How to Design Classes Languages

Version 4.1

August 12, 2008

The languages documented in this manual are provided by DrScheme to be used with the *How to Design Classes* book.

Contents

1 ProfessorJ Beginner

```
Program = Import ... Def ...

Import = import Name ;
         import Name.* ;

Def = class Id {Member Member ...}
      class Id implements Id {Member Member ...}
      interface Id {Signature ...}

Signature = Type Id (Type Id ,...) ;

Member = Field
         Method
         Constructor

Field = Type Id = Expression ;
        Type Id ;

Method = Type Id (Type Id ,...) {Statement}

Constructor = Id (Type Id ,...) {Init ...}

Init = this.Id = Id ;

Statement = if (Expression) {Statement} else {Statement}
            return Expression ;

Expression = Expression Op Expression
             - Expression
             ! Expression
             this
             Expression.Id (Expression ,...)
             Expression.Id
             new Id (Expression ,...)
             check Expression expect Expression
             check Expression expect Expression within Expression
             (Expression)
             Id
             Number
             Character
             String
             true
             false

Name = Id. ... Id
```

```
Op = +
     -
     *
     /
     <
     <=
     ==
     >
     >=
     &&
     ||

Type = Id
       boolean
       int
       char
       double
       float
       long
       byte
       short
```

An `Id` is a sequence of alphanumeric characters, _, and $.

1.1 `import`

- `import Name ;`

  Imports a specified class to use within the program.

- `import Name.* ;`

  Imports a group of classes that can all be used within the program.

1.2 `class`

- `class Id {Member Member ...}`

  Creates a class named Id. One member is required and must be a constructor.

- `class Id implements Id {Member Member ...}`

  Creates a class named Id implements the interface named by (scheme implements). One member must be a constructor. Any method defined by the interface must be a member of this class.

1.3 interface

```
interface Id {Signature ...}
```

Creates an interface named Id that specifies a set of method signatures for classes to implement.

```
Type Id (Type Id ,...) ;
```

The signature specifies a method named Id, expecting the listed arguments. All classes implementing the (scheme interface) must contain a method with the same name, return type, and argument types.

1.4 Field

- `Type Id ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and must be initialized to its value using an Init in the constructor.

- `Type Id = Expression ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and the value of the evaluated `Expression`. `Expression` may not refer to other fields in the current class.

1.5 Method

```
Type Id (Type Id ,...) {Statement}
```

Creates a method, bound to Id, that can be called on the current object, or instances of this class. The body of the method, the statement, will be evaluated when the method is called. The method name may not be the name of any classes defined in the same program or of any fields or methods in the same class.

1.6 Constructor

```
Id (Type Id ,...) {Init ...}
```

Creates a constructor that is used in creating an instance of a class (called an object). The arguments given when creating an instance must be of the same type, and in the same order,

as that specified by the constructor. All of the uninitialized fields of the class must be set in the constructor by the Init sequence.

```
this.Id = Id ;
```

The initialization statements pass the value provided to the constructor to the specified field for later use.

1.7 Statement

- `if (Expression) {Statement} else {Statement}`

  In this statement the expression should have a boolean type. It is evaluated first. If the expression evaluates to `true`, then the first statement (known as the then clause) is evaluated. If the expression evaluates to `false`, the statement following else (the else clause) is evaluated.

- `return Expression ;`

  This form evaluates the expression, and then returns the value of the expression as the result of the method in which it is contained.

1.8 Expression

- `Expression Op Expression`

  Performs the mathematical or logical operation Op on the value of the two expressions.

- `- Expression`

- `! Expression`

  Performs logical negation on the value of the expression.

- `this`

  Allows access to the current object. Within a class, fields and methods of the current class must be accessed through `this`.

- `Expression.Id (Expression ,...)`

  The first expression must evaluate to an object value. Id names a method of this object to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `Expression.Id`

  The first expression must evaluate to an object value. Id names a field of this object, whose value is retrieved by this expression.

- `new Id (Expression ,...)`

  Evaluates to a new instance (object) of the Id class. The class's constructor will be run with the given values (evaluated from left to right) as its arguments. These values must be the correct number and type as specified by the constructor.

- `check Expression expect Expression`

  Compares the resulting values of the two expressions through a deep comparison, including the fields of objects. The resulting value will be a boolean. Neither expression can have type float or double. When test reporting is enabled, results of checks appear in the testing window.

- `check Expression expect Expression within Expression`

  Compares the resulting values of the first two expressions through a deep comparison. The third value must be numeric. If the resulting values of the compared expressions are numeric, their values must be within the third value of each other. For example, in `check a expect b within c`, the absolute value of a-b must be less than or equal to c. If the compared expressions evaluate to objects, any numeric fields will be compared with this formula. The resulting value will be a boolean. When test reporting is enabled, results of checks appear in the testing window.

- `(Expression)`

- `Id`

- `Number`

- `Character`

  Values of type `char` are ASCII characters enclosed by single quotes such as 'a' is the character a. They can be used as numbers as well as characters.

