) {
      char c = textChars[i];
      switch (c) {
        case '\t':
        case ' ': {
          String segmentText = new String(textChars,
                        segmentOffset, i - segmentOffset);
          Rectangle2D segmentMetrics =
                        graphics.getFontMetrics().getStringBounds(
                                segmentText, graphics);
          if (segmentMetrics.getWidth() > maxWidth) {
            if (lastWhitespace > 0) {
              // going back to last whitespace
              for (int j = lastWhitespace; j > segmentOffset; j--) {
                if ((textChars[j] == '(') || (textChars[j] == '[')) {
                  // break before last paranthesis or bracket
                  lastWhitespace = j - 1;
                  break;
                }
              }
              segmentText = new String(textChars,
                      segmentOffset,
                      lastWhitespace - segmentOffset + 1);
              segmentMetrics =
                      graphics.getFontMetrics().getStringBounds(
                          segmentText, graphics);
              this.textSegments.add(new TextSegment(segmentText,
                    0.0, yOffset, segmentMetrics.getWidth(),
                    segmentMetrics.getHeight()));
              yOffset += segmentMetrics.getHeight();
              height += segmentMetrics.getHeight();
              width = Math.max(width, segmentMetrics.getWidth());
              --i;
              segmentOffset = lastWhitespace + 1;
              lastWhitespace = 0;
            } else {
                // don't know where to break, so adding (too) long segment.
                this.textSegments.add(new TextSegment(segmentText,
                        0.0, yOffset,
                        segmentMetrics.getWidth(),
                        segmentMetrics.getHeight()));
                yOffset += segmentMetrics.getHeight();
                height += segmentMetrics.getHeight();
                width = Math.max(width, segmentMetrics.getWidth());
                segmentOffset = i + 1;
                lastWhitespace = 0;
              }
            }
            lastWhitespace = i;
          }
          break;

          case '\n': {
            // line break found, breaking line
            String segmentText = new String(textChars,
                  segmentOffset, i - segmentOffset);
            Rectangle2D segmentMetrics =
                  graphics.getFontMetrics().getStringBounds(
                      segmentText, graphics);
            this.textSegments.add(new TextSegment(
                  segmentText, 0.0, yOffset,
                  segmentMetrics.getWidth(),
                  segmentMetrics.getHeight()));
            yOffset += segmentMetrics.getHeight();
            height += segmentMetrics.getHeight();
            width = Math.max(width, segmentMetrics.getWidth());
            segmentOffset = i + 1;
            lastWhitespace = 0;
          }
          break;
        }
      }
      if (segmentOffset < textChars.length) {
        String segmentText = new String(textChars,
                segmentOffset, textChars.length - segmentOffset);
        Rectangle2D segmentMetrics =
              graphics.getFontMetrics().getStringBounds(
                  segmentText, graphics);
        if (segmentMetrics.getWidth() > maxWidth) {
          if (lastWhitespace > 0) {
            // going back to last whitespace
            for (int j = lastWhitespace; j > segmentOffset; j--) {
              if ((textChars[j] == '(') || (textChars[j] == '[')) {
                // break before last paranthesis or bracket
                lastWhitespace = j - 1;
                break;
              }
            }
            segmentText = new String(textChars,
                segmentOffset,
                lastWhitespace - segmentOffset + 1);
            segmentMetrics =
                  graphics.getFontMetrics().getStringBounds(segmentText,
graphics);
            this.textSegments.add(new TextSegment(segmentText,
                  0.0, yOffset,
                  segmentMetrics.getWidth(),
                  segmentMetrics.getHeight()));
            yOffset += segmentMetrics.getHeight();
            height += segmentMetrics.getHeight();
            width = Math.max(width, segmentMetrics.getWidth());
            // adding the rest
            segmentOffset = lastWhitespace + 1;
            segmentText = new String(textChars, segmentOffset,
                textChars.length - segmentOffset);
            segmentMetrics =
                graphics.getFontMetrics().getStringBounds(segmentText,
graphics);
            this.textSegments.add(new TextSegment(segmentText, 0.0,
                  yOffset, segmentMetrics.getWidth(),
                  segmentMetrics.getHeight()));
            height += segmentMetrics.getHeight();
            width = Math.max(width, segmentMetrics.getWidth());
          } else {
            // don't know where to break, so adding (too) long segment.
            this.textSegments.add(new TextSegment(segmentText,
                    0.0, yOffset, segmentMetrics.getWidth(),
                    segmentMetrics.getHeight()));
            yOffset += segmentMetrics.getHeight();
            height += segmentMetrics.getHeight();
            width = Math.max(width, segmentMetrics.getWidth());
          }
        } else {
          this.textSegments.add(new TextSegment(segmentText,
                  0.0, yOffset,
                  segmentMetrics.getWidth(),
                  segmentMetrics.getHeight()));
          width = Math.max(width, segmentMetrics.getWidth());
          yOffset += segmentMetrics.getHeight();
          height += segmentMetrics.getHeight();
        }
      }
    } else {
      // no line breaks necessary
      width = textMetrics.getWidth();
      height = textMetrics.getHeight();
      this.textSegments.add(new TextSegment(text, 0.0, 0.0, width, height));
    }

    switch (position.getHorizontalPositionMode()) {
      case ChartItemPosition.ABSOLUTE:
        xPos = (int) (position.getX());
        break;
      case ChartItemPosition.LEFT:
        xPos = (int) viewport.getX();
        break;
      case ChartItemPosition.CENTER:
        xPos = (int) (viewport.getCenterX()  - width / 2.0);
        break;
      case ChartItemPosition.RIGHT:
        xPos = (int) (viewport.getX() + viewport.getWidth() - width);
        break;
    }

    switch (position.getVerticalPositionMode()) {
      case ChartItemPosition.ABSOLUTE:
        yPos = (int) (viewport.getY() + position.getY());
        break;
      case ChartItemPosition.TOP:
        yPos = (int) (viewport.getY());
        break;
      case ChartItemPosition.MIDDLE:
        yPos = (int) (viewport.getCenterY()  - height / 2.0);
        break;
      case ChartItemPosition.BOTTOM:
        yPos = (int) (viewport.getY() + viewport.getHeight() - height);
        break;
    }

    graphics.setFont(oldFont);
    return new Rectangle(xPos, yPos, (int) width, (int) height);
  }

  /**
   * Draws the item to the given graphic instance.
   */
  public void draw(Rectangle2D viewport, Graphics2D graphics) {
    Rectangle2D area = getOutputArea(viewport, graphics);
    Font oldFont = graphics.getFont();
    graphics.setFont(this.font);
    int numSegments = textSegments.size();
    for (int i = 0; i < numSegments; i++) {
      TextSegment segment = (TextSegment) textSegments.get(i);
      int xPos = (int) (area.getX() + segment.getXOffset());
      int yPos = (int) (area.getY() + segment.getHeight()
            + segment.getYOffset());
      // adjusting horizontal offset
      switch (hAlign) {
        case ChartItemPosition.RIGHT:
          xPos += (int) (area.getWidth() - segment.getWidth());
          break;
        case ChartItemPosition.CENTER:
          xPos += (int) ((area.getWidth() - segment.getWidth()) / 2.0);
          break;
      }
      graphics.drawString(segment.getSegmentText(), xPos, yPos);
    }
    graphics.setFont(oldFont);
    //graphics.draw(area);
  }
}
```