

# Method Properties in Java

Dirk Riehle

SKYVA International

www.skyva.com, www.skyva.de

riehle@acm.org, www.riehle.org

## Abstract

As Java developers, we talk a lot about methods of classes and interfaces. We talk about types of methods like getters and setters, command methods, and factory methods. Next to classifying methods by purpose, we also talk about properties of methods like being a primitive or composed method, a hook or template method, a class or instance method, or a convenience method.

Obviously, we have a large vocabulary for talking about method types and method properties. We use this vocabulary to communicate and document different aspects of a method, for example, what it is good for, who may use it, and how it is implemented. Understanding this vocabulary is a key to fast and effective communication among developers.

This article presents seven key method properties that we use in our daily design and programming work. It illustrates them using a running example and catalogs them for use as part of a shared vocabulary. Some of the method properties have their own naming convention. Mastering this vocabulary helps us better implement our methods, better document our classes and interfaces, and communicate more effectively.

## 1 Method types and method properties

We first need to distinguish between method types and method properties. A method type captures the primary purpose of a method, for example, whether the method changes a field of an object or carries out some helper task. The name of a method type serves as a succinct label on a method that indicates what type of service the method provides to a client.

A method property, in contrast, describes some additional property of a method that adds to its specification. Examples of method properties are whether the method is primitive or composed, whether it is available for overriding through subclasses, or whether it is a mere wrapper around a more complicated method. While a method is (most of the time) of exactly one method type, it can have several method properties.

Method types and properties are orthogonal and can be composed (almost) arbitrarily.

### 1.1 Method types

In [1], we discuss nine key method types. These method types are grouped into three main categories:

- *Query methods.* A query method (a.k.a. accessing method) is a method that returns some information about the object being queried. It does not change the object's state.
- *Mutation methods.* A mutation method is a method that changes the object's state (mutates it). Typically, it does not return a value to the client.

- *Helper methods.* A helper method (a.k.a. utility method) is a method that performs some support task. A helper method does not change the object’s state, but only performs operations on the method arguments.

Each of these categories comprises a set of method types, provided in Table 1. Most methods are of one specific type, even though it is sometimes helpful to mix types.

Query method	Mutation method	Helper method
get method (getter)	set method (setter)	factory method
boolean query method	command method	assertion method
comparison method	initialization method	
conversion method		

Table 1: Common method types.

The exact definition of these method types can be found at [www.riehle.org/java-report](http://www.riehle.org/java-report). Suffice it to say that they help in communicating effectively and efficiently about methods of classes and interfaces. Several companies, including some I have worked for, use these method types as part of their programming guidelines.

Method types can be documented using Javadoc tags. For example, a get method like “String component(int i)” that returns a String at some index i of an object is tagged as “@methodtype get”. Or, an assertion method like “void assertIsValidIndex(int i) throws InvalidIndexException” that throws an exception if an invalid index is passed in is tagged as “@methodtype assertion”.

By consistently tagging methods this way, we document a method using a single keyword that communicates the method’s purpose succinctly to its readers. Documentation is only one part of the game, though: using method type names helps in more effectively communicating in design and programming sessions as well.

## 1.2 Method properties

Browsing Table 1, you may wonder where that well-known pattern “template method” of the Design Patterns book [4] went. After all, Table 1 lists factory method, which is the other method-level pattern of [4]. Here the distinction between method types and method properties comes into play. Factory method is the name of a method type, while template method is the name of a method property.

Documenting methods using only method types leaves out crucial information about the implementation of a method and its role within the class. Method types are outward-looking, which is why they are of most value in interface specifications. Method properties, in contrast, are inward-looking, which is why they are of highest value in the implementation of a class or interface.

In this article, we distinguish four general method properties plus further Java-specific method properties. The general properties are class implementation, inheritance interface, class/instance level distinction and convenience.

- *Class implementation.* The class implementation property defines how the method is implemented. Its values primitive, composed, or regular describe how methods in a class interact and depend on each other.
- *Inheritance interface.* The inheritance interface property defines the role of the method in the inheritance interface of a class. Its values hook, template, or regular describe how subclasses are to use them.
- *Class/instance level distinction.* The class/instance distinction level property of a method defines whether the method is a class-level or an instance-level method. Its values are instance, class-instance, and class.
- *Convenience.* The convenience property defines whether the method implements its service itself or is a wrapper around a (typically) more complicated method. The property may be set or not.