- `String`

  Strings are created through placing text inside of double quotes. For example "I am a string" is a String. A String value is an instance of the class String, which descends from Object, and can also be created with a constructor.

- `true`

- `false`

2 ProfessorJ Intermediate

```
Program = Import ... Def ...

Import = import Name ;
         import Name.* ;

Def = Class
      Interface

Class = class Id {Member ...}
        class Id implements Id ,Id ... {Member ...}
        class Id extends Id {Member ...}
        class Id extends Id implements Id ,Id ... {Member ...}
        abstract class Id {Member ...}
        abstract class Id implements Id ,Id ... {Member ...}
        abstract class Id extends Id {Member ...}
        abstract class Id extends Id implements Id ,Id ... {Member ...}

Interface = interface Id {Signature ...}
            interface Id extends Id ,Id ... {Signature ...}

Signature = MethodReturn Id (Type Id ,...) ;
            abstract MethodReturn Id (Type Id ,...) ;

Member = Field
         Method
         Constructor

Field = Type Id = Expression ;
        Type Id ;

Method = MethodReturn Id (Type Id ,...) {Statement ...}
         abstract MethodReturn Id (Type Id ,...) ;

MethodReturn = void
               Type

Constructor = Id (Type Id ,...) {Statement ...}
```

```
Statement = if (Expression) {Statement ...} else {Statement ...}
            return Expression ;
            return ;
            {Statement ...}
            super (Expression ,...) ;
            Type Id ;
            Type Id = Expression ;
            StatementExpression ;

StatementExpression = Id (Expression ,...)
                      Expression.Id (Expression ,...)
                      super.Id (Expression ,...)

Expression = Expression Op Expression
             - Expression
             ! Expression
             this
             Id. (expression ,...)
             Expression.Id (Expression ,...)
             super.Id (Expression ,...)
             Expression.Id
             new Id (Expression ,...)
             (Type) Expression
             Expression instanceof Type
             check Expression expect Expression
             check Expression expect Expression within Expression
             (Expression)
             Id
             Number
             Character
             String
             null
             true
             false

Name = Id. ... Id
```

```
Op = +
     -
     *
     /
     <
     <=
     ==
     >
     >=
     &&
     ||

Type = Id
       boolean
       int
       char
       double
       float
       long
       byte
       short
```

An `Id` is a sequence of alphanumeric characters, _, and $.

2.1 `import`

- `import Name ;`

  Imports a specified class to use within the program.

- `import Name.* ;`

  Imports a group of classes that can all be used within the program.

2.2 `class`

- `class Id {Member ...}`

  Creates a class named Id. If no constructor is present, one is generated that takes no arguments.

- `class Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that implements the listed interfaces named by (scheme implements). If no constructor is present, one is generated that takes no arguments. Any method defined by the listed interface must be a member of this class.

- `class Id extends Id {Member ...}`

  Creates a class named Id that inherits and expands the behavior of the extended class. If no constructor is present, one is generated that takes no arguments. If the parent class contains a constructor that requires arguments, then none can be generated and the current class must contain a constructor that contains `super`.

- `class Id extends Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that inherits from the extended class and implements the listed interfaces.

- `abstract class Id {Member ...}`

  Creates a class named Id that cannot be instantiated. Members may contain abstract methods. Non-abstract classes extending this class are required to implement all `abstract` methods.

- `abstract class Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id that implements the listed interfaces. Members can include abstract methods. This class need not implement all methods in the interfaces, but all non-abstract subclasses must.

- `abstract class Id extends Id {Member ...}`

  Creates an abstract class named Id that inherits from the extended class. Members can include abstract methods. If the parent is abstract, the current class does not need to implement all inherited abstract methods, but all non-abstract subclasses must.

- `abstract class Id extends Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id, that inherits from the extended class and implements the listed interfaces.

2.3 `interface`

- `interface Id {Signature ...}`

  Creates an interface named Id that specifies a set of method signatures for classes to implement.

- `interface Id extends Id ,Id ... {Signature ...}`

  Creates an interface named Id that specifies a set of method signatures for classes to implement, and inherits the method signatures of the interfaces specified in the extends list.

```
MethodReturn Id (Type Id ,...) ;
```

The signature specifies a method named Id, expecting the listed arguments. All classes implementing the (scheme interface) must contain a method with the same name, return

type, and argument types. A method that does not return a value uses the `void` designation instead of a Type.

```
abstract MethodReturn Id (Type Id ,...) ;
```

A signature may be declared `abstract`. This does not impact the method behavior; all signatures are by default abstract.

2.4 Field

- `Type Id ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and will contain a default value of this type if uninitialized.

- `Type Id = Expression ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and the value of the evaluated `Expression`.

2.5 Method

```
MethodReturn Id (Type Id ,...) {Statement ...}
```

Creates a method, bound to Id, that can be called on the current object, or instances of this class. The body of the method, the statements, will be evaluated sequentially when the method is called. The method name may not be the name of any classes defined in the same program or of any fields or methods in the same class. A method that does not return a value uses the `void` designation instead of a Type for MethodReturn.

```
abstract MethodReturn Id (Type Id ,...) ;
```

Creates a method, bount to Id, inside an abstract class. Like an interface signature, non-abstract classes that inherit this method must provide an implementation.

