2006

# Implementing Built-in Properties for the Java Programming Language

Alexandre Alves
*San Jose State University*

CS298 Spring 2006

Writing Project

# Implementing Built-in Properties for the Java Programming Language

Advisor: Dr. Cay Horstmann

Student: Alexandre de Castro Alves

Computer Science Department

San José State University

# ABSTRACT

The purpose of this project is to improve the programming experience of using the Java language by implementing properties as a built-in facility. In this project, the Java compiler tool, and the Java documentation tool were modified. In addition, a new Java annotation processor that generates Java BeanInfo source files was created. These new features result in a more productive development environment for the Java programming language.

# Acknowledgments

I would like to thank my advisor Dr. Horstmann for his leadership, insights, and counseling. I would also like to thank my committee members, Dr. Taylor and Dr. Louden. Finally, I would like to dearly thank my wife, Erica, for her enduring patient and support.

# Table of Contents

# Table of Figures

# 1. Introduction

## 1.1. Project Description

Properties are an important language feature natively present in many programming languages, such as C#, Visual Basic, etc. Properties are used for identification or characterization of an entity. For example, one may define properties for a class in OOP. Properties are useful for specifying, that is, marking certain aspects of the entity that are to be highlighted to a client of this entity, and in doing so, allowing this client to deduce special behavior from these markings. For example, a Widget class can define properties to represent its visual aspects, which can later be used by a GUI builder tool.

The Java programming language does not directly support properties. Properties of a Java class are deduced by having its methods follow certain naming conventions, such as getX and setX. The naming conventions defined by Java are specified at the JavaBeans API specification [1].

This mechanism is both cumbersome as it demands boilerplate code to be authored and is error-prone. As it is not an explicit language feature, compile-time type checking is also not provided.

In this project we intend to solve this problem by implementing properties as a first class feature of the Java language. To allow for this, we have changed the Java language and its supporting tools, including the Java compiler tool (i.e. javac), and the Java documentation tool (i.e. javadoc). It is the intent of this project to integrate Java properties seamlessly into the Java development environment.

### 1.1.1. An Example

The best way to describe this project is by providing an example of its usage.

The following example is of a simple Student Java class, which has several public methods:

```
public class Student {
        private String name;
```

```
        private String major;
        private String [] classes;
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getMajor() {
                return major;
        }
        public void setMajor(String major) {
                this.major = major;
        }
        public String[] getClasses() {
                return classes;
        }
        public void setClasses(String[] classes) {
                this.classes = classes;
        }
}
```

The following code fragment presents the same example ported to use the Java property feature implemented in this project:

```
public class Student {
        @Property private String name;
        @Property private String major;
        @Property private String [] classes;
}
```

Both examples are functionally identical, but obviously the latter one is easier to program, to comprehend, to maintain, and to support with tooling, such as code generation wizards.

## 1.2. Related Work in the Field

The importance of properties is well recognized. We will demonstrate how existing tools deal with another 'flavor' of properties, that is, JavaBeans' properties. JavaBeans are defines in section 2.2. Particularly we will look into Eclipse, the Spring framework, and the C# language.

### 1.2.1. Eclipse

First consider the Eclipse development environment [9]. Eclipse is one of the most popular IDEs for Java programming. It includes a Java editor, providing features as syntax highlighting, and automatic code re-factoring. In particular, it provides a code editing feature where one can select class fields and request that the corresponding getter and/or setter be generated in the source file being edited.

The following picture illustrates this idea using Eclipse version 3.1:

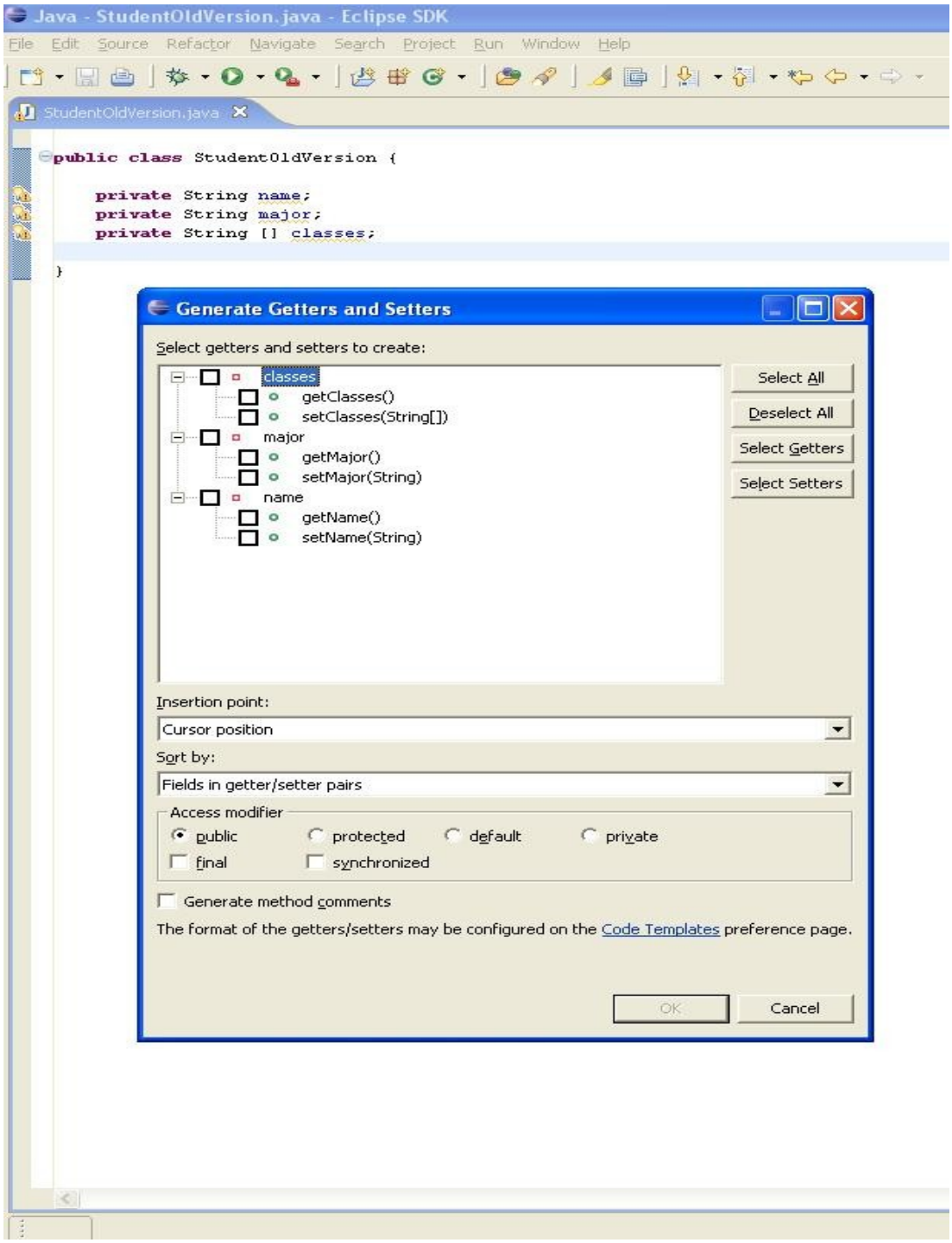


**Figure 1 Eclipse Getter/Setter Generation**

The programmer selects a Java class in the Java editor, and then selects the menu option Generate Getters and Setters. Notably, the programmer can select which class fields

to use, the location (e.g. cursor position), and the access privilege (e.g. public, private) of the generated source.

Although this approach may improve the programmer's productivity, it still falls short of being optimal. The boiler-plate code generated for the getters and/or setters pollute the original source code, and no visual cues are provided for the properties.

### 1.2.2. Spring framework

The Spring framework [10] is a popular development framework for Java applications. It contains abstractions for handling persistence, transactions, web development, and other common enterprise features. It was designed to overcome the complexity of J2EE 1.4 technology [13], by minimizing dependencies among its components. This is done in great part due to its support of dependency injection. Dependency injection [11], also known as Inversion of Control (IoC), is a design pattern [8] where a primary entity gets injected with the dependencies it needs to function by decoupled secondary entities; thus the primary entity is not tightly coupled to references it uses to execute its role.

The implementation of the dependency injection mechanism of the Spring framework is mostly done through the use of JavaBeans.

Spring uses an XML [12] configuration file to setup the dependency injection. The following XML fragment injects (i.e. sets) Course into Student:

```
<bean id="myCourse" class="Course"/>
<bean id="student" class="Student">
  <property name="course"><ref bean="myCourse"/></property>
</bean>
```

The Student class must have a Course JavaBeans property, as illustrated in the following code fragment:

```
public class Student {
        private Course course;
```

```
        public void setCourse(Course value) {
            course = value;
        }
    }
```

The configuration mechanism of Spring is a clear example of a user of JavaBeans properties. Spring, and other examples like GUI builders, assumes that the client code does declare the methods that are needed to fulfill the implicit contract of JavaBeans properties.

### 1.2.3. The C# Language

The C# language, similarly to Java, is an object-oriented language, and as such, provides the usual data encapsulation features.

