RICHARD ROBINSON

JAVA LANGUAGE, II

Contents

Introduction

Utility 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`. For example, the `Math` utility class methods can simply be used via `Math.methodName`.

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.

Arguments & Parameters

Classes are know to be *pass-by-value*; this means that upon calling a class' method, its arguments are created new from memory of the argument of the caller, instead of simply its memory location. The argument values do not change regardless of the method; however, properties of objects may be mutated.

Parameters are the variable arguments in the method declaration header. These parameters are then initialized to the passed values of the arguments. For example, considering an object `obj`, a method may be of the form:

```
public static void methodName(obj objName) {
    objName.setAtt(2 * objName.getAtt); // mutator: uses the setAtt and getAtt methods of obj
}
```

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. Note however a method of the form

```
static void methodName(type arg) { arg++; }
```

has no effect as it does not return a value (it is void) nor mutates an object. 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;
}
```

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

```
javadox -d directorybame 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.

@Assert