2.6 Constructor

```
Id (Type Id ,...) {Statement ...}
```

Creates a constructor that is used in creating an instance of a class (called an object). The arguments given when creating an instance must be of the same type, and in the same order, as that specified by the constructor. The statements are executed in sequence in intializing

the object. If the parent of the current class contains a constructor, that expects parameters, then the first statement in the constructor must be a `super` call.

2.7 Statement

- `if (Expression) {Statement ...} else {Statement ...}`

  In this statement the expression should have a boolean type. It is evaluated first. If the expression evaluates to `true`, then the first group of statements (known as the then clause) are evaluated. If the expression evaluates to `false`, the group of statements following else (the else clause) are evaluated.

- `return Expression ;`

  This form evaluates the expression, and then returns the value of the expression as the result of the method in which it is contained.

- `return ;`

  This form causes the method to cease evaluation, without producing a value. Should be used in conjunction with `void` for the MethodReturn.

- `{Statement ...}`

  This statement groups the sequence of statements together, commonly called a block. The statements evaluate sequentially.

- `super (Expression ,...) ;`

  May only appear as the first statement of a constructor. Calls the constructor for the parent class using the given expressions as arguments. Expressions are evaluated left to right.

- `Type Id ;`

  Creates a local variable Id within a method body or a block statement; it is not visible outside the block or method, or to statements the preceed the declaration. The variable must be initialized prior to use.

- `Type Id = Expression ;`

  Creates a local variable Id within a method body or a block statement.

- `StatementExpression ;`

  This set of expressions can be used in a statement position, provided they are followed by ';'.

}

2.8 Expression

- `Expression Op Expression`

  Performs the mathematical or logical operation Op on the value of the two expressions.

- `- Expression`

- `! Expression`

  Performs logical negation on the value of the expression.

- `this`

  Allows access to the current object. Within a class, fields and methods of the current class can be accessed through `this`.

- `Id (Expression ,...)`

  Id names a method of the current class to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `Expression.Id (Expression ,...)`

  The first expression must evaluate to an object value. Id names a method of this object to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `super.Id (Expression ,...)`

  Evaluates the overridden method body using the provided expressions as its arguments.

- `Expression.Id`

  The first expression must evaluate to an object value. Id names a field of this object, whose value is retrieved by this expression.

- `new Id (Expression ,...)`

  Evaluates to a new instance (object) of the Id class. The class's constructor will be run with the given values (evaluated from left to right) as its arguments. These values must be the correct number and type as specified by the constructor.

- `(Type) Expression`

  Evaluates Expression and then confirms that the value matches the specified type. During compilation, the resulting expression has the specified type. If during evaluation, this is not true, an error is raised; otherwise the result of this expression is the result of Expression.

- `Expression instanceof Type`

  Evaluates Expression and then confirms that the value matches the specified type. Returns `true` when the type matches and `false` otherwise.

- `check Expression expect Expression`

  Compares the resulting values of the two expressions through a deep comparison, including the fields of objects. The resulting value will be a boolean. Neither expression can have type float or double. When test reporting is enabled, results of checks appear in the testing window.

- `check Expression expect Expression within Expression`

  Compares the resulting values of the first two expressions through a deep comparison. The third value must be numeric. If the resulting values of the compared expressions are numeric, their values must be within the third value of each other. For example, in `check a expect b within c`, the absolute value of a-b must be less than or equal to c. If the compared expressions evaluate to objects, any numeric fields will be compared with this formula. The resulting value will be a boolean. When test reporting is enabled, results of checks appear in the testing window.

- `(Expression)`

- `Id`

  May refer to either a local variable, method parameter, or field of the current class.

- `Number`

- `Character`

  Values of type `char` are ASCII characters enclosed by single quotes such as 'a' is the character a. They can be used as numbers as well as characters.

- `String`

  Strings are created through placing text inside of double quotes. For example "I am a string" is a String. A String value is an instance of the class String, which descends from Object, and can also be created with a constructor.

- `null`

  A value representing an object with no fields or methods. It should be used as a placeholder for uninitialized fields.