Data encapsulation is done through the use of classes or structures, encapsulating data members. The programmer can declare non-public data fields, and then provide a set of public getter and setter methods to access these data fields, just as expected. The following code fragment illustrates this:

```
class Student {
        private int age;
        public int getAge() {
                return age;
        }
        public void setAge(int myAge) {
                age = myAge;
        }
}
```

A typical usage of the Student class is:

```
student.setAge(30);
student.getAge();
```

C# Properties are a genuine extension of data fields. Again, let's consider the Student class, but now using the built-in property mechanism:

```
class Student {
        private int age;
        public int Age {
                get {
                        return age;
                }
                set {
                        age = value; // 'value' is assigned by the compiler
                }
        }
}
```

In this case, the usage pattern is:

```
student.Age = 30;
int myAge = student.Age;
```

Student class has two data members, the private data field age, and the public data property Age. Both are considered data members of the Student class, hence they share the same namespace and cannot have the same name.

The programmer may decide to omit the property's getter or the setter, but the programmer can only omit both access methods if the property is marked as abstract. Abstract properties are explained in the next section.

The usage pattern of data properties is similar to that of data fields, inheriting its simplicity, but with the added advantage of being protected by implied methods (e.g. getter and setter). In addition, it is evident in the source code which class members are properties and what are the getters and setters associated with them.

#### 1.2.3.1. Inheritance and Polymorphism

Data properties are inherited just like any other data member. The extended classes can override the property's getter and setter. This is illustrated in the following code fragment:

```
class Person {
        protected int age;
        public virtual int Age {
                get {
                        return age;
                }
        }
}
class Student {
        public override int Age {
                get {
                        Console.Write("Derived Class");
                        return age;
                }
        }
}
```

Note that the modifiers, like virtual and public, are associated to the property itself, and not to a specific getter or setter. This means that the programmer cannot declare different

modifiers for the getter and setter of the same property. For example, a data property cannot have a public getter and a protected setter.

Properties can also be abstract. Abstract properties do not define the implementation of their getters and setters; this must be done by the derived classes.

### 1.3. Technologies Used

This project is based upon Java technology and its supporting tools. In particular, it uses:

- Java Standard Edition (SE) 6.0 (a.k.a. Mustang)
- Java Reflection API
- JavaBeans API Specification
- JSR 175: A Metadata Facility for the Java Programming Language
- Ant 1.2
- JUnit

These technologies are explained in the following section.

This project modifies and extends the following components of Java SE:

- Java Compiler tool (i.e. javac)
- Java Documentation tool (i.e. javadoc)

## 2. Technical Background

### 2.1. Java Reflection API

The Java Reflection API allows the programmer to examine a class present in a Java Virtual Machine.

The API allows for the detailed examination of the classes' constructors, fields, and methods, as well as their modifiers. The API also allows for the examination of the classes' extension and implementation hierarchy.

These are the Reflection API types:

- java.lang.reflect.Class
- java.lang. reflect.Constructor

- java.lang. reflect.Method
- java.lang. reflect.Field
- java.lang. reflect.annotation.Annotation

Collectively, these types represent the metadata of a Java type.

### 2.2. JavaBeans API

The JavaBeans API is a simple, consistent API for third parties to create reusable software components and thus to promote the composition of software applications.
A Java Bean is defined as "... a reusable software component that can be manipulated visually in a visual tool." [1].
Java Beans have five main characteristics:

- Properties: Java Beans expose properties that define their component's interface;
- Methods: Java Beans expose public methods that other components may call upon;
- Event handling: Java Beans fire events, which other components can receive;
- Persistence: Java Beans are serializable, and may be stored and rebuilt at a later time;
- Introspection: the JavaBeans API defines ways to introspect, that is, discover the properties, methods, and events of a Java Bean;

Collectively, these characteristics allow Java Beans to be reusable and customizable by application builders.

#### 2.2.1 Java Bean Events

Events are a generic mechanism for decoupling interacting objects.
Listener objects register into source objects. Source objects will fire events notifying all registered listening objects of the source object's state changes. This is similar to the Observer design pattern [12]. The goal is to decouple the application logic, which is part of the user code and is implemented in the listeners, from source components, which are part of the infrastructure code.

If the source object allows for multiple listeners to be registered at a time, then the source object is called a multicast event source.

If the source object only allows for a single listener to be registered at a time, then the source object is called a unicast event source. Unicast event sources may throw a java.util.TooManyListenersException exception if a second listener tries to register.

The following code fragment illustrates this collaboration:

```
class MyEventListener extends PropertyChangeListener {
        ...
        void propertyChange(PropertyChangeEvent evt) {
                // do something with event
                Object newValue = evt.getNewValue();
                ...
        }
}
...
sourceComponent.addPropertyChangeListener(myEventListener);
```

MyEventListener first registers into the source component that fires property change events. When the event is fired, the corresponding method in the listener is trigged.

To allow for extensibility, the programmer can define own listener types and event types. In this case, the following rules must be observed:

- The Listener type must implement the tagging interface java.util.EventListener;
- The Event type must extend the class java.util.EventObject;
- Event sources must provide methods for event listeners to register using the following pattern:

```
public void addListenerType (ListenerType listener);
public void removeListenerType(ListenerType listener);
```

### 2.2.2. Java Bean Properties

Java Bean Properties are named attributes of a Java Bean that determine its behavior. Properties have no restrictions on their type.

#### 2.2.2.1. Accessor Methods

Readable properties are always accessed through a getter method. Writable properties are accessed through a setter method.
Both getter and setter method should adhere to the following naming convention:

```
public < PropertyType> get< PropertyName>();
public void set< PropertyName>(< PropertyType> a);
```

The only exception is for properties whose type is Boolean, in which case the naming convention for the getter method becomes:

```
public boolean is< PropertyName>();
```

Accessor methods may declare checked exceptions, which are considered application-level exceptions.
Properties do not necessarily have to define both a getter and a setter, but at least one of these must be defined.

#### 2.2.2.2. Indexed Properties

Properties can be part of an array of elements. These properties can be indexed by using an integer value. In the case of indexed properties, additional methods must be provided. These methods should adhere to the following naming convention:

```
public < PropertyElement> get< PropertyName>(int a);
public void set< PropertyName>(int a, < PropertyElement> b);
```

#### 2.2.2.3. Bounded Properties

Components may be interested in listening to changes on certain properties. These properties are called bounded properties, and provide a PropertyChangeListener interface for reporting updates.

#### 2.1.2.4. Constrained Properties

Properties may choose to restrict or validate updates done to its value. These properties are called constrained properties. Setter methods for constrained properties must support the exception PropertyVetoException to indicate that changes done by other components may be vetoed.

In addition, a VetoableChangeListener event listener can be used to report updates to a constrained property.

### 2.2.3. Java Bean Methods

By default all public methods, including the Java Bean accessor methods and event listener methods, are part of the public interface of a Java Bean.

### 2.2.4 Java Bean Introspection

At runtime or at component assembly time, the process of determining the methods, events, and properties of a Java Bean is called introspection.

Introspection can be realized in two forms, either by using Java reflection [13] to discover the methods of a class and then deduce its Java Bean's methods, events, and properties by applying the naming conventions previously explained; or by using BeanInfo.

A Java Bean implementor may choose to associate a BeanInfo class to its Java Bean class. The BeanInfo class programmatically provides information about the Java Bean methods, events, and properties. This is done by declaring the BeanInfo class name to be equal to the Java Bean class name with the added suffix of "BeanInfo". For example, considering our previous Student Java class, its corresponding BeaInfo class must be named StudentBeanInfo.

For either approaches a Introspector utility class is provided by the Java platform. The Introspector class introspects a Java Bean combining both mechanisms, and thus provides a uniform and concise way of working with Java Beans.

When BeanInfo is not provided, the introspection of a Java Bean is largely based upon design patterns, that is, an implicit contract determined by naming conventions. Contrary to this, the Java property feature implemented in this project provides an explicit contract.

### 2.3. Java Annotations (JSR 175)

Java annotation, or metadata, were introduced as a language feature in J2SE 5.0. Annotation is used to decorate a Java program and affects the way the program is treated by tools and other libraries.

Certain APIs require side-files to co-exist with programs. For example, Enterprise JavaBeans requires deployment descriptors, and JavaBeans make use of BeanInfo classes. Annotations were created to avoid or minimize the need for these side-files.

Annotations provide a standard way of associating metadata to Java declarations. These annotations were not meant to directly affect the program semantic, but rather to provide additional information to be used by tools and libraries.

The following code fragment is an example of a Java class annotated for J2EE:

```
@Entity Student student
```

Java Annotations define syntax for declaring annotation types, syntax for annotating Java declarations, and an API for accessing these annotations.

#### 2.3.1. Annotation Types

Annotation types are very similar to Java interfaces.

Annotation type declarations precede the keyword interface by the character '@'. Methods declared for annotation types must follow several restrictions:

- The methods must not have any parameters;
- The methods must return either primitive types, a Class type, an enumeration type, an array type, or an annotation type;

- The methods must not have a throw clause.

However, annotation type methods can associate default values to their declarations.

Annotation types themselves can be annotated; these annotations that annotate other annotation types are called meta-annotations.

