Method Types in Java

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de

riehle@acm.org, www.riehle.org

Abstract

As Java developers, we talk about query methods, command methods, and factory methods. We talk about convenience methods, helper methods, and assertion methods. We talk about primitive methods, composed methods, and template methods.

Obviously, we have a rich vocabulary for talking about methods of a class or interface. We use this vocabulary to quickly communicate and document what a method does, who it may be used by, and how it is implemented. Understanding this vocabulary is key to fast and effective communication among developers.

This article presents three method type categories and nine key method types that we use in our daily work. It presents them using a running example and then catalogs them for use as part of a shared vocabulary. Each of the method types comes with its own naming convention. Mastering this vocabulary helps us better implement our methods, better document our classes and interfaces, and communicate more effectively.

1 Introduction

We use a common class design pattern to illustrate the different types of methods: the combination of an interface with an abstract superclass and two concrete subclasses that add to the superclass and that implement the interface. The example is a Name interface with a complementing abstract superclass AbstractName and two concrete implementations StringName and VectorName. This example is rich enough to provide all the core types of methods that make up large parts of our everyday vocabulary when we talk about methods.

Walking through this design and discussing the methods lets us encounter three main categories of method types: *query methods*, *mutation methods*, and *helper methods*. We then use specific method types like *conversion method* or *assertion method* to better design the interface and classes as well as to better document them. Using the naming conventions associated with each method type lets us communicate more efficiently.

The method types discussed in this article describe the purpose of a method as it serves a client. A follow-up article discusses the additional properties a method may have. Method properties include being a primitive, hook, or convenience method. Every method has exactly one method type, but it may have several method properties. So, a method may be of type “get method” and have properties “primitive” and “hook”.

Consider the design example now: we frequently give names to objects so that we can store the objects under these names and retrieve them later. Names may be as simple as a single string, and they may be as complex as a multi-part URL. Names we use frequently are class, interface, and package names, for example, “java.lang.String”. Such a name consists of several parts, called name components. The name components of “java.lang.String” are “java”, “lang”, and “String”. Generally speaking, we can view a name as a sequence of name components. In the example, we represent each name as a Name object and each name component as a String object.

Dirk Riehle. “Method Types in Java.” Java Report, February 2000.
Copyright 2000 by Dirk Riehle. All rights reserved.

Name objects do not exist without a purpose: they are always part of a naming scheme. We can manage objects named by a Name object by using an appropriate naming scheme. For example, viewing a name as a sequence of name components lets us manage named objects in a hierarchical fashion, which is the most common (and convenient) management scheme I know of. Examples of Name objects that are interpreted hierarchically are internet domain names, file names, and Java class names.

For example, to look up the class file for "com.mystartup.killerapp.Main", the Java runtime environment first takes the classpath and then searches for a directory called "com", followed by searching for a directory called "mystartup", and so on. This is a recursive descent into a directory hierarchy. Whether we represent the original name as a sequence of components or not, the lookup algorithm requires these name components one after another.

We may choose to interpret a name as a sequence of name components. However, this does not say anything about its implementation. We may implement name objects using a single string or a Vector of strings. In the first case, the whole name is in the string, in the second case, each element of the Vector is one component of the name. Let us call the class of the first case "StringName" and the class of the second case "VectorName" to indicate how they are implemented.

These two different implementations have different behavior. A StringName is optimal (minimal) in memory consumption, but slow if you need to retrieve a specific component, for example the class name on index position 4. A VectorName is (close to) optimal in lookup speed, but requires more memory, because of the additional Vector object. You might argue to introduce yet another third class StringArrayName, where we hold the name components in a String array. That's true, but still StringArrayName has disadvantages over StringName, for example when you have to represent the name object as a single string, as you might have to do when writing out name objects to a stream.