Next to these general method properties, Java provides further property dimensions as keywords on the language level. These properties include accessibility (public, protected, etc. plus associated access rules), evolution (final, deprecated), and others. The Java-specific dimensions are not discussed in this article, because they can be found in any Java programming book and are well-known to most Java developers.

The following sections describe the general method properties and show how they can be used to improve the communication among developers and the documentation of source code.

## 2 Class implementation

As the first type of method property, we discuss is the structure of a method's implementation. Viewed from this perspective, a method may be a primitive, composed, or regular method:

- A *primitive method* is a method that carries out one specific task, usually by directly referring to the fields of the object. It does not rely on any (non-primitive) methods of the class that defines the primitive method.
- A *composed method* is a method that organizes a task into several subtasks that it glues together as a linear succession of method calls. Each subtask is represented by another method, primitive or non-primitive.
- A *regular method* is a method that is not a primitive or composed method. (We list this property value for sake of completeness, but do not discuss it further).

These property values are not the only possible values. There may be more that may just have slipped our attention. However, this does not mean they are not there!

### 2.1 Primitive methods

Recall the Name object example from [1]. A Name object represents a name that consists of several succeeding name components. Name objects are used to give names to objects when storing and later retrieving them. Name objects are part of a naming scheme that lets us manage named objects efficiently. Examples of naming schemes are internet domain names, file names in a file system, and Java class names.

We represent each name component in a Name object as a String object. The concatenation of the name components makes up the name. Examples of names are "\$HOME/bin" with the name components "\$HOME" and "bin" and "java.lang.Object" with the name components "java", "lang", and "Object".

Given a Name object, we want to get and set individual name components. Hence, we want methods like "String component(int i)" for getting the name component at index i and "void component(int i, String cs)" for setting the String object cs as the name component at position i. In addition to doing the basic work, both methods have to check whether the index i is actually a valid index. Also, we want to send out events to notify clients about potential or actual changes to the object.

One possible implementation of Name objects is to store the name components in a Vector. Hence, we design and implement a class VectorName that does exactly this. The set method "void component(int i, String cs)" of class VectorName looks as displayed in Listing 1.

```
/**
 * Set a name component to a new value.
 * @methodtype set
 */
public void component(int index, String cs) throws InvalidIndexException {
    assertIsValidIndex(index); // check whether index is valid
    fComponents.setElementAt(cs, index); // set new value
    notifyListeners(new ComponentChangeEvent(...)); // inform listeners
}
```

Listing 1: The set method "void component(int, String)" of VectorName in a preliminary version.

Now assume that a name object also provides a command method “void components(int i, String[] csa)” that replaces a whole subrange of name components, starting at index i, with the name components in the “String[] csa” array. If you implement “components” as a for loop that calls “component” for each name component in “csa”, you get as many ComponentChangeEvents as there are elements in “csa”. Typically, however, you want only one event, to be signaled once the overall task is done.

This is why you split up the functionality of “component” into two methods: one method that does all the embellishing work like checking the index and sending out events, and one method that carries out the basic task of setting the new value for a specific name component. This latter method is called a *primitive method*.

- A *primitive method* is a method that carries out one specific task, usually by directly referring to the fields of the object. It does not rely on any (non-primitive) methods of the class that defines the primitive method.

Listing 2 shows this primitive method, using the customary “basic” prefix in the method’s name.

```
/**
 * Set a name component to a new value.
 * @methodtype set
 * @methodproperties primitive
 */
protected void basicComponent(int index, String component) {
    fComponents.setElementAt(component, index);
}
```

Listing 2: The primitive method “void basicComponent(int, String)” of VectorName.

You can now implement “void component(int, String)” and “void components(int, String[])” using “void basicComponent(int, String)”. This makes efficient use of the resources: you avoid any redundant and unwanted boundary checks or events. Also, you can now use the primitive methods to implement other methods more cleanly.

