RICHARD ROBINSON

JAVA LANGUAGE, II

Contents

Introduction

JavaDoc & Generics

Utility and other classes may have internal and/or external documentation. The former appears via standard comments (that is, `//` or `/**/`) and explains the specifics of the class. Consequently, the external documentation known as the *API* explains the class' usage via `/** */` and is formatted in HTML. They are placed before public attributes, constructors, methods, and the classes themselves.

There are also special tags for API comments such as `@param` which documents the method parameters, and `@return` which documents what the method returns. In addition, the `@pre` tag specifies the conditions and `@throws` specifies possible exceptions. The API may be extracted in command line via

```
javadoc -d directoryname ClassName.java
```

Generally, avoiding overloading is best, which may be achieved by using the most general interface possible to declare parameters. Additionally, the generic type `T` allows for any type to be used in a method. To restrict the generic type `T` such that it must implement a specific interface, the syntax

```
access static <T> type methodName
    (Object<? extends InterfaceName<? super T>> rstArg, T nonrstArgs...) { }
```

wherein `?` is a *wildcard*, meaning `? super T` is matched by `T` or `super(T)`, and the code within the outer angle brackets can be considered a restricted generic type itself. Finally, the `@assert` tag ensures that even non-standard conditions are taken care of.

Terminology

Classes

Static Classes

A Java class has attributes, constructors, and methods, and typically instantiated by using the `new` keyword to invoke a constructor. A class wherein its objects are stateless or have the same attribute values is called a *utility class*. These classes need not be instantiated and are prefixed with `static`.

Classes that have no or single-state attributes are typically utility classes, and only uses the arguments passed to it and not on any other parameters. A UML diagram of a utility class has the format:

```
<<utility>>
java::packageName::className
+ publicAtt : type
+ publicMethod1(arg types) : type
```

The general form of the implementation of a utility class in Java is given by the syntax

```
public class UtilName {
    access static final type attribute = value; // attributes
    private UtilName(){} // empty constructor
    access static type methodName(type args) {
        // method code
        return methodName
    }
    access static void methodName{obj args} {
        args.objMethod; // mutator
    }
}
```

wherein `access` is one of `public`, `private`. All non-final attributes should be private. The `final` keyword in itself is optional, and used if the attribute is a constant. In utility classes, the entire class is within the attribute's scope. Such a class may be tested in `main` by testing it for a variety of random arguments.

To avoid confusion between identical parameter and attribute names, the syntax `ClassName.name` to refer to the attribute and `name` for the parameter. *Overloading* of a method occurs if there are multiple methods with the same name but different parameter types. For arguments that do not meet a specific condition, a method may throw a new exception via the syntax

```
access static type methodName(type args) throws IllegalArgumentException {
    if (condition) throw new IllegalArgumentException("Error Message");
```

```
    // method code
    return methodName;
}
```

Non-Static Classes

This chapter focuses on classes entirely with non-static features. In the case of these classes, the client must first create an instance of it; that is, an object with the syntax

```
ClassName name = new ClassName(); // creates new object using default constructor
ClassName newName = new ClassName(args) // new object with arguments
newName.methodName(args); // mutator
```

The UML diagram for a non-static class is as follows:

```
ClassName
+ methodName(optionalArgs) : type
```

Additionally, the syntax itself for the definition of a non-static class with multiple types of constructors is given by

```
public class ClassName extends OptParent implements OptInterface {
    access type field; // attributes

    // constructors
    public ClassName() { this(defaultValue); }
    public ClassName(type field) { this.setField(field) }

    // methods
    public type getField() { return this.field; }
    public void setField(type field) { this.field = field; }

    // obligatory methods
    public String toString() { return "str"; }
    public boolean equals(ClassName obj) { return this.field == obj.field; }
    public int hashCode() { return this.getFieldi() }
}
```

When possible, the attributes should be of access `private` as it prevents possible mistakes, nor should they be initialized. Importantly, the `this.var` keyword always references the attribute variable, not the parameter one, as is similar to the `ClassName.var` syntax for static classes so as to disambiguate between variables of the same name.

Static vs Non-Static

Here is an example of the same class using static and non-static properties:

```
class Math {
    public static int sum = 0;
    private Math(){}
    public static double Avg(int... nums) {
        for (int i : nums) Math.sum += i;
        return Math.sum/nums.length
    }
}


class Main {
    public static void main(String[] args) {
        double hyp;
        hype = ExtendedMath.Avg(1,3,5);
    }
}
```

```
class Math {
    private int[] nums;
    Math(int... nums) {this.nums = nums;}
    public double Avg() {
        int sum = 0;
        for (int i : this.nums) sum += i;
        return sum/this.nums.length
    }
}

class Main {
    public static void main(String[] args) {
        NumSet nums = new NumSet(1,3,5);
        nums.Avg();
    }
}
```