- `true`

- `false`

3 ProfessorJ Intermediate + Access

```
Program = Import ... Def ...

Import = import Name ;
         import Name.* ;

Def = Class
      Interface
      public Class
      public Interface

Class = class Id {Member ...}
        class Id implements Id ,Id ... {Member ...}
        class Id extends Id {Member ...}
        class Id extends Id implements Id ,Id ... {Member ...}
        abstract class Id {Member ...}
        abstract class Id implements Id ,Id ... {Member ...}
        abstract class Id extends Id {Member ...}
        abstract class Id extends Id implements Id ,Id ... {Member ...}

Interface = interface Id {Signature ...}
            interface Id extends Id ,Id ... {Signature ...}

Signature = MethodReturn Id (Type Id ,...) ;
            abstract MethodReturn Id (Type Id ,...) ;

Member = Field
         Modifier Field
         Method
         Modifier Method
         Constructor
         Modifier Constructor

Modifier = public
           private
           protected

Field = Type Id = Expression ;
        Type Id ;

Method = MethodReturn Id (Type Id ,...) {Statement ...}
         abstract MethodReturn Id (Type Id ,...) ;

MethodReturn = void
               Type
```

```
Constructor = Id (Type Id ,...) {Statement ...}

Statement = if (Expression) {Statement ...} else {Statement ...}
            return Expression ;
            return ;
            {Statement ...}
            super (Expression ,...) ;
            this (Expression ,...) ;
            Type Id ;
            Type Id = Expression ;
            StatementExpression ;

StatementExpression = Id (Expression ,...)
                      Expression.Id (Expression ,...)
                      super.Id (Expression ,...)

Expression = Expression Op Expression
             - Expression
             ! Expression
             this
             Id. (expression ,...)
             Expression.Id (Expression ,...)
             super.Id (Expression ,...)
             Expression.Id
             new Id (Expression ,...)
             (Type) Expression
             Expression instanceof Type
             check Expression expect Expression
             check Expression expect Expression within Expression
             (Expression)
             Id
             Number
             Character
             String
             null
             true
             false

Name = Id. ... Id
```

```
Op = +
     -
     *
     /
     <
     <=
     ==
     >
     >=
     &&
     ||

Type = Id
       boolean
       int
       char
       double
       float
       long
       byte
       short
```

An `Id` is a sequence of alphanumeric characters, _, and $.

3.1 `import`

- `import Name ;`

  Imports a specified class to use within the program.

- `import Name.* ;`

  Imports a group of classes that can all be used within the program.

3.2 `class`

- `class Id {Member ...}`

  Creates a class named Id. If no constructor is present, one is generated that takes no arguments.

- `class Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that implements the listed interfaces named by (scheme implements). If no constructor is present, one is generated that takes no arguments. Any method defined by the listed interface must be a member of this class.

- `class Id extends Id {Member ...}`

  Creates a class named Id that inherits and expands the behavior of the extended class. If no constructor is present, one is generated that takes no arguments. If the parent class contains a constructor that requires arguments, then none can be generated and the current class must contain a constructor that contains `super`.

- `class Id extends Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that inherits from the extended class and implements the listed interfaces.

- `abstract class Id {Member ...}`

  Creates a class named Id that cannot be instantiated. Members may contain abstract methods. Non-abstract classes extending this class are required to implement all `abstract` methods.

- `abstract class Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id that implements the listed interfaces. Members can include abstract methods. This class need not implement all methods in the interfaces, but all non-abstract subclasses must.

- `abstract class Id extends Id {Member ...}`

  Creates an abstract class named Id that inherits from the extended class. Members can include abstract methods. If the parent is abstract, the current class does not need to implement all inherited abstract methods, but all non-abstract subclasses must.

- `abstract class Id extends Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id, that inherits from the extended class and implements the listed interfaces.

3.3 `interface`

- `interface Id {Signature ...}`

  Creates an interface named Id that specifies a set of method signatures for classes to implement.

- `interface Id extends Id ,Id ... {Signature ...}`

  Creates an interface named Id that specifies a set of method signatures for classes to implement, and inherits the method signatures of the interfaces specified in the extends list.

```
MethodReturn Id (Type Id ,...) ;
```

The signature specifies a method named Id, expecting the listed arguments. All classes implementing the (scheme interface) must contain a method with the same name, return

type, and argument types. A method that does not return a value uses the `void` designation instead of a Type.

```
abstract MethodReturn Id (Type Id ,...) ;
```

A signature may be declared `abstract`. This does not impact the method behavior; all signatures are by default abstract.

3.4 Modifiers

The modifiers `public`, `private`, and `protected` controll access to the modified member. A public member can be accessed by any class. A private member can only be accessed by the containing class. A protected member can be accessed by the containing class and subclasses.

3.5 Field

- `Type Id ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and will contain a default value of this type if uninitialized.

- `Type Id = Expression ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and the value of the evaluated `Expression`.

3.6 Method

```
MethodReturn Id (Type Id ,...) {Statement ...}
```

Creates a method, bound to Id, that can be called on the current object, or instances of this class. The body of the method, the statements, will be evaluated sequentially when the method is called. The method name may not be the name of any classes defined in the same program or of any fields or methods in the same class. A method that does not return a value uses the `void` designation instead of a Type for MethodReturn.

```
abstract MethodReturn Id (Type Id ,...) ;
```

Creates a method, bount to Id, inside an abstract class. Like an interface signature, non-abstract classes that inherit this method must provide an implementation.

3.7 Constructor

`Id (Type Id ,...) {Statement ...}`

Creates a constructor that is used in creating an instance of a class (called an object). The arguments given when creating an instance must be of the same type, and in the same order, as that specified by the constructor. The statements are executed in sequence in intializing the object. If the parent of the current class contains a constructor, that expects parameters, then the first statement in the constructor must be a `super` call.