Primitive methods are typically “fragile” methods. They expect calling methods to maintain all pre- and postconditions and class invariants. For example, “basicComponent” expects that the index position passed in is valid. It does not check its validity to avoid repeated redundant checking. As a consequence, primitive methods are frequently protected or even private methods, because they are used only from inside the class.

In any non-trivial class implementation, you do not only have one primitive method, but several. Actually, you design the implementation of a class around a core set of primitive methods, each one devoted to handling one specific aspect of the object, typically a single field. For example, the set of primitive methods for handling name components of Name objects comprises four methods, displayed in Listing 3.

```
protected String basicComponent(int index) { ... };
protected void basicComponent(int index, String component) { ... };
protected void basicInsert(int index, String component) { ... };
protected void basicRemove(int index) { ... };
```

Listing 3: Primitive methods of VectorName.

The whole scheme of using primitive methods as a basis for implementing classes is called *design by primitives*. It serves to break up a class implementation into smaller, more manageable pieces. Also, it is a key constituent of designing reusable classes, as we see in the next section on hook and template methods.

## 2.2 Composed methods

Listing 4 displays the final “void component(int, String)” method, using the “basicComponent” primitive method.

```

/**
 * Set a name component to a new value.
 * @methodtype set
 */
public void component(int index, String component) throws InvalidIndexException {
    assertIsValidIndex(index); // check whether index is valid
    basicComponent(index, component); // set new value
    notifyListeners(new ComponentChangeEvent(...)); // inform listeners
}

```

Listing 4: The method “void component(int, String)” in its final version.

As you can see, the method’s body consists of three successive invocations of further methods. After an initial call to “assertIsValidIndex”, the method calls “basicComponent”, followed by “notifyListeners”. The “void component(int, String)” method is called a *composed method*.

- A *composed method* is a method that organizes a task into several subtasks that it glues together as a linear succession of method calls. Each subtask is represented by another method, primitive or non-primitive.

A composed method implements a method as a linear succession of calls to other simpler methods to achieve the method’s purpose. I think it is important to emphasize “linear succession” which rules out loops and other concepts.

The reason for this constraint is to better distinguish primitive methods from composed methods and from regular methods. Kent Beck defines the original meaning of composed method in [3]. I read his definition to mean “any kind of method that is not a primitive method.” With this definition, there is no distinction between composed method and regular method and we should get rid of the name “composed method”.

For this reason, I prefer to define composed method more narrowly, as described above.

## 3 Inheritance interface

Let us now discuss method properties that describe the role of a method in class inheritance. The key to reusing a class through inheritance is to properly define an inheritance interface that subclasses use for their implementation. Here, the following three property values are possible:

- A *hook method* is a method that declares a well-defined task and makes it available for overriding through subclasses.
- A *template method* is a method that defines an algorithmic skeleton for a task by breaking it up into subtasks. Some of the subtasks are deferred to subclasses by means of hook methods.
- A *regular method* is a method that is not a hook or template method. (Again, we list this property value for sake of completeness, but do not discuss it further).

In [6] and [7], you can find an in-depth discussion of how to use these concepts in the design and implementation of classes and interfaces.

### 3.1 Hook methods

VectorName is just one way of implementing Name objects. Next to representing the sequence of name components as a Vector, we can also represent the sequence of name components as a single String object in which the different components are separated by an appropriate delimiter char. We call this class StringName. A StringName object represents the name “java.lang.Object” as the String “java#lang#Object” using ‘#’ as the delimiter char.<sup>1</sup>

---

<sup>1</sup> For storing characters like the delimiter char as regular chars, we mask them using an escape char like ‘\’. So the general string “#5” would be represented as the “\#5” StringName object.

Using `StringName` objects, we can represent a name with less memory than with a `VectorName`. However, a `StringName` needs more time to retrieve a specific name component, because it first has to look up the substring that represents the name component and then has to create a new `String` from it. Thus, not one single implementation is best, and we need both `VectorName` and `StringName` objects, depending on how we use the `Name` objects.