Experience has it that for different use-contexts, you will want different implementations, and so we keep both the StringName and the VectorName class. To make programming convenient, we hide these implementation classes behind a common Name interface that captures all the methods available both for StringName and VectorName objects. We also introduce a shared abstract superclass "AbstractName" from which StringName and VectorName inherit to reuse code [5, 6]. Figure 1 shows the design visually using UML syntax.

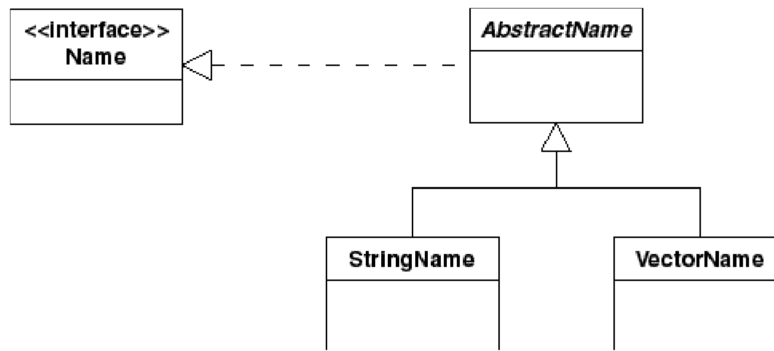

Figure 1: The Name example design.

2 The Name interface

How does the Name interface look like? Listing 1 provides an answer.

```
package org.riehle.Name;

public interface Name {

public static final char DELIMITER_CHAR = '#';
public static final String DELIMITER_STRING = "#";
public static final char ESCAPE_CHAR = '\\';
public static final String ESCAPE_STRING = "\\";

/**
 * Returns a simple human readable description. May not show all
 * fields but is guaranteed to be distinct for distinct values.
 */
public String asString();
public String asStringWith(char delimiter);

/**
 * Gets/sets a name component.
 */
public String component(int index) throws InvalidIndexException;
public void component(int index, String comp) throws InvalidIndexException;

/**
 * Appends/prepends/inserts/removes a name component.
 */
public void prepend(String comp);
public void append(String comp);
public void insert(int index, String comp) throws InvalidIndexException;
public void remove(int index) throws InvalidIndexException;

/**
 * Returns delimiter/escape char/string used by this name object.
 */
public char delimiterChar();
public String delimiterString();
public char escapeChar();
public String escapeString();

/**
 * Returns true if objects are equal.
 * Objects may be of different classes.
 */
public boolean isEqual(Name name);

/**
 * Returns true if name.isEqual("")
 */
public boolean isEmpty();

/**
 * Returns name component enumeration.
 */
public NameEnumeration components();

/**
 * Returns number of components in this name object.
 */
public int noComponents();

}
```

Listing 1: The Name interface.

Browsing over the methods, we can see immediately two main categories of methods: *query methods* and *mutation methods.*

Query methods are methods through which you request some information, but that don't change the state of the object. Examples of query methods are "String component(int)" that requests a specific name component and "boolean isEmpty()" that asks whether the name object is equal to "".

Mutation methods are methods that do something to the object (change its state). Typically, they do not return anything (for exceptions, see below). Examples of mutation methods are "void component(int i, String c)" that sets the component on index i to value c and "void remove(int i)" that removes the component on index i.

Finally, there is a third category of method types: *helper method.* You may already have noticed the "NameEnumeration components()" method. It creates a NameEnumeration object for iterating over all name components of a name object. This method is a factory method [3]. A factory method is neither a query nor a mutation method, because it does not return information about the object being queried nor does it change the object. A factory method is a *helper method,* which is a third category of method types.

3 Purpose of a method

So we have three main categories of method types: query methods, mutation methods, and helper methods. Query methods return information about the object, but don't change its state. Mutation methods change the object's state (cause a state transition), but do not return anything. Helper methods neither return information about the object nor do they change its state, but they carry out some helper task.

These three method type categories answer what a method is good for, what its purpose is. Other valid perspectives on methods are who may access a method (accessibility, visibility) and how it is implemented. This article discusses method types. It leaves out method properties, which will be discussed in a follow-up article.