Multiple constructors can appear in a class body, provided that for each constructor the type of arguments or the number of arguments us unique. Each constructor may set its own access. A constructor in the same class can be called using a `this` call. This must be the first statement.

3.8 Statement

- `if (Expression) {Statement ...} else {Statement ...}`

  In this statement the expression should have a boolean type. It is evaluated first. If the expression evaluates to `true`, then the first group of statements (known as the then clause) are evaluated. If the expression evaluates to `false`, the group of statements following else (the else clause) are evaluated.

- `return Expression ;`

  This form evaluates the expression, and then returns the value of the expression as the result of the method in which it is contained.

- `return ;`

  This form causes the method to cease evaluation, without producing a value. Should be used in conjunction with `void` for the MethodReturn.

- `{Statement ...}`

  This statement groups the sequence of statements together, commonly called a block. The statements evaluate sequentially.

- `super (Expression ,...) ;`

  May only appear as the first statement of a constructor. Calls the constructor for the parent class using the given expressions as arguments. Expressions are evaluated left to right.

- `this (Expression ,...) ;`

  May only appear as the first statement of a constructor. Calls a different constructor from the same class, chosen by analyzing the given expressions.

- `Type Id ;`

  Creates a local variable Id within a method body or a block statement; it is not visible outside the block or method, or to statements the preceed the declaration. The variable must be initialized prior to use.

- `Type Id = Expression ;`

  Creates a local variable Id within a method body or a block statement.

- `StatementExpression ;`

  This set of expressions can be used in a statement position, provided they are followed by ';'.

3.9 Expression

- `Expression Op Expression`

  Performs the mathematical or logical operation Op on the value of the two expressions.

- `- Expression`

- `! Expression`

  Performs logical negation on the value of the expression.

- `this`

  Allows access to the current object. Within a class, fields and methods of the current class can be accessed through `this`.

- `Id (Expression ,...)`

  Id names a method of the current class to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `Expression.Id (Expression ,...)`

  The first expression must evaluate to an object value. Id names a method of this object to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `super.Id (Expression ,...)`

  Evaluates the overridden method body using the provided expressions as its arguments.

- `Expression.Id`

  The first expression must evaluate to an object value. Id names a field of this object, whose value is retrieved by this expression.

- `new Id (Expression ,...)`

  Evaluates to a new instance (object) of the Id class. The class's constructor will be run with the given values (evaluated from left to right) as its arguments. The number and types of these values select which constructor is used.

- `(Type) Expression`

  Evaluates Expression and then confirms that the value matches the specified type. During compilation, the resulting expression has the specified type. If during evaluation, this is not true, an error is raised; otherwise the result of this expression is the result of Expression.

- `Expression instanceof Type`

  Evaluates Expression and then confirms that the value matches the specified type. Returns `true` when the type matches and `false` otherwise.

- `check Expression expect Expression`

  Compares the resulting values of the two expressions through a deep comparison, including the fields of objects. The resulting value will be a boolean. Neither expression can have type float or double. When test reporting is enabled, results of checks appear in the testing window.

- `check Expression expect Expression within Expression`

  Compares the resulting values of the first two expressions through a deep comparison. The third value must be numeric. If the resulting values of the compared expressions are numeric, their values must be within the third value of each other. For example, in `check a expect b within c`, the absolute value of a-b must be less than or equal to c. If the compared expressions evaluate to objects, any numeric fields will be compared with this formula. The resulting value will be a boolean. When test reporting is enabled, results of checks appear in the testing window.

- `(Expression)`

- `Id`

  May refer to either a local variable, method parameter, or field of the current class.

- `Number`

- `Character`

  Values of type `char` are ASCII characters enclosed by single quotes such as 'a' is the character a. They can be used as numbers as well as characters.

- `String`

  Strings are created through placing text inside of double quotes. For example ”I am a string” is a String. A String value is an instance of the class String, which descends from Object, and can also be created with a constructor.

- `null`

  A value representing an object with no fields or methods. It should be used as a placeholder for uninitialized fields.