Furthermore, we introduce a `Name` interface to capture the interface common to both `Vector-` and `StringName` objects. Also, we introduce an `AbstractName` class that captures the implementation common to both `Vector-` and `StringName` objects. Figure 1 shows the resulting class design.

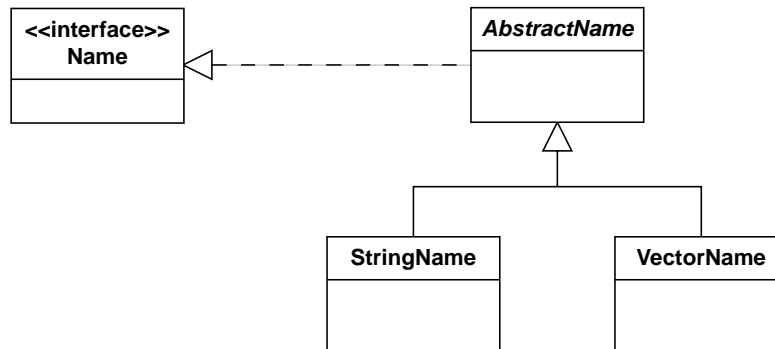


Figure 1: The `Name` example design.

Compare the composed method “`void component(int, String)`” from Listing 4 with the primitive method “`void basicComponent(int, String)`” from Listing 3. The composed method is all about how to accomplish a task by gluing subtasks together. It does not say anything about how the more primitive tasks are carried out, and most importantly, which fields of the object they refer to. The primitive method, in contrast, focuses solely on implementing a primitive task in terms of the object’s fields.

`AbstractName` implements the composed method “`component`” but not the primitive method “`basicComponent`”, because the composed method applies to all subclasses while the primitive method does not. `VectorName` implements the primitive method, because it makes direct use of its fields. However, it is now impossible for the composed method to call the primitive method, because `AbstractName` does not know about it. To remedy this situation, we declare an abstract method “`void basicComponent(int, String)`” in `AbstractName`. The composed method can now use this method without requiring an actual implementation by `AbstractName`. Listing 5 shows this declaration.

```

/**
 * Set a name component to a new value.
 * @methodtype set
 * @methodproperties primitive, hook
 */
protected abstract void basicComponent(int index, String component);
  
```

Listing 5: The abstract method “`void basicComponent(int, String)`” of the `AbstractName` class.

The abstract method “`void basicComponent(int, String)`” is a *hook method*.

- A *hook method* is a method that declares a well-defined task and makes it available for overriding through subclasses.

A hook method is typically an abstract method. Still, it may also provide a default implementation of the method in a few cases to make life easier for subclass implementors.

Now that `AbstractName` requires the implementation of “`basicComponent`” by all its subclasses, we need to define the `StringName` implementation of “`basicComponent`” (next to the `VectorName` implementation of Listing 2). Listing 6 shows an implementation.

```

class StringName extends AbstractName {
    /**
     * Hold the name in a single string.
     */
    public String fName;
    ...

    /**
     * Set a name component to a new value.
     * @methodtype set
     * @methodproperties primitive
     */
    protected void basicComponent(int index, String component) {
        component= maskString(component); // mask control chars
        int startPos= startPosOfComponent(index);
        String head= fName.substring(0, startPos);
        int endPos= endPosOfComponent(index);
        String tail= fName.substring(endPos, fName.length());
        fName= head + component + tail; // compose new name
    }
    ...
}

```

Listing 6: The method “void basicComponent(int, String)” of class StringName.

Most non-trivial classes have several hook methods. The overall set of hook methods represents the *inheritance interface* of a class. Frequently, the hook methods coincide with the primitive methods, so that the set of primitive methods equals the set of hook methods and hence the inheritance interface of a class. However, this is not true in all cases. Listing 7 shows the inheritance interface of AbstractName.

```

protected abstract String basicComponent(int index);
protected abstract void basicComponent(int index, String component);
protected abstract void basicInsert(int index, String component);
protected abstract void basicRemove(int index);
protected abstract Name newName(String name);

```

Listing 7: Inheritance interface of AbstractName composed of hook methods.

Listing 7 adds to Listing 3 the factory method “Name newName(String)” for creating an instance of the same class as the class of the current object (an AbstractName object does not and should not know whether it is actually a StringName or VectorName object).