The Java platform predefines several annotation types, these are:

- java.lang.annotation.Target
- java.lang.annotation.RetentionPolicy
- java.lang.annotation.Inherited
- java.lang.annotation.Override
- java.lang.annotation.SuppressWarnings
- java.lang.annotation.Deprecated

### 2.3.2. Annotations

Annotations may be used as modifiers in any package, class, field, method, parameter, constructor, or local variable declaration of a Java program. The annotation consists of the name of an annotation type and of zero or more element-value pairs, where the element maps to a method name of the annotation type.

### 2.3.3. The Annotation Processor Tool (APT)

APT is a command line tool that can be used to process annotations in Java source files. This is done by implementing annotation processors. Annotation processors have a read-only design-time view of Java classes, and can be used to generate artifacts and other Java source files.

APT works in cycles, it processes Java source files, which may cause the generation of more source files to be processed, and then compiled.

## 2.4. Apache Ant

Ant is a build tool. Similarly to other build tools, such as make, and nmake, Ant allows one to specify how software products are to be compiled, packaged, tested, etc. The main advantage of Ant is that it is based upon Java, so it can be extended by adding Java code

to it. Another advantage is that Ant build files are specified in XML, which makes Ant more suitable for tooling. The Eclipse IDE includes a source editor for Ant that supports code complete, syntax highlighting, and syntax checking of Ant build files.
Ant commands are specified as tasks. Several tasks are already provided by default and can be used for the most common build functions. Some examples of Ant tasks are:

- copy
- mkdir
- javac
- tar
- untar
- rename

Users can add new tasks to Ant by writing Java program that use the Java API of Ant.

# 3. Specification

This project implements built-in support for properties. We name this feature Java properties to distinguish it from JavaBeans properties.
This section specifies what Java properties are and how they are used in a Java program.

## 3.1. Property Field

A class field is identified as a Java property by setting the annotation type javax.lang.Property to it. A field annotated with the Property annotation type is thus named a property field. In its simplest form, a property field has an implicit getter and setter method named using the JavaBeans naming convention. These implicit getter and setter methods associated to the property field are named property access methods.
Let's consider the following Student class that contains the property field name:

```
public class Student {
        @Property private String name;
}
```

This is equivalent to:

```
public class Student {
        private String name;
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
}
```

### 3.2. Property Operator

The new operator .@, named property operator, is defined to access properties. It follows the pattern of an object selector, that is, it is an infix operator where the prefix must be a valid object reference and its suffix must be the identifier of a property field.
Considering our previous Student class, one may access its name property as:

```
String studentName = student.@name;
```

Note that in this case, the property operator is used in the context of a RHS (riht-hand-side) expression. When used as a RHS expression, the property operator maps into the getter method of the field. Should the property operator be used in the context of a LHS (left-hand-side) expression, it will map into the setter method of the field, as the code fragment outlines:

```
student.@name = “Alex”;
```

The advantage of a property field is obvious. It avoids the developer from having to write boilerplate code. There are also other less apparent advantages that we will discuss later on. However, the benefit of the property operator is not directly evident. To understand it, we first need to look into another feature of the annotation type Property, the ability to specify the property access method names.

### 3.3. Property Type Getter and Setter Attributes

The Property annotation type has the attributes Getter and Setter. Both these attributes have a default value of an empty string. These attributes can be used to specify the name of the getter and setter methods that will be associated to the property field. If the default value of an empty string is used, then the getter and setter property access methods will be named using the JavaBeans naming convention as we have seen previously.
Continuing with our Student class example:

```
public class Student {
        @Property(getter="getStudentName", setter="setStudentName")
        private String name;
}
```

As expected, this is equivalent to:

```
public class Student {
        private String name;
        public String getStudentName() {
                return name;
        }
        public void setStudentName(String name) {
                this.name = name;
        }
}
```

Now we can go back to our investigation of the property operator .@. The property operator allows the developer to reference the correct property access method without knowing the property access method name. This uncoupling allows the developer to change the property access method name without disrupting the property client code.
In the previous example, although we changed the getter and setter name from getName and setName to getStudentName and setStudentName, we don't have to make any changes to the client code, which remains the same:

```
String studentName = student.@name;
student.@name = "Alex";
```

### 3.4. Custom Property Access Methods

So far we have only seen scenarios where the property access methods did not exist explicitly in the class declaration that contains the property field, but that does not need to be the case. It is permitted for methods that match the property access methods' signatures to exist in the class declaration. This is useful in the case of legacy Java code.
The developer has the flexibility to select any existing method. The restriction is that the selected methods must match the return type and parameters of the expected property access method's signature.
Let's consider the following example:

```
public class Student {
        @Property(getter="getStudentName") private String name;
        public String getStudentName() {
                return name;
        }
        public void setStudentName(String name) {
                this.name = name;
        }
}
```

In this case, the methods getStudentName and setStudentName are already defined in the Student class. The property field Name is configured to use the getter method getStudentName; hence, as this method already exists in the Student class, there is no need to implicitly declare this method. This is contrary to the setter method. As the property field does not specify the name of the setter method, the default method name of setName, defined in the JavaBeans naming convention specification, will be used. As the Student class does not define a setName method, a setName method will be automatically generated by the field property Name.
Therefore the equivalent class for the previous example is:

```
public class Student {
        private String name;
        public String getStudentName() {
                return name;
        }
        public void setStudentName(String name) {
                this.name = name;
        }
        public void setName(String name) {
                this.name = name;
        }
}
```

Aside from being able to leverage legacy code, this allows the developer to write custom property access methods, as exemplified in the following code fragment:

```
public class Student {
        @Property(getter="getStudentName") private String name;
        public String getStudentName() {
                // Lazy loading
```

```
            if (name == null) {
                name = deserialize();
            }
            return name;
        }
    }
```

### 3.5. Indexed Properties

As we have seen previously, the type of the property field determines the signature of the property access methods. The property field type may be of any Java class type or any Java primitive type. A property field of array type is called an indexed property and is considered a special case.

Indexed properties have two additional property access methods, which are used to access individual items of the array object. In another words, an indexed property has a total of four property access methods, one getter and setter that works with the array object as a whole, and an additional getter and setter that works with one item of the array object. These additional access methods declare one additional parameter, the index parameter, which is used to select an item in the array object. This follows the same pattern established by the JavaBeans specification for JavaBeans indexed properties.

Let's extend our Student class to use indexed properties:

```
public class Student {
    @Property private String [] names;
}
```

This is equivalent to:

```
public class Student {
    private String [] names;
    public String [] getNames() {
```

```
            return names;
        }
        public void setNames(String [] names) {
            this.names = names;
        }
        public String getNames(int index) {
            return names[i];
        }
        public void setNames(int index, String name) {
            names[i] = name;
        }
}
```

Note that both getter methods and both setter methods share the same name, that is, the methods are overloaded.

There is one caveat to the usage of indexed properties, consider the following scenario:

```
student.@name[i]
```

This is equivalent to student.getName()[i] instead of student.getName(i), as it would be expected. To able to support the latter, a new operator .@[] would have to be defined. However, as indexed properties are rarely used in practice, it was felt that this is not necessary. For example, the JSP [15] and JSF [16] expressions languages also do not support indexed properties.

### 3.6. Inherited Properties

We have seen that property fields must always have property access methods, and that if the property access methods are not declared in the class, then they will be automatically generated.

But so far we have not discussed how inheritance is handled. Inherited methods can be used as property access methods.

Let's consider the following example in which class Student extends class Person:

```
public class Person {
        private String myName;
        public String getName() {
                return myName;
        }
}
public class Student extends Person {
        @Property private String name;
}
```

In this case, class Student inherits the method getName. Therefore the only missing property access methods for the property field name is the setName method, which will need to be generated.

As expected, the equivalent class for the previously Student class is:

```
public class Student extends Person {
        private String name;
        public void setName(String name) {
                this.name = name;
        }
}
```

### 3.7. Property Visibility

Java properties are part of the public API of a class; hence the property access methods should not have restricted access.

The Property annotation type defines the attribute Visibility, whose valid values are PUBLIC and PACKAGE. This attribute is used to determine the accessibility of the property access methods.
For example, consider the following Student class:

```
public class Student {
        @Property(visibility = Visibility.PACKAGE)
   private String name;
}
```

The equivalent class for the previous example is:

```
public class Student {
        private String name;
        String getName() {
                return name;
        }
        void setName(String value) {
                name = value;
        }
}
```

The default accessibility of a Java property is defined as PUBLIC.

### 3.8 The Property Annotation Type

In the previous sub-sections we have detailed all of the features of Java properties. The annotation type javax.lang.Property is defined here as:

```
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
```

```
public @interface Property {
        enum Visibility {
                PUBLIC,
                PACKAGE,
    }
        enum AnnotationInjection {
                GETTER,
                SETTER,
                BOTH,
                NONE
        }
        Visibility visibility() default Visibility.PUBLIC;
        AnnotationInjection injection() default AnnotationInjection.GETTER;
        String getter() default "";
        String setter() default "";
}
```

### 3.9. Property BeanInfo

The Java property annotation processor factory generates BeanInfo source files for Java classes that declare Java property fields. This generated BeanInfo class has a PropertyInfo for each Java property field. The PropertyInfo includes the name of the property field, the name of the getter method, and the name of the setter method.