- `true`

- `false`

4 ProfessorJ Advanced

```
Program = Import ... Def ...
          PackageDec Import ... Def ...

PackageDec = package Name ;

Import = import Name ;
         import Name.* ;

Def = Class
      Interface
      public Class
      public Interface

Class = class Id {Member ...}
        class Id implements Id ,Id ... {Member ...}
        class Id extends Id {Member ...}
        class Id extends Id implements Id ,Id ... {Member ...}
        abstract class Id {Member ...}
        abstract class Id implements Id ,Id ... {Member ...}
        abstract class Id extends Id {Member ...}
        abstract class Id extends Id implements Id ,Id ... {Member ...}

Interface = interface Id {Signature ...}
            interface Id extends Id ,Id ... {Signature ...}

Signature = MethodReturn Id (Type Id ,...) ;
            abstract MethodReturn Id (Type Id ,...) ;

Member = Field
         Modifier Field
         Method
         Modifier Method
         Constructor
         Modifier Constructor
         {Statement ...}

Modifier = public
           private
           protected
```

```
Field = Type Id = Expression ;
        Type Id = ArrayInit ;
        Type Id ;
        static Type Id = Expression ;
        static Type Id = ArrayInit ;
        static Type Id ;

Method = MethodReturn Id (Type Id ,...) {Statement ...}
         abstract MethodReturn Id (Type Id ,...) ;
         final MethodReturn Id (Type Id ,...) {Statement ...}
         static MethodReturn Id (Type Id ,...) {Statement ...}

MethodReturn = void
               Type

Constructor = Id (Type Id ,...) {Statement ...}

Statement = Expression = Expression ;
            if (Expression) Statement else Statement
            if (Expression) Statement
            return Expression ;
            return ;
            {Statement ...}
            while (Expression) {Statement ...}
            do {Statement ...} while (Expression)
            for (ForInit ForExpression ForUpdate ...) {Statement ...}
            break ;
            continue ;
            super (Expression ,...) ;
            this (Expression ,...) ;
            Type Id ;
            Type Id = Expression ;
            Type Id = ArrayInit ;
            StatementExpression ;

StatementExpression = Id (Expression ,...)
                      Expression.Id (Expression ,...)
                      super.Id (Expression ,...)
                      Expression ++
                      ++ Expression
                      Expression --
                      -- Expression

ForInit = Type Id = Expression ;
          Type Id = ArrayInit ;
          StatementExpression ,StatementExpression ... ;
          ;
```

```
ForExpression = Expression ;
                ;

ForUpdate = StatementExpression
            Expression = Expression

ArrayInit = {ArrayInit ,...}
            {Expression ,...}

Expression = Expression Op Expression
             - Expression
             + Expression
             ! Expression
             ++ Expression
             -- Expression
             Expression --
             Expression ++
             this
             Id. (expression ,...)
             Expression.Id (Expression ,...)
             super.Id (Expression ,...)
             Expression.Id
             Expression [Expression] [Expression] ...
             new Id (Expression ,...)
             new Type [Expression] [Expression] ...
             (Type) Expression
             Expression instanceof Type
             Expression ? Expression : Expression
             check Expression expect Expression
             check Expression expect Expression within Expression
             (Expression)
             Id
             Number
             Character
             String
             null
             true
             false

Name = Id. ... Id
```

```
Op = +
     -
     *
     /
     <
     <=
     ==
     >
     >=
     &&
     ||

Type = Id
       boolean
       int
       char
       double
       float
       long
       byte
       short
       Type []
```

An `Id` is a sequence of alphanumeric characters, _, and $.

4.1 package

```
package Name ;
```

This declaration asserts that all classes contained in the current file are members of the named package.

4.2 import

- `import Name ;`

  Imports a specified class to use within the program.

- `import Name.* ;`

  Imports a group of classes that can all be used within the program.

4.3 `class`

- `class Id {Member ...}`

  Creates a class named Id. If no constructor is present, one is generated that takes no arguments.

- `class Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that implements the listed interfaces named by (scheme implements). If no constructor is present, one is generated that takes no arguments. Any method defined by the listed interface must be a member of this class.

- `class Id extends Id {Member ...}`

  Creates a class named Id that inherits and expands the behavior of the extended class. If no constructor is present, one is generated that takes no arguments. If the parent class contains a constructor that requires arguments, then none can be generated and the current class must contain a constructor that contains `super`.

- `class Id extends Id implements Id ,Id ... {Member ...}`

  Creates a class named Id that inherits from the extended class and implements the listed interfaces.

- `abstract class Id {Member ...}`

  Creates a class named Id that cannot be instantiated. Members may contain abstract methods. Non-abstract classes extending this class are required to implement all `abstract` methods.

- `abstract class Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id that implements the listed interfaces. Members can include abstract methods. This class need not implement all methods in the interfaces, but all non-abstract subclasses must.

- `abstract class Id extends Id {Member ...}`

  Creates an abstract class named Id that inherits from the extended class. Members can include abstract methods. If the parent is abstract, the current class does not need to implement all inherited abstract methods, but all non-abstract subclasses must.

- `abstract class Id extends Id implements Id ,Id ... {Member ...}`

  Creates an abstract class named Id, that inherits from the extended class and implements the listed interfaces.

4.4 `interface`

- `interface Id {Signature ...}`

Creates an interface named Id that specifies a set of method signatures for classes to implement.

- `interface Id extends Id ,Id ... {Signature ...}`

  Creates an interface named Id that specifies a set of method signatures for classes to implement, and inherits the method signatures of the interfaces specified in the extends list.

```
MethodReturn Id (Type Id ,...) ;
```

The signature specifies a method named Id, expecting the listed arguments. All classes implementing the (scheme interface) must contain a method with the same name, return type, and argument types. A method that does not return a value uses the `void` designation instead of a Type.

```
abstract MethodReturn Id (Type Id ,...) ;
```

A signature may be declared `abstract`. This does not impact the method behavior; all signatures are by default abstract.