The application of the design by primitives scheme mentioned above to maximize code reuse in inheritance hierarchies is called the *narrow inheritance interface principle*. Basically, it states that you should base your abstract class implementations on a minimal set of primitive hook methods. These hook methods are then used by composed or template methods.

## 3.2 Template methods

With all primitive and hook methods in place, we can now take a look at how they are used. A first example that uses primitive methods and hook methods is the composed method “void component(int, String)” of Listing 4.

Another example is the “Name contextName()” method defined in the Name interface and implemented by the AbstractName class. The “contextName” method is a get method that returns a new Name object. The Name object represents a name that contains all name components of the old Name object, except for the last name component. So, “contextName” called on “java.lang.Object” returns a “java.lang” Name object. This method is useful for separating the context of a name from its last name component (for example, to separate the package name from a class name or the directory name from a file name).

Listing 7 shows the implementation of “contextName” as defined by AbstractName.

```

/**
 * Return context name of Name object.
 * @methodtype get
 */
public Name contextName() throws InvalidIndexException {
[1]   int nc= noComponents()-1;
      assertIsValidIndex(nc);

[2]   if (nc == 0) {
      return newName("");
    }

    StringBuffer sb= new StringBuffer();
    for (int i= 0; i < nc; i++) {
[3]       sb.append(basicComponent(i));
[4]       sb.append(delimiterChar());
    }
[5]   sb.setLength(sb.length()-1);
      return newName(sb.toString());
    }
}

```

Listing 8: The “Name contextName()” method of AbstractName.

Effectively, Listing 8 shows several calls to primitive methods (references 1, 2, 3, 4, and 5), of which some are also hook methods (references 2, 3, and 5). These calls are embedded into further algorithmic work that glues the results of the individual subtasks together.

The “Name contextName()” method is a template method [4].

- A *template method* is a method that defines an algorithmic skeleton for a task by breaking it into subtasks. The implementation of some subtasks is deferred to subclasses by means of hook methods.

Let us compare regular methods with composed methods and template methods. What are the differences? First of all, a regular method is neither a primitive nor a composed method and neither a hook nor a template method. Rather, it makes use of further methods at will. There are no constraints on how it uses these methods.

In contrast to regular methods, composed methods and template methods are constrained in the way they use other methods. From a composed method, we expect a linear succession of calls to other methods, with only minimal glue in between. There are no constraints on these other methods. Also, no complex algorithm is assumed.

From a template method, we do not only expect an algorithmic skeleton, but also the use of hook methods for subtasks. By delegating work to hook methods, a template method expects a subclass to specialize the behavior of the algorithm to the subclass’ implementation without changing the algorithmic structure. Basically, the subclass fleshes out the skeleton defined by its superclass.

When designing the inheritance interface of an abstract superclass, it is not clear whether hook or template methods come first. Without hook methods, you cannot define template methods. Without template methods, you do not know what the variant parts of your behavior are. So you need both, and defining hook and template methods goes hand in hand.

## 4 Class/instance level distinction

Another important distinction is between class methods and instance methods.

- An *instance method* is a method that applies to an instance of a class.
- A *class-instance method* is a method that applies to a class object.
- A *class method* is a method that applies to a class.

In Java, instance methods and class-instance methods are trivial, but class methods are not.



## 4.1 Instance and class-instance methods

All methods of a class that are not static methods are *instance methods*. An instance method always has an implicit “this” argument that refers to the object the method is executed on. Static methods are excluded from the set of instance methods, because they do not refer to a specific object. Both “basicComponent” and “component” are examples of instance methods.

*Class-instance methods* are a specialized form of instance methods. Class-instance methods are instance methods of class objects. In Java, the set of class-instance methods is defined by the methods of `java.lang.Class`. In fully reflective systems, you can subclass `Class` and provide new implementations of it. Unfortunately, Java is not such a system. This is, why we need a workaround in the form of (static) class methods.