Considering the previous Student example, its corresponding StudentBeanInfo is defined as:

```
public class StudentBeanInfo extends SimpleBeanInfo {
        public PropertyDescriptor[] getPropertyDescriptors() {
                try{
                        return new PropertyDescriptor[] {
                                new PropertyDescriptor("name",
```

```
Student.class,"getName", "setName")
                        };
                } catch (IntrospectionException e) {
                        throw new RuntimeException(e.getMessage());
                }
        }
}
```

The Java property annotation processor factory will override any existing BeanInfo source file,

### 3.10. Java Property Documentation

The javadoc tool generates HTML pages describing the public interfaces of Java classes. It divides a Java class documentation page into separate sections; these are the field summary, constructor summary, constant summary, methods summary, nested summary, field details, constructor details, constant details, and method details.

Two new sections are added, the property summary and the property details sections.

The property summary lists the name of all Java property fields of a class, providing they have been with public access methods.

The property details list the name of the Java property, as well their property access methods.

Appendix I provides an example of Java HTML documentation for a class containing Java properties.

## 4. Design and Implementation

### 4.1. Architecture

In the previous sections we described what Java properties are and how they are used in Java programs. In this section we describe how Java properties are implemented and integrated into the Java platform.

From the perspective of the programmer, this project contains the following public artifacts:

- The property annotation type
- The property operator
- The property BeanInfo

In addition to these, we hereby define two additional artifacts:

- Property documentation: HTML web pages describing the public interface of Java classes that contain properties
- Property ant tasks for compilation and javadoc generation: ant tasks to aid with the compilation and javadoc generation of classes that contain Java properties

In total there are five public artifacts for the built-in Java property facility presented in this project.

These artifacts are related as illustrated by the following diagram:

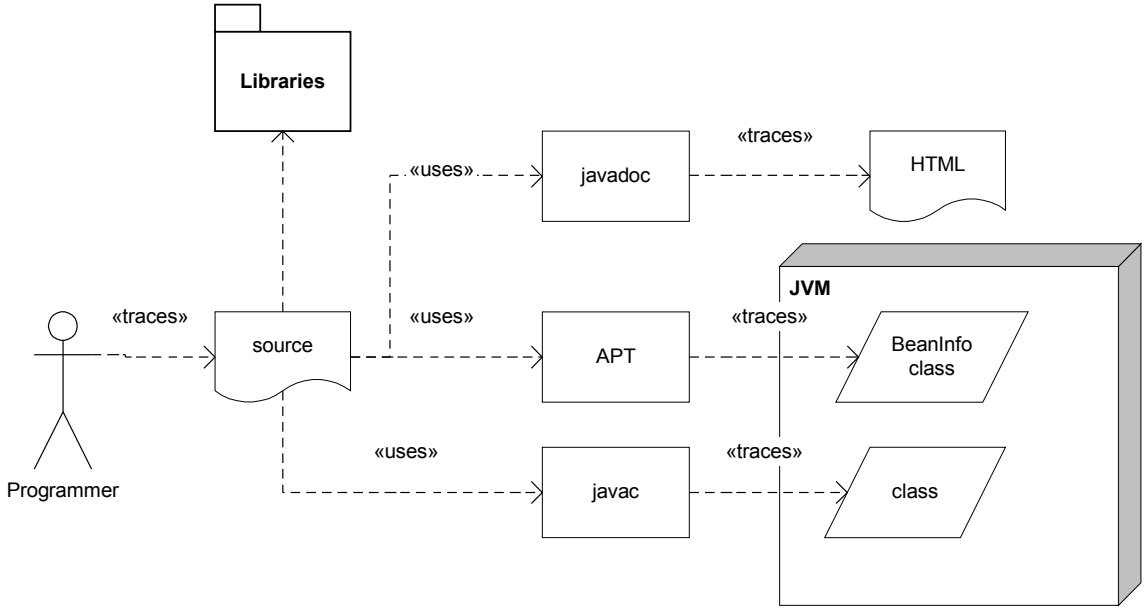


**Figure 2 System Context**

As it can be seen from the illustration, these artifacts can potentially impact the following platform components:

- The Java platform libraries and third-party libraries
- The Java compiler (i.e. javac)

- The Java documentation tool (i.e. javadoc)
- The Java Virtual Machine (i.e. JVM)
- The annotation processing tool (i.e. APT)

Therefore, we need to consider each public artifact in turn and substantiate the changes that it may cause to the Java platform.

This elaboration is done in the following sub-sections, where we will discover that all platform components, with the exception of the JVM itself, need to be modified.

### 4.1.1. The Property Annotation Type

A Java property is declared by means of using the Java annotation type javax.lang.Property. To make it easily accessible, this Java annotation type is defined in the javax.lang package and included in a jar file as a separate third-party library, named propjavac.jar. The javax.lang package is a privileged package reserved for extensions of the Java platform. As Java property is indeed an extension to the Java language, it is felt that it is appropriate to use this package.

One effect of annotating a class field with the property annotation type is that property access methods must be created. The approach taken to create property access methods is to synthesize these methods as part of the compilation process. Obviously this results in modifications to the Java compiler.

### 4.1.2. The Property Operator

The property operator ".@" is a new language construct, and as such impacts the java compiler. In terms of syntax, the java compiler needs to be modified to understand this new operator. In terms of semantic, the property operator is similar to a regular Java method call with the additional intelligence of being able to deduce which method to call based upon some metadata, that is, the metadata defined by the Java property annotation. Modifications to the Java compiler for both syntactic and semantic requirements are described in section 4.2.

### 4.1.3. The Property BeanInfo

Optionally the programmer may decide to generate a BeanInfo class, which describes classes containing Java properties. This is done by executing APT with the Java property specific annotation factory.

The Java property annotation factory is packaged in the Java property library propjavac.jar.

The annotation factory name is aalves.processor.PropertyProcessorFactory.

APT has two options for resolving the annotation processor factories. The first option is to include the factory class in the APT class path, and then to specify the factory class name by using the "-factory" command line option. The following Unix shell command illustrate this approach:

```
apt –classpath propjavac.jar \
–factory aalves.processor.PropertyProcessorFactory  Student.java
```

The second option is to include a registry file in the META-INF/services directory of a jar files present in the APT class path. I take this second approach with the propjavac.jar library, which makes it simpler to be used. In this case, the following Unix shell command can be used:

```
apt –classpath propjavac.jar Student.java
```

### 4.1.4. Property documentation

The javadoc tool generates documentation in the form of HTML pages for the public declarations of Java classes. Since Java properties are potentially a public aspect of Java classes, they should also be considered by the javadoc tool.

Therefore, the javadoc needs to be extended to support Java properties. The extension is done through means of providing a new doclet implementation, which is mirrored in the standard HTML doclet implementation provided by default by the platform.

### 4.1.5. Property Ant tasks

We have seen that to support Java properties, modifications and extensions are needed to both javac and javadoc tools. The details of these modifications and extensions are elaborated on subsequent sections.

Ant tasks are provided to allow easy usage of these tools. These Ant tasks are:

- aalves.tools.PropJavacTask
- aalves.tools.PropJavadocTask

## 4.2. The Java Compiler Tool

The java compiler tool, henceforth referenced as javac, is responsible for converting Java source files into Java class (i.e. bytecode) files. Javac has been completely re-written by Sun for the Java Standard Edition 5.0, mostly to be able to support the new 5.0 language features, such as for-each, Java annotations, and generics. Javac is currently written in Java.

The javac compiler is an intrinsically complicated system. It contains over 100 Java source files. Moreover several of these files are over 2500 lines of code each, and highly cohesive with each other.