4.5 Modifiers

The modifiers `public`, `private`, and `protected` controll access to the modified member. A public member can be accessed by any class. A private member can only be accessed by the containing class. A protected member can be accessed by the containing class and subclasses and all classes that are members of the same package. A member without a modifier can be accessed by all members of the same package.

4.6 Field

- `Type Id ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and will contain a default value of this type if uninitialized.

- `Type Id = Expression ;`

  Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field will have the declared type and the value of the evaluated `Expression`.

- `Type Id = ArrayInit ;`

Creates a field, bound to Id, that can be used within the current class, or on instances of the current class using an expression. This field must have an array type and the value is that of the evaluated array initialization specification.

All fields with `static` preceeding their declaration are tied to the class and not tied to an instance of the class. They can be accessed and initialized using the standard techniques for non-static fields. They may also be accessed with the class name preceeding the field name: `Id.Id`. An initializing expression cannot use the `this` expression.

4.7 Method

```
MethodReturn Id (Type Id ,...) {Statement ...}
```

Creates a method, bound to Id, that can be called on the current object, or instances of this class. The body of the method, the statements, will be evaluated sequentially when the method is called. The method name may not be the name of any classes defined in the same program or of any fields or methods in the same class. A method that does not return a value uses the `void` designation instead of a Type for MethodReturn.

```
abstract MethodReturn Id (Type Id ,...) ;
```

Creates a method, bount to Id, inside an abstract class. Like an interface signature, non-abstract classes that inherit this method must provide an implementation.

```
final MethodReturn Id (Type Id ,...) {Statement ...}
```

Creates a method, bound to Id, that cannot be overridden by future classes.

`static MethodReturn Id (Type Id ,...) {Statement ...}` Creates a method, bound to Id, that is tied to the class, not the instance of the class. This method cannot use the `this` expression within the Statement body.

Multiple methods can appear in a class body with the same name, provided that for each method with a given name the type of arguments or the number of arguments is unique.

4.8 Constructor

```
Id (Type Id ,...) {Statement ...}
```

Creates a constructor that is used in creating an instance of a class (called an object). The arguments given when creating an instance must be of the same type, and in the same order, as that specified by the constructor. The statements are executed in sequence in intializing the object. If the parent of the current class contains a constructor, that expects parameters, then the first statement in the constructor must be a `super` call.

Multiple constructors can appear in a class body, provided that for each constructor the type of arguments or the number of arguments is unique. Each constructor may set its own access. A constructor in the same class can be called using a `this` call. This must be the first statement.

4.9 Statement

- `Expression = Expression ;`

  The first expression must be a field reference, array position reference, or a variable. The value of this variable, field, or array position will be changed to be the value of the evalauated expression on the right-hand side of =.

- `if (Expression) Statement else Statement`

  In this statement the expression should have a boolean type. It is evaluated first. If the expression evaluates to `true`, then the first statement (known as the then clause) is evaluated. If the expression evaluates to `false`, the statement following else (the else clause) is evaluated. Both statements may be blocks, including { }.

- `if (Expression) Statement`

  In this statement the expression should have a boolean type. It is evaluated first. If the expression evaluates to `true`, then the statement is evaluated; otherwise the statement has completed evaluation.

- `return Expression ;`

  This form evaluates the expression, and then returns the value of the expression as the result of the method in which it is contained.

- `return ;`

  This form causes the method to cease evaluation, without producing a value. Should be used in conjunction with `void` for the MethodReturn.

- `{Statement ...}`

  This statement groups the sequence of statements together, commonly called a block. The statements evaluate sequentially.

- `while (Expression) {Statement ...}`

  Evaluates the expression, which must have type boolean. If the resulting value is true, then the statements are evaluated. After evaluating the statements, the expression is evaluated again. This repeats until the resulting value is false; once false, the statements are not evaluated and the while statement has completed evaluation.

- `do {Statement ...} while (expression)`

  The do statement evaluates the Statements, and then evaluates the expression, which must have type boolean. If the expression is true the statements evaluate again. This

repeats until the expression is false, afterwhich the do statement has completed evaluation.

- `for (ForInit ForExpression ForUpdate ,...) {Statement ...}`

  The ForInit first initializes a variable, or evaluates a set of expressions. The variable can only be seen within the ForExpression, ForUpdate, and within the Statements. Then the ForExpression is evaluated, when true the Statements are evaluated. Subsequently, the ForUpdate evaluates a set of statement expressions or assignments before evaluating the expression again. Other than the ForInit stage, this repeats until the expression evaluates to false, afterwhich the for statement has completed evaluation.

- `break ;`

  Can only appear within the statement group of a while loop, do loop, or for loop. Causes the loop evaluation to complete without checking the expression again.

- `continue ;`

  Can only appear within the statement group of a while loop, do loop, or for loop. Causes the statement group evaluation to complete, repeating to the evaluation of the conditional.