## 4.2 Class methods

Assume that our `Name` objects were *immutable objects*. An immutable object is an object that has no mutation methods so that its state can not be changed. We would replace command methods like “insert” and “remove” with methods that do not change the `Name` object but rather return a new object that represents the result of the “insert” or remove method. Immutable objects are free of the dangers of side-effects, because they cannot be changed. A consequence of this property is that you can safely share immutable objects and reduce memory consumption by avoiding redundant objects.<sup>2</sup> To achieve this, you must gain control of object creation.

You do this by offering your own object creation methods. So you design and implement a static `StringName` method “`public static StringName newInstance(String name)`”, as displayed in Listing 9. This method is used by clients to receive a new `StringName` object. The method first tries to reuse existing `StringName` objects that it has stored in a hashtable. If no matching `StringName` is found, it creates a new one that it returns to the client.

```
public class StringName extends AbstractName {
    /**
     * sStringNames stores all StringNames retrieved using newInstance
     */
    protected static Hashtable sStringNames= new Hashtable();
    ...

    /**
     * Returns a pre-existing StringName, if one exists.
     * Creates a new StringName and stores it, if none exists.
     * @methodtype factory
     * @methodproperties class
     */
    public static StringName newInstance(String name) {
        StringName sn= (StringName) sStringNames.get(name);
        if(sn == null) {
            sn= new StringName(name);
            sStringNames.put(name, sn);
        }
        return sn;
    }
    ...
}
```

Listing 9: The “`StringName newInstance(name)`” class method of `StringName`.

The “`newInstance`” method is a *class method*, and the “`sStringNames`” field is a *class field*.

- A *class method* is a method that applies to a class.
- A *class field* is a field of a class (rather than a field of an instance of it.)

A class method is a method that provides class-level functionality. A class method typically refers to class fields much like an instance method refers to fields of an instance of the class.

---

<sup>2</sup> If you are curious what you can gain from this technique, take a look at [www.jvalue.org](http://www.jvalue.org).

In Java, there is no direct way of expressing class methods. However, we can emulate them using static methods. Emulating class methods through static methods is not perfect; actually, it is pretty cumbersome, because you do not have inheritance and polymorphism. However, it works for many common tasks.

This “newInstance” method is just one instance of the common “use static methods to represent class methods” idiom. Other examples are the “String.valueOf(…)” methods of the JDK String class and the “BigInteger.valueOf(…)” methods of the JDK BigInteger class.

Not all static methods are class methods. Examples of static methods that are not class methods are the “maskString” and “demaskString” methods that mask or demask a String from control characters (not shown in any listing).

What do we gain by making class methods explicit? Well, we cleanly separate concerns. An instance method should not access or manipulate class fields unless this decision is easy to undo during system evolution. Rather, any class field should be encapsulated behind appropriate class methods. This separation of concerns makes our code easier to understand and easier to change as requirements evolve.

## 5 Convenience methods

Finally, let us review convenience methods [5].

- A *convenience method* is a method that simplifies the use of another, more complicated method by providing a simpler signature and by using default arguments where the client supplies no arguments.

Effectively, a convenience method is a wrapper around a more complicated method, here called the wrapped method. Hence, the convenience method is implemented using the wrapped method. However, it makes using the wrapped method easier by requiring fewer arguments from the client and by supplying default arguments for the missing arguments.

Consider Listings 10-12. They show the convenience methods “String asString()” and “String asString(char)” and the more complicated wrapped method “String asString(char, char)”.

```
/**
 * Returns string representation of name.
 * @methodtype conversion
 * @methodproperties convenience
 */
public String asString() {
    return asString(' ');
}
```

Listing 10: The “String asString()” convenience conversion method.

```
/**
 * @methodtype conversion
 * @methodproperties convenience
 */
public String asString(char delimiter) {
    return asString(delimiter, ESCAPE_CHAR);
}
```

Listing 11: The “String asString(char)” convenience conversion method.

```
/**
 * @methodtype conversion
 * @methodproperties primitive
 */
public String asString(char delimiter, char escape) {
    if (isEmpty()) {
        return "";
    }
    StringBuffer sb= new StringBuffer();
    String delimiterString= String.valueOf(delimiter);
```

```

    for (int i= 0; i<noComponents(); i++) {
        sb.append(basicComponent(i));
        sb.append(delimiterString);
    }
    sb.setLength(sb.length()-1);
    return sb.toString();
}

```

Listing 12: The “String asString(char, char)” primitive conversion method.