The javac compiler has separate distinct phases or stages, ranging from the scanning of the source files to the generation of the output files. The separate stages of javac and the intermediate shared structures are illustrated in Figure 3:

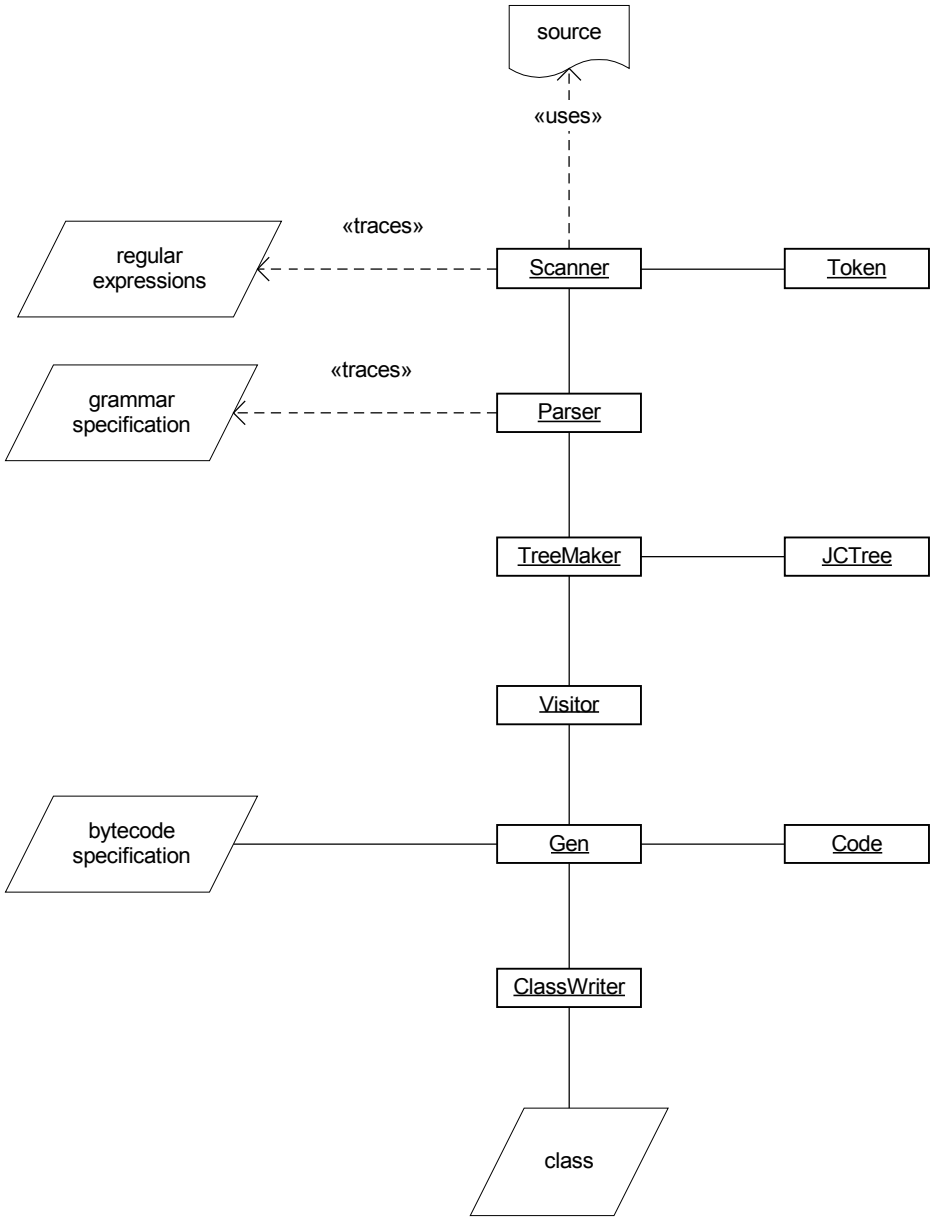


**Figure 3 javac Collaboration diagram**

We detail each one of these phases in the next sub-sections, focusing in the areas where changes and extensions are needed to support Java properties.

### 4.2.1. Scanner

Scanners are responsible for reading the characters of source files and recognizing patterns or tokens out of this stream of characters. The tokens represent the lexical

breakdown of the program, and include entities such as comments, numbers, operators, identifiers, etc.

Particularly, the javac Scanner defines the following set of tokens:

- A token for each language keyword, e.g. if, then, while;
- A token for each language operator and delimitator, e.g. '=', '<', '>', ';', '.';
- A token for each literal type, e.g. string literal, integer literal, long literal;
- A identifier token;
- A error token;

We will add a new token to this list, the PROPACCESS token, whose lexeme is specified as the character sequence ".@".

```
public enum Token {
        ...
        PROPACCESS(".@"),
        ...
}
```

This token is used to specify the Java property operator, as in the following code fragment:

```
student.@name = "Alex";
```

The process of recognizing tokens from streams of characters can be described, or rather encoded in a deterministic state machine (DSM). Hence scanners are generally implementations of DSMs. Typically, a DSM can be coded in two forms, (1) using a table-driven approach, or (2) by using a hand-crafted approach. In the former approach, the scanner code is rather small and basically constitute of a loop that checks on a table for what actions to take, or rather how to change its state. The table contains the machine's productions. This is the approach generally taken by scanner generation tools, such as lex [14]. For the latter approach, the programmer codes the productions directly

in the code, in the form of 'if-then-else' conditions. As one can imagine for a large DSM, a hand-crafted scanner becomes quite large and hard to maintain.

The javac scanner is a hand-crafted scanner. The entry point to this class is the method void Scanner::nextToken(). nextToken() reads the next token from the source file and stores it in a field, accessible by the caller. nextToken() is implemented as a single while loop that runs until the end of the file is found. Within the while loop, there is a switch statement with case conditions for each admissible character.

To recognize the Java property operator token, we have to add a new condition within the nextToken(), tied to the '.' case condition, as illustrated in the following code fragment:

```
void nextToken() {
        ...
        case '.':
                scanChar();
                if ('0' <= ch && ch <= '9') {
                        ...
                } else if (ch == '.') {
                        ...
                } else if ('@' == ch) {
                        scanChar();
                        token = PROPACCESS;
                } else {
                        ...
                }
                return;
        ...
}
```

The probable reason for the scanner in javac to be hand-crafted as opposed to table-driven is performance. Hand-crafted scanners generally execute faster, as they have to do

less memory access, otherwise caused by referencing the table state. The fact that hand-crafted scanners are harder to maintain was probably not a major issue, as the scanner is not a public interface.

### 4.2.2 Parser

Typically, parsers take streams of tokens provided by scanners, verify if these tokens can form a syntactically valid language construct as specified by a language grammar, and outputs a tree structure representing the abstract syntax of the parsed input.

The language grammar can be specified in the Backus-Naur Form (BNF), which is essentially a set of production rules mapping variables, or non-terminals, into another set of variables and terminals. Terminals are essentially tokens. A start production rule is also provided.

Programming languages are classified by their complexity and computational power. One of the simplest and least powerful languages is the regular languages. Scanners recognize regular languages. Java, in the other hand, is quite expressive and computational powerful, Java is a context-free language, and its grammar can be expressed using a context-free grammar. The context-free grammar of the Java language is reasonably large; it contains over 250 productions.

Consider the grammar specification for the non-terminal Selector:

```
Selector      := "." [TypeArguments] Ident [Arguments]
                 | "." THIS
                 | "." [TypeArguments] SUPER SuperSuffix
                 | "." NEW [TypeArguments] InnerCreator
                 | "[" Expression "]"
```

The Selector non-terminal represents the selection of a member (e.g. field, method) of an object in an expression. For example, the underlined section of the following code fragment is an example of a selection:

```
String name = student.getName();
```

To support the Java property operator, a new grammar production rule is added to the non-terminal Selector:

```
Selector      := ".@" Ident
                 | "." [TypeArguments] Ident [Arguments]
                 | "." THIS
                 | "." [TypeArguments] SUPER SuperSuffix
                 | "." NEW [TypeArguments] InnerCreator
                 | "[" Expression "]"
```

Another approach would have been to create a new PropertySelector non-terminal with this production rule, instead of adding it to the existing Selector non-terminal. However, it was felt that this approach was more complicated, and it did not particularly aide with the understanding and implementation of the project.

Parsers are implemented with the goal of recognizing their grammar productions and generating an abstract syntax tree in the process (AST). An AST is also known as a parse tree. There are two well-known approaches for doing so, a bottom-up approach, and a top-bottom approach. The bottom-up approach starts with the leaves and grows it up, that is, adding nodes until getting into the root. It is used for a sub-kind of context-free grammars called LR (i.e. parse from left to right using the rightmost derivation) grammars. The top-down approach starts with the root of the tree and grows it down, that is, adding leaves. Top-down parsing is used for a sub-kind of context-free grammars called LL (i.e. parse from left to right using the rightmost derivation) grammars.
The bottom-up approach is usually implemented by using a table to drive the parser. Several parser generation tools, e.g. yacc, use this approach [14]. These tools generate the

parser code, and a table describing its actions based upon the grammar specified by the programmer.

The top-down approach generally uses a technique called recursive descendent parsing. A recursive descendent parser has a recursive method representing each non-terminal of the grammar. These methods recognize the terminal tokens from the input and invoke other methods representing the non-terminals present in the right-hand-side of its production rules. Recursive descendent parsers are typically hand-crafted.

The javac parser is a hand-crafted recursive descendent parser. Again, similarly to a hand-crafted scanner, a hand-crafted parser is harder to maintain, as the implementation is not generated automatically from its specification. To add the new production rule for the Java property operators, I had to find the appropriate method representing the non-terminal Selector, which had been merged into its parent non-terminal method for Expression3:

```
Expression3 := PrefixOp Expression3
             | (Expression | Type) Expression3
             Primary Selector+ PostfixOp+
```

It would have been preferable if we could have just changed a BNF grammar specification, instead of probing a parser class file with over 2700 lines of code. Nonetheless, the hand-crafted parser should yield a better performance over a table-driven parser.

The Java language is formally specified using a LL(k) grammar, not a LL(1), that is, there are cases where more than one look-ahead is needed. Thus the probable reason why some productions rules were merged together in the recursive descent parser implementation in javac, as noted previously for the non-terminal Selector.