- `super (Expression ,...) ;`

  May only appear as the first statement of a constructor. Calls the constructor for the parent class using the given expressions as arguments. Expressions are evaluated left to right.

- `this (Expression ,...) ;`

  May only appear as the first statement of a constructor. Calls a different constructor from the same class, chosen by analyzing the given expressions.

- `Type Id ;`

  Creates a local variable Id within a method body or a block statement; it is not visible outside the block or method, or to statements the preceed the declaration. The variable must be initialized prior to use.

- `Type Id = Expression ;`

  Creates a local variable Id within a method body or a block statement.

- `StatementExpression ;`

  This set of expressions can be used in a statement position, provided they are followed by ';'.

4.10 ArrayInit

This syntax specifies an array to be created holding initial values as specified.

`{Expression ,...}`

This form creates a one dimensional array, where the values are the result of evaluating each expression, left to right.

`{ArrayInit ,...}`

This form creates a multi-dimensional array, where the values for this array are arrays, with their values specified by the ArrayInit.

4.11 Expression

- `Expression Op Expression`

  Performs the mathematical or logical operation Op on the value of the two expressions.

- `- Expression`

- `! Expression`

  Performs logical negation on the value of the expression.

- `++ Expression`

  The expression must be a field access, variable, or array position access, where the type must be a number. This form causes 1 to be added to the value of the field, variable, or array position. Returns the augmented number.

- `Expression ++`

  Like `++ Expression`, except the returned value is the initial number held by Expression.

- `-- Expression`

  The expression must be a field access, variable, or array position access, where the type must be a number. This form causes 1 to be subtracted from the value of the field, variable, or array position. Returns the decremented number.

- `Expression --`

  Like `-- Expression`, except the returned value is the initial number held by Expression.

- `this`

  Allows access to the current object. Within a class, fields and methods of the current class can be accessed through `this`.

- `Id (Expression ,...)`

  Id names a method of the current class to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method

as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `Expression.Id (Expression ,...)`

  The first expression must evaluate to an object value. Id names a method of this object to be called by the current expression. The expressions following Id are evaluated from left to right and passed in to the method as its arguments. The number and types of the arguments must match the method's declaration. These values replace the argument names in the body of the method, and the result of the body is the result of this expression.

- `super.Id (Expression ,...)`

  Evaluates the overridden method body using the provided expressions as its arguments.

- `Expression.Id`

  The first expression must evaluate to an object value. Id names a field of this object, whose value is retrieved by this expression.

- `Expression [Expression]`

  The first expression must evaluate into an array object, and the second expression must evaluate into an integer. Evaluation of the full expression retrieves the value stored in the corresponding position in the array. If the integer value is equal to or greater than the size of the array a runtime error will occur. The indexing of the array begins at 0.

- `new Id (Expression ,...)`

  Evaluates to a new instance (object) of the Id class. The class's constructor will be run with the given values (evaluated from left to right) as its arguments. The number and types of these values select which constructor is used.

- `new Type [Expression] [Expression] ...`

  The expressions must all evaluate to integers. Evaluates to a new array value, where the base array holds values of the specified type, and the size is specified by the integer values.

- `(Type) Expression`

  Evaluates Expression and then confirms that the value matches the specified type. During compilation, the resulting expression has the specified type. If during evaluation, this is not true, an error is raised; otherwise the result of this expression is the result of Expression.

- `Expression instanceof Type`

  Evaluates Expression and then confirms that the value matches the specified type. Returns `true` when the type matches and `false` otherwise.

- `Expression ? Expression : Expression`

  This form is an expression form of `if`. The first expression is evaluated (and must have type boolean), if it is true then the second expression is evaluated and this is the result of the ? expression. If the first expression is false, then the third expression is evaluated and this is the result of the ? expression. The second and third expressions must have types that are assignable to one another.

- `check Expression expect Expression`

  Compares the resulting values of the two expressions through a deep comparison, including the fields of objects. The resulting value will be a boolean. Neither expression can have type float or double. When test reporting is enabled, results of checks appear in the testing window.

- `check Expression expect Expression within Expression`

  Compares the resulting values of the first two expressions through a deep comparison. The third value must be numeric. If the resulting values of the compared expressions are numeric, their values must be within the third value of each other. For example, in `check a expect b within c`, the absolute value of a-b must be less than or equal to c. If the compared expressions evaluate to objects, any numeric fields will be compared with this formula. The resulting value will be a boolean. When test reporting is enabled, results of checks appear in the testing window.

- `(Expression)`

- `Id`

  May refer to either a local variable, method parameter, or field of the current class.

- `Number`

- `Character`

  Values of type `char` are ASCII characters enclosed by single quotes such as 'a' is the character a. They can be used as numbers as well as characters.

- `String`

  Strings are created through placing text inside of double quotes. For example "I am a string" is a String. A String value is an instance of the class String, which descends from Object, and can also be created with a constructor.

- `null`

  A value representing an object with no fields or methods. It should be used as a placeholder for uninitialized fields.

- `true`

- `false`

Index