As we can see, all “asString” methods do the same thing, which is to provide a string representation of a Name object. However, some “asString” methods require fewer arguments than others and hence are more convenient to use. Only one method, the “String asString(char, char)” method, does the actual work. All other methods are implemented by delegating to another method.

The method signature of a convenience method typically defines fewer arguments than the signature of the wrapped method. When calling the wrapped method (which may be just another slightly more complicated convenience method), the convenience method uses default values or derived values as the missing arguments.

In the example of Listing 10-12, “String asString()” uses the space character as the default delimiter argument, and “String asString(char)” uses the default escape char as the escape char. Finally, “String asString(char, char)” uses no default arguments but does the actual work. A client is free to use any of these methods, depending on which arguments it wants to supply.

A variation of this idiom are convenience constructors. Java supports them through the “this(...)” construct [2].

- A *convenience constructor* is a constructor that simplifies the use of another, more complicated constructor by providing a simpler signature and by using default arguments where the client supplies no arguments.

Listing 13 shows a convenience constructor of StringName, and Listing 14 shows the wrapped constructor used by the convenience constructor.

```

/**
 * @methodtype constructor
 * @methodproperties convenience
 */
public StringName(String name) {
    this(name, DELIMITER_CHAR, ESCAPE_CHAR);
}

```

Listing 13: The “StringName(String)” convenience constructor.

```

/**
 * @methodtype constructor
 * @methodproperties composed
 */
public StringName(String name, char delimiter, char escape) {
    super();
    initialize(name, delimiter, escape);
}

```

Listing 14: The “StringName(String, char, char)” constructor.

Typically, convenience methods have the same name as the method they are wrapping. (Convenience constructors obviously must have the same name.) However, this need not be true in all cases.

Also, you can frequently find a cascading implementation of convenience methods. In the first example, “asString” is implemented based on “asString(char)” which in turn is implemented based on “asString(char, char)”. Most developers I know consider cascading convenience methods good style, because changing a default argument can be carried out in exactly one place.

## 6 What have we gained?

We can now tag methods with their types and properties. Methods like “void basicComponent(int, String)” and “Name contextName()” can now be documented as shown in Listing 15.

```
/**
 * Set a name component to a new value.
 * @methodtype set
 * @methodproperties primitive hook
 */
protected abstract void basicComponent(int index, String component);

/**
 * Return context name of Name object.
 * @methodtype get
 * @methodproperties template
 */
public Name contextName() throws InvalidIndexException { ... }
```

Listing 15: Two example methods documented using @methodtype and @methodproperties tags.

When you talk about methods, you chain the name of the type and the names of the properties. For example, you call the “basicComponent” method a “primitive hook set method”, if you want to give a full specification. Type binds stronger to “method” than any of the properties. Among the properties, class/instance binds stronger than hook/template than primitive/composed than convenience. You can also call the “basicComponent” method a “primitive method” or a “hook method” or a “set method” and leave out the other properties, depending on what method aspect you are talking about.

Talking about methods and documenting them this way is easy, lightweight, and precise. Once you agree on a common catalog of method types and properties as presented in [1] and in this article, you can explain what a method does much faster and more successfully than possible with any lengthy explanation.

## 7 Summary

This article provides us with a vocabulary to more effectively talk about methods in Java interfaces and classes. It provides the most common method properties, distinguishes them from method types, and gives them a precise definition. If you feel that it leaves out some key method properties, I’d be happy to hear from you. If you know further aliases for the names of the method properties provided here, let me also know. At [www.riehle.org/java-report](http://www.riehle.org/java-report) you can find a growing catalog of these and other method properties as well as the source code of the examples.

## 8 Acknowledgments

I would like to acknowledge the feedback of Robert Hirschfeld, Wolf Siberski, and Hans Wegener. Their combined comments helped me improve the article in many ways.