### 4.2.3 Intermediate Representation

As previously mentioned, the output of parsing a recognizable input is a parse tree. A parse tree is an intermediate representation of the program, that is, it is the half-way

representation between the input string being parsed and the output bytecodes to be generated. Most back-end modules of a parser works off from the intermediate representation, transforming the intermediate representation in a more suitable structure that can be used to generate optimized code.

The intermediate representation in javac is based upon a tree hierarchy, where each node represents a syntactic construction of the language. This is described in the class diagram of figure 4:

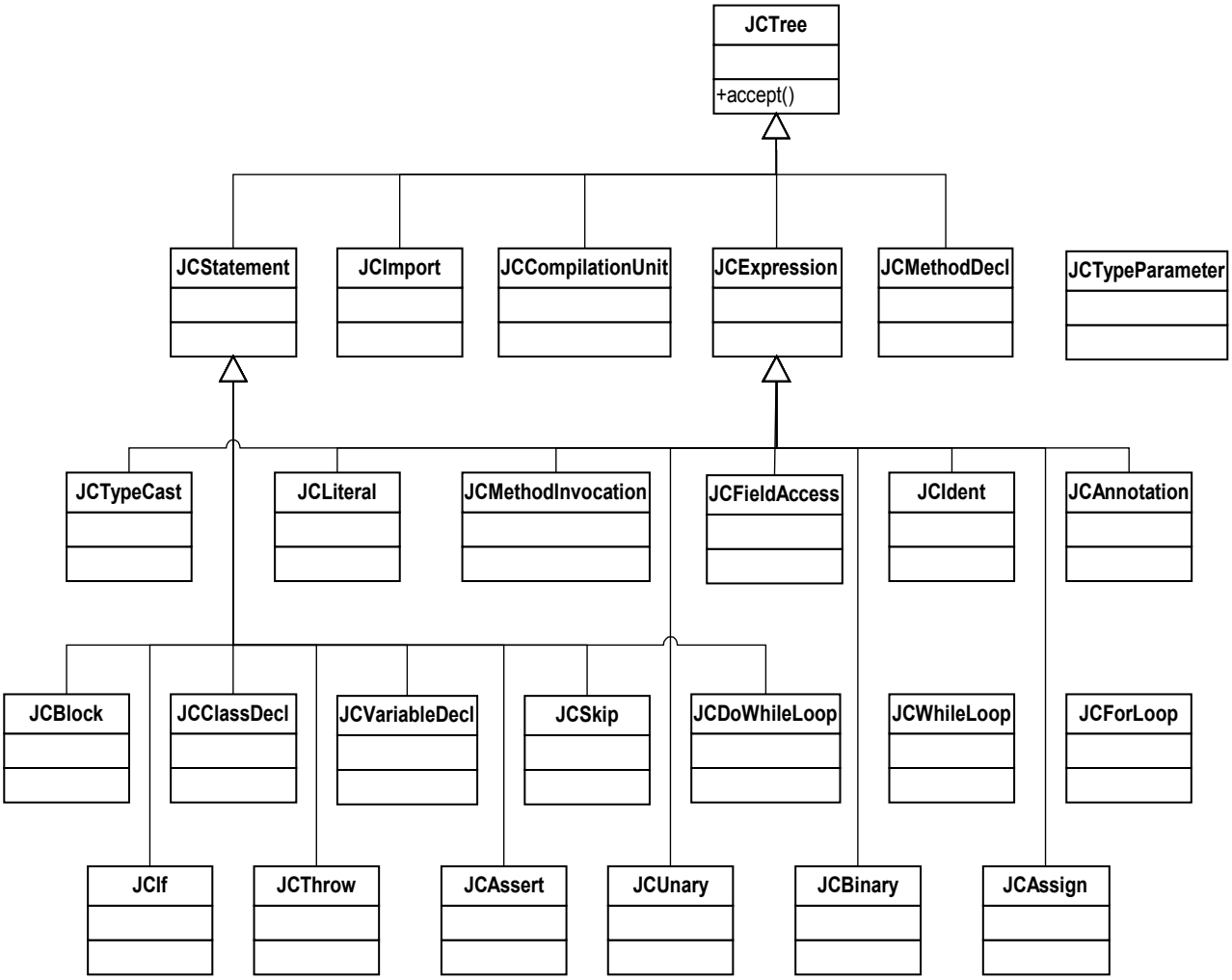


**Figure 4 Parse Tree Class Diagram**

The root node is named JCTree. Although several of these nodes are interesting to us in the subsequent sections, particularly for now we are concerned with the JCFieldAccess JCTree nodes.

In the previous section, recall that we added a new production rule to the non-terminal Selector in the parser. The result of recognizing, that is, executing a production rule is to insert a JCTree node in the parse tree that is being built. The non-terminal Selector production rule inserts a JCFieldAccess JCTree node into the tree. For the new production rule I added to recognize the Java property operator. I also defined a new JCTree node, the JCPropertyFieldAccess JCTree node, which extends the JCFieldAccess JCTree node, and is likewise inserted into the AST tree as the property operator production rule is performed.

JCTree nodes are created by a factory class named TreeMaker. I added a new factory method for the creation of the JCPropertyFieldAccess JCTree node.

```
JCPropertyAccess PropertySelect(JCExpression selected, Name selector)
```

It takes as argument the parsed expression, the name of the selected member, and the position in the source file where the expression is located. The position is used to report any errors, syntactic or otherwise type errors. The name, which is an identifier, is used to link to the Java property being referenced. This is explained in the next section.

### 4.2.4. Visitors

The parser generates a parse tree that is syntactically correct, but has not gone through any semantic validation. The role of the visitor objects is to perform the semantic validation.

The Java compiler defines a Visitor class. The role of the Visitor class is to traverse the nodes of the parse tree, performing semantic validations and transforming the tree into a more suitable form in preparation for the generation of the bytecodes.

The nodes of the JCTree define a method void accept(Visitor v). While traversing the JCTree, each of its node has the accept method called in turn, taking the visitor being

used at the time. The accept method implementation calls back into the visitor, using a method in the Visitor that resembles its type, e.g. void Visitor::visitSelect( JCFieldAccess that). This pattern, even though it hard-codes the type of nodes of a tree, allows one to plug-in different visitors. This is the right trade-off, as generally a tree does not change its structures often, but rather it is more common for new features in the form of visitors to be added. This is similar to the Visitor design pattern [8].

Different visitor implementations are plugged into the javac. The visitors are then executed in a pipelined fashion. Each visitor transforms the AST into a new AST and feeds its output as the input of the next visitor. The last visitor is responsible for the actual bytecode generation.
Some of the javac visitors are:

- Attr: does name resolution, type checking and inference, and constant folding.
- Pretty: prints out the tree as an indented Java source.
- Lower: removes syntactic sugar such as inner classes, for-each loops, and assertion.
- Flow: does live-ness analysis, exception analysis, definite assignment analysis, etc.
- Enter: registers all encountered definitions into the symbol table

#### 4.2.4.1. Property Visitor

I introduced a new visitor, the Property Visitor.
The Property Visitor has two main roles:

1. Checks if a class has fields annotated with the property annotation type
2. Checks for expressions that make use of the Java property operator

The property visitor traverses the parse tree, checking for the existence of Java properties. This is done by verifying if any class being parsed has any of its member fields annotated by the property annotation type. If a Java property is found, then the property visitor proceeds to check if the following condition is met:

- The class declaring, that is, owning the Java property either declares or inherits a corresponding getter method and setter method that respectively matches the signature of the property getter and setter access methods as specified in section 3 of this document

If this condition is not met, then the property visitor must synthesize the missing method declarations in this class that is being evaluated.

It is not trivial to synthesize code during the parsing of a compilation unit. In no other place within the javac is this carried out. Synthesize of the methods is a three-step process described next.

## I.I. Parse Tree Creation

First of all, we must build a parse sub-tree that represents the method declaration and method implementation of the getter and setter being synthesized. This is done by building a JCMethodDecl node that contains the following nodes:

- The method name in the form of a JCIdentifier node; the method name must match the name specified in the getter or setter attribute of the property annotation type, or follow the JavaBeans API naming conventions if these attributes are empty;
- The method modifier in the form of a integer flag; the method modifier must match the visibility attribute of the property annotation type;
- The return type in the form of a JCExpression node;
- The type parameter list in the form of an array of JCTypeParameter nodes;
- The variable declaration list in the form of an array of JCVariableDecl nodes;
- The throw list in the form of an array of JCExpression nodes;
- The method implementation in the form of a JCBlock node; The block node contains a return statement in the case of the getter method, and a assignment statement in the case of the setter method

## I.II. Java Symbol Creation

Next, we must build Java symbols representing the synthesized methods and associate them to the parse sub-tree built in the previous step.

As explained previously, the parse tree represents the syntactic form of the program. However, to proceed with the code generation, the compiler needs another structure that better resembles the class organization, a structure that is not so tied to the program syntax, but rather that groups together the resolved entities and enriches these entities with semantic value, a process which is commonly known as attribution. In javac, this is the role of the Symbol class. The following diagram illustrates the class hierarchy of the Symbol classes:

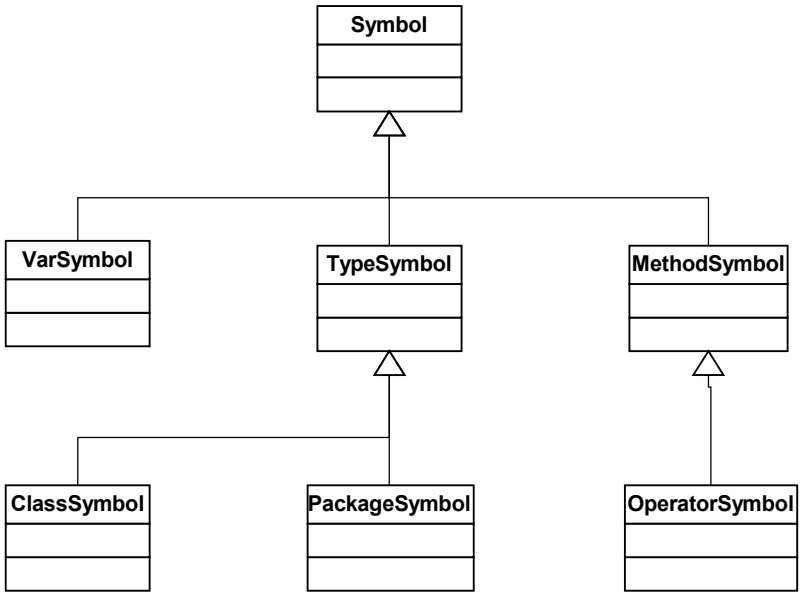


**Figure 5 Symbol Class Diagram**

To build the symbols that represent the synthesized methods, we have to build a MethodSymbol for the method declaration, a VarSymbol for the parameter list, and retrieve the corresponding TypeSymbol for the return type from a symbol table. Furthermore, if the annotationInjection attribute of the Java property annotation type is being used, then the JCTree nodes and symbols representing the annotations to be injected are copied to the synthesized methods.

### I.III. Class Declaration Modification

Finally, the appropriate parse sub-tree nodes and associated symbols representing the synthesized method declarations are added to the JCClassDecl node, which represent the class declaration.

This three-step process fulfills the first role of the property visitor, which is trigged by the presence of a Java property in a class declaration. The second role of the property visitor deals with Java property operators, and is described in the subsequent two-step process:

### II.I. RHS or LHS Property Operator

First, it must be determined if the Java property operator is being used and in which context it is being used, that of a left-hand-side (LHS) expression or that of a right-hand-side (RHS) expression. To do this, the property visitor overrides the methods void visitSelect(JCFieldAccess tree) and void visitAssign(JCAssign tree). If the former is called upon in the context of a variable declaration and the tree parameter is an instance of JCPropertyAccess, then it is RHS expression that uses the Java property operator. If the later is called upon, then check if either LHS or RHS of the assignment expression is an instance of JCPropertyAccess, in which case respectively determining if it is a LHS or RHS expression that uses the Java property operator.

### II.II. Method Application

Having determined that a Java property operator is being used, we have to replace it for a method application. If the operator is in the LHS, then replace it for the method application of the setter property access method, otherwise if the operator is in the RHS, then replace it for the method application of the getter property access method.

This is done by replacing the JCPropertyAccess node by a JCMethodInvocation node, whose selected method name matches the name of the property access method as specified in the property annotation. In the case of LHS property operators, the RHS expression of the assignment is passed along as the parameter of the setter method invocation.

#### 4.2.4.2. Property Visitor Registration

So far all changes had consisted in modifying existing javac source files, i.e. scanner, parser; as they are tightly coupled with the implementation. However, for the property visitor we were able to create a separate Java source file, whose class name is com.sun.tools.javac.comp.Property.

This is possible because the visitor framework is somewhat pluggable. To allow for the property visitor to be invoked during the compilation process, we modified the javac entry-point class, the class com.sun.tools.javac.main.JavaCompiler. The order of invocation of the visitors is important. The property visitor is the second visitor being invoked, just after the Enter visitor, and before all other visitors. The reason is that any synthesized code built by the property visitor must still undergo semantic validation, hence the property vsisitor must execute before the Attrib visitor, the Flow visitor, and any other visitor that deals with semantic validation; however the property visitor uses the symbol table, therefore the property visitor must be executed after the Enter visitor, which is responsible for populating the symbol table with all parsed definitions of the compilation unit.

### 4.2.5. Code Generation

The code generation is performed by referencing a bytecode specification and acting on the symbols built by the visitors. By the time that the code generation is executed, all Java property related artifacts have already been mapped into existing Java entities, hence no additional changes are needed to support Java properties at the code generation level.

## 4.3. The Java Documentation Tool

The javadoc tool was designed so that one can extend it by specifying external components called doclets. The Java platform includes the javadoc toolkit framework and a standard HTML doclet, which is responsible for the HTML formatting generated by default when javadoc is executed.

A doclet is organized around two main concepts, builders and writers. This is illustrated by the following diagram:

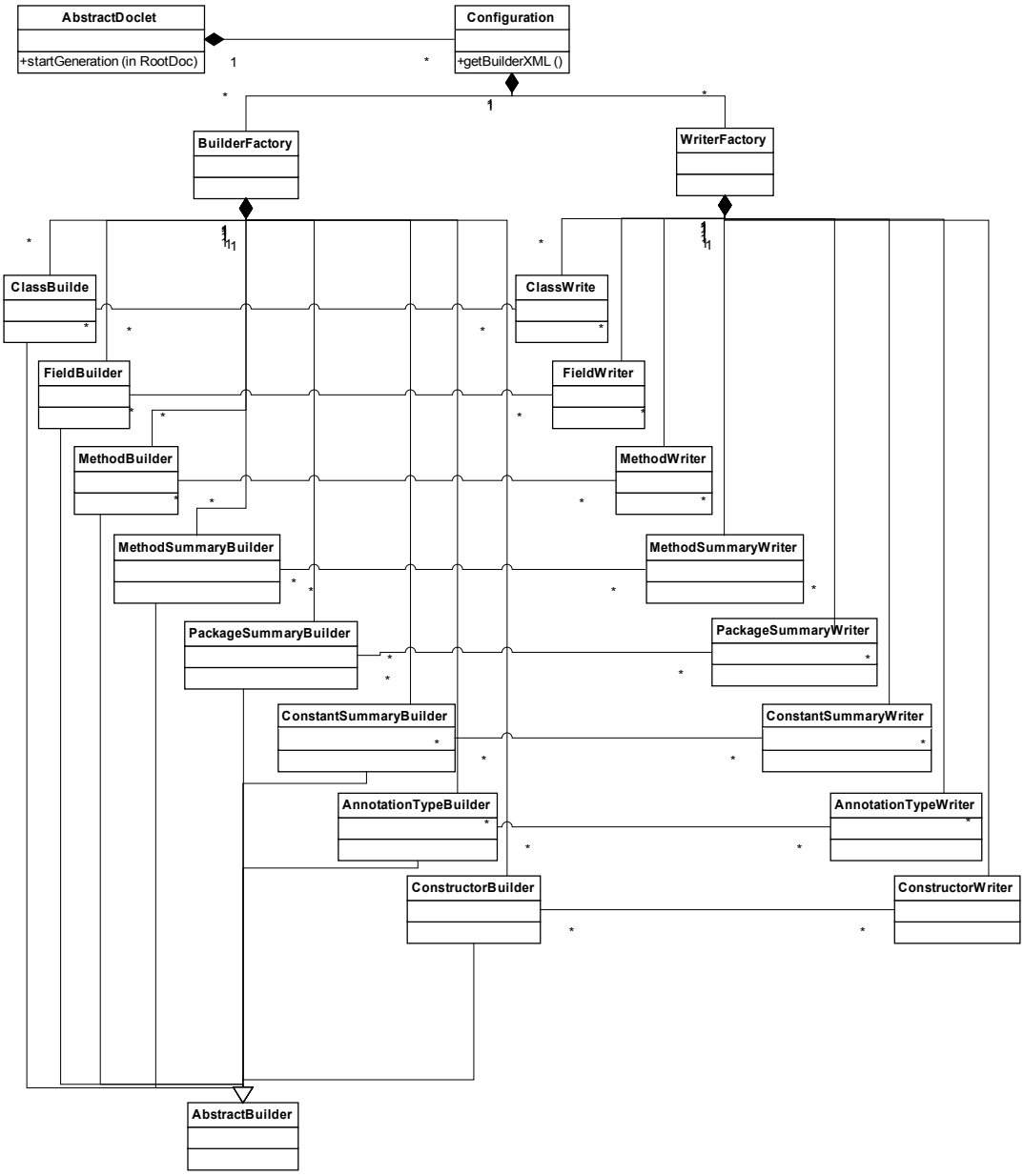


**Figure 6 javadoc Class Diagram**

Doclets are configured by the Configuration class. The Configuration class aggregates the BuilderFactory and the WriterFactory, and includes a builder XML configuration file. The doclet is trigged by a client application, which parses the Java class and provides

a simple, reduced, parse tree to the doclet. The doclet retrieves its builder XML configuration file, and uses it to drive the generation of the documentation output. For each XML element defined in the configuration file, the doclet uses the element name to invoke a build method in a builder class using the Java reflection API. Each build method may recursively invoke other builder classes. Non-leaf elements of the configuration file build by forwarding to other builder classes, which have a build method for each enclosed element of the original non-leaf element.