The method property names are taken from several different sources:

- Arnold and Gosling’s Java the Language [2],
- Kent Beck’s book Smalltalk Best Practice Patterns [3],
- Gang-of-four’s book Design Patterns [4],
- Robert Hirschfeld’s article on Convenience Methods [5],

and most importantly, from common knowledge and use.

## 9 References

- [1] Dirk Riehle. "Method Types in Java". Java Report 5, 2 (February 2000).
- [2] Ken Arnold and James Gosling. Java, the Programming Language. Addison-Wesley, 1996.
- [3] Kent Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1996.
- [4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, 1995.
- [5] Robert Hirschfeld. "Convenience Method." Pattern Languages of Program Design 4. Addison-Wesley, 2000.
- [6] Dirk Riehle and Erica Dubach. "Working with Java Interfaces and Classes: How to Separate Interfaces from Implementations." Java Report 4, 7 (July 1999). Page 35pp.
- [7] Dirk Riehle and Erica Dubach. "Working with Java Interfaces and Classes: How to Maximize Design and Code Reuse in the Face of Inheritance." Java Report 4, 10 (October 1999). Page 34pp.

## 10 Catalog of method properties

### 10.1 Class implementation

#### 10.1.1 Primitive method

Definition:	<i>A primitive method</i> is a method that carries out one specific task, usually by directly referring to the fields of the object. It does not rely on any (non-primitive) methods of the class that defines the primitive method.
JDK example:	-
Name example:	void AbstractName::assertIsValidIndex(int), String StringName::basicComponent(int).
Prefixes:	basic, do.
Comment:	Design by Primitive is a key principle of good class design that uses primitive methods.

#### 10.1.2 Composed method

Definition:	<i>A composed method</i> is a method that organizes a task into several subtasks that it glues together as a linear succession of method calls. Each subtask is represented by another method, primitive or non-primitive.
JDK example:	-
Name example:	String AbstractName::component(int), void AbstractName::component(int, String).
Prefixes:	-
Comment:	Name taken from [3].

### 10.2 Inheritance interface

#### 10.2.1 Hook method

Definition:	<i>A hook method</i> is a method that declares a well-defined task and makes it available for
-------------	---

	overriding through subclasses.
JDK example:	-
Name example:	void AbstractName::basicComponent(int, String).
Prefixes:	-
Comment:	-

### 10.2.2 Template method

Definition:	A <i>template method</i> is a method that defines an algorithmic skeleton for a task by breaking it into subtasks. Some of the subtasks are deferred to subclasses by means of hook methods.
JDK example:	-
Name example:	Name Name::contextName().
Prefixes:	-
Comment:	Name taken from [4]

## 10.3 Class/instance level distinction

### 10.3.1 Instance method

Definition:	An <i>instance method</i> is a method that applies to an instance of a class.
JDK example:	Every non-static query or mutation method of a class or interface.
Name example:	Every non-static query or mutation method of a class or interface.
Prefixes:	-
Comment:	In reflective systems, an instance method may also be a class method (but need not).

### 10.3.2 Class method

Definition:	A <i>class method</i> is a method that applies to a class.
JDK example:	static String String::valueOf(...), static BigInteger BigInteger::valueOf(long).
Name example:	static StringName StringName::newInstance(String name).
Prefixes:	-
Comment:	In reflective systems, a class method is always an instance method of a class object.

## 10.4 Miscellaneous

### 10.4.1 Convenience method

Definition:	A <i>convenience method</i> is a method that simplifies the use of another, more complicated method by providing a simpler signature and by using default arguments where the client supplies no arguments.
JDK example:	String BigInteger::toString() (wraps String BigInteger::toString(int radix)).

Name example:	String Name::asString(), String Name::asString(char). (But not: asString(char, char).)
Prefixes:	-
Comment:	Name taken from [5].

### 10.4.2 Convenience constructor

Definition:	A <i>convenience constructor</i> is a constructor that simplifies the use of another, more complicated constructor by providing a simpler signature and by using default arguments where the client supplies no arguments.
JDK example:	BigInteger::BigInteger(String) (wraps BigInteger::BigInteger(String value, int radix)).
Name example:	StringName StringName::StringName(String name).
Prefixes:	-
Comment:	-