For example, consider the following fragment of the standard HTML doclet configuration file:

```
<ClassDoc>
        <FieldDetails/>
        <ConstructorDetails/>
        <MethodDetails/>
</ClassDoc>
```

We will assume that the initial builder is the ClassBuilder. The process is started by invoking the method buildFieldDetails on ClassBuilder, which recursively invokes the build method of FieldBuilder. Next, the method buildConstructorDetails on ClassBuilder is invoked, which recursively invokes the ConstructorBuilder. This process continues until all elements have been traversed.

Each builder class is associated with a writer class, which is retrieved from the writer class factory. The builders use the writer classes to generate output specific to some format. As mentioned, the default format is HTML. In this case, all writer classes inherit from an HtmlWriter class, which is an extension of the PrintWriter class.

To implement the property summary and property details sections, we need to modify the standard HTML doclet. However, although the javadoc toolkit is extensible, individual doclets are generally not so extensible. Thus we create a new doclet, the property doclet, which is originally a copy of the standard HTML doclet. Next, we apply the following changes to the property doclet:

- Add the <PropertiesSummary/> and <PropertiesDetails/> elements to the builder XML configuration file
- Create a PropertyWriter, an extension of FieldWriter, that outputs the HTML tags for the property summary and property details sections
- Modify the WriterFactory so it returns the PropertyWriter when a property member is identified
- Class members are deduced from the parse tree by the class VisibleMemberMap. Modify VisibleMemberMap to consider any field annotated by the property annotation type as a Java property, and to consider the visibility of its access methods instead of the visibility of the field itself. For example, the following property is still part of the public interface, even though it is specified as a private member field:

```
@Property (visibility = Property.Visibility.PACKAGE) private String name
```

This is so because the property defines its access methods as being public, and hence being part of the public API.

### 4.4. APT

The PropertyProcessorFactory annotation factory is provided for the generation of BeanInfo classes.

The PropertyProcessorFactory instantiates a property annotation processor for each annotation processing execution. The property annotation processor retrieves all declarations that are annotated with the Java property annotation type using the APT environment. These declarations are field declarations, and include both the field name, and the Java property annotation itself.

The processor checks the Java property annotation attributes to determine the signature of the property getter and setter access methods. If necessary, the processor generates the name of these methods based upon the field name using the standard JavaBeans naming conventions.

The processor uses a template of a BeanInfo source file, and creates a PropertyDescriptor for each Java property, setting the field name, the getter method signature, and the setter method signature. This instantiated template is written to a file using a PrintWriter, which is part of the APT filer environment.

It is important to highlight that APT does not allow the modification of the original source file. For example, if processing a Student Java source file, APT can generate a StudentBeanInfo class file, but APT cannot change the original Student Java source file, or directly modify the Student class file. Hence we are not able to use APT to synthesize the getter and setter property methods.

### 4.5. Java Property Ant Tasks

Two Ant tasks are implemented for this project. The goal of these Ant tasks is to facilitate the usage and acceptance of Java properties.

#### 4.5.1. PropJavacTask

The PropJavacTask Ant task compiles Java source using the extended version of javac that support Java properties. The following Ant build file illustrates its usage:

```
<project name="example.student" basedir=".">
        <taskdef name="propjavac" classname="aalves.tools.PropJavacTask">
          <classpath>
            <pathelement location="${basedir}/../../lib/propjavac.jar"/>
          </classpath>
        </taskdef>

        <propjavac
                srcdir="${basedir}"
                includes="*.java" />
</project>
```

The PropJavacTask Ant task is implemented as an extension of the Javac Ant task. Hence, PropJavacTask supports all of the features of the Javac Ant task, such as the specification of the class path, and the output directory.

PropJavacTask reuses Javac for the compilation of the Java sources; however PropJavacTask changes the Javac boot class path to include the Java property jar file. Specifically, the execute() method of the Javac Ant task is overridden to perform the following actions:

- Verifies that the environment variables JAVA_HOME and PROPJAVA_HOME have been set. If either of them does not exist, the task will raise a BuildException exception.
- Retrieves the location of the library propjavac.jar based upon PROPJAVA_HOME and pre-appends its location to the javac boot class path.
- Retrieves the location of the library tools.jar based upon JAVA_HOME and appends its location to the javac boot class path.
- Sets the fork option of javac to true. This is needed so that the modified boot class path is used during the execution of the compilation.
- Calls the execute method of the javac task to perform the actual compilation.

### 4.5.2. PropJavadocTask

The PropJavadocTask Ant task generates the HTML documentation using the doclet implementation that includes Java properties. PropJavadocTask extends the Javadoc Ant task. Specifically, PropJavadocTask sets Javadoc to use the Java property doclet, and then executes it.

# 5. Conclusion

The initial goal of this project was very clear when the project was started: implement properties as a built-in feature of the Java language and thus improve the productivity of Java programmers. However, how to achieve this was completely unknown. Were changes needed to the JVM? How much of the Java compiler could be re-used? What

programming language would we use to implement this feature, would it be C++ or Java? How do Java properties work with legacy code?

I tried several prototypes. First, I tried to use APT to implement the built-in support for properties, without success. Eventually after an extensive analysis of the internals of the javac compiler, I was able to determine which components to change, and how to change them. Particularly, at the time of writing, this is the first known case in which an annotation is used to modify the compiled output of the annotated source file itself. Generally, annotations are used to generate additional source files.

Before proceeding to the actual development, I fully specified the semantic of Java properties. This helped uncover further research topics.

In the end, I was able to implement Java properties as a native language feature. In addition, I was able to seamlessly integrate Java properties with the platform, making it easy for programmers to use this new language feature.

Java properties remove the need for boiler-plate code, and establish a formal contract between caller and callee. I hope to be able to convince the designers of the the Java language to incorporate Java properties in future versions of the Java platform.

### 5.1. Key Achievements

- Implemented Java properties as a built-in language feature by modifying the javac tool:
    - Modified scanner
    - Modified parser
    - Extended intermediate representation tree
    - Created additional visitor
    - Implemented method synthesization
- Modified other supporting tools, such as javadoc, and provided APT factory.
- Provided flexible implementation that allows one to specify visibility and name of the property access methods, and to inject annotations present in the property field into the access methods.

- Provided non-intrusive implementation that supports legacy code, and is easy to be used. Ant tasks are provided to further lower the entry-barrier for new comers.

### 5.2. Future Work

- Although several test-cases have been tried and automated, the Java language is enormous, and thus further testing is warranted.
- Java IDEs (e.g. Eclipse) need to be modified to support Java properties, especially the Java property operator.
- Work with the Java Committee Process (JCP) to incorporate Java properties into future versions of the Java platform.

## 6. References


[1] C# Language Specification, Version 1.2; http://msdn.microsoft.com/vcsharp/programming/language/default.aspx

[2] The Java Language Specification, Third Edition; http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[3] The Java Virtual Machine Specification, Second Edition; Tim Lindholm, Frank Yellin; http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[4] The Java API Documentation Generator; http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html

[5] JavaBeans Specifications for Java 2; http://java.sun.com/products/javabeans/glasgow/index.html

[6] JSR 175: A Metadata Facility for the Java™ Programming Language; http://www.jcp.org/en/jsr/detail?id=175

[7] Java Annotations for Java 5.0; http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series) (Hardcover).

[9] The Eclipse Project

http://www.eclipse.org/

[10] Spring Application Framework

http://www.springframework.org/

[11] Inversion of Control Containers and the Dependency Injection Pattern

http://www.martinfowler.com/articles/injection.html

[12] Extensible Markup Language (XML)

http://www.w3.org/XML/

[13] The J2EE 1.4 Tutorial

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

[14] The Lex & Yacc Page

http://dinosaur.compilertools.net/

[15] JavaServer Pages Technology

http://java.sun.com/products/jsp/

[16] JavaServer Faces Technology

http://java.sun.com/javaee/javaserverfaces/

# 7. Appendix I – Java documentation Example

Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

### Class Student

```
java.lang.Object
  Student
```

```
public class Student
extends java.lang.Object
```

## Property Summary

| | |
|---|---|
| java.lang.String[] | **classes** |
| java.lang.String | **major** |

| | |
|---|---|
| java.lang.String | **name**<br>Property annotation generates getters and setters |

# Constructor Summary

| | |
|---|---|
| **Student**() | |

# Method Summary

| | |
|---|---|
| static void | **main**(java.lang.String[] args) |

| Methods inherited from class java.lang.Object |
|---|
| clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait |

# Property Detail

## name

java.lang.String **getName()**

void **setName(java.lang.String)**
Property annotation generates getters and setters

---

## major

java.lang.String **getMajor()**

`void` **`setMajor(java.lang.String)`**

### classes

`java.lang.String[]` **`getClasses()`**

`void` **`setClasses(java.lang.String[])`**

# Constructor Detail

### Student

`public` **`Student`**`()`

# Method Detail

### main

`public static void` **`main`**`(java.lang.String[] args)`