The purpose point of view comprises the largest number of method types. Table 1 shows all method types that are discussed in this article. This list is by no means exhaustive! However, it covers most of the method types I have worked with and I believe that it captures the most important ones.

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: Method type categories and method types.

One rule of good Java interface design is to make a method provide exactly one service and not to overload it with too many tasks. Kent Beck programmatically states: "Divide your program into methods that perform one identifiable task" [2].

The main advantage of keeping methods simple and well focused is that you make your methods easier to understand. You also make them easier to use, because a method that is easier to understand is easier to use. In addition, small methods make a system more flexible. If you break up behavior into many small methods, you can more easily change your system. You can more easily use small methods and compose them to larger methods. You can also override smaller methods in subclasses more easily than you can override large methods.

In general, this rule leads to a method being either in the query method, mutation method, or helper method category. However, no rule is without exceptions, which we discuss after the discussion of the method type categories.

3.1 Query methods

Listing 2 shows the set of query methods from the Name interface.

```
public String asString();
public String asStringWith(char delimiter);
public String component(int index) throws InvalidIndexException;
public char delimiterChar();
public String delimiterString();
public char escapeChar();
public String escapeString();
public boolean equals(Object no);
public boolean isEmpty();
public int noComponents();
```

Listing 2: Query methods from the Name interface.

We can distinguish four different query method types: get methods, boolean query methods, comparison methods, and conversion methods.

- *Get method* (or *getter* for short). A get method is a query method that returns a (logical) field of the object. The field may be a technical field or it may be a logical field, derived from other fields. Examples of get methods are “String component(int)” to retrieve a specific name component and “char delimiterChar()” to retrieve the default delimiter char used to separate name components when representing the name as a single string.
- *Boolean query method.* A boolean query method is a query method that returns a boolean value. Examples of boolean query methods are “boolean isEqual(Name)” and “boolean isEmpty()”. Boolean query methods are so frequent and have special naming conventions that I decided to give them an extra name rather than just calling them query method.
- *Comparison method.* A comparison method is a query method that compares two objects and returns a value that tells you which object is “greater” or “lesser” than the other one according to an ordering criterion. The most well-known example is “boolean equals(Object)” to which Name adds the “boolean isEqual(Name)” method.[1] Some comparison methods are boolean query methods, but not all are necessarily so. It is a frequent idiom to return an integer in the range “-1..1” to indicate that an object is less, equal, or greater than another object.
- *Conversion method.* A conversion method is a query method that returns an object that represents the object being queried using some other (typically simpler) object. The most well-known example is “String toString()” to which Name adds the “String asString()” and “String asStringWith(char d)” methods. The “asStringWith” method represents the name object with the d character as the delimiting character for name components.

You may know these method types under some other name. For example, conversion methods are also known as converter or interpretation methods. The appended catalog lists the aliases I know.

---

[1] It is important to be precise about the meaning of equals and isEqual. Depending on the task at hand, checking for equality may provide different results. Object implements “equals” as an identity check. Specific subclasses, for example Vector, do a deep comparison of two objects for field equality. For this to return “true”, the two objects must be of the same class. Our “isEqual” method goes further and accepts instances of two different classes, for example, StringName and VectorName, as equal if they have the same abstract field values.

The same argument applies to the “toString” and “asString” methods. By definition, “toString” provides a string representation of an object usable for debugging, while “asString” provides human-readable output for a GUI. It would be confusing for clients to override the original implementation that is provided by Object.

Each method type comes with its own naming convention. Getters typically have a “get” prefix (or none at all) followed by the field being asked for. Boolean query methods start with prefixes like “is”, “has”, or “may”, followed by an indication of the flag being queried. Conversion methods start with “to” or “as” followed by the class name of the object being used as the representation. The method type catalog appended to this article lists these conventions.

3.2 Mutation methods

So what about mutation methods? What types of mutation methods are there? Listing 3 shows the set of mutation methods from the Name interface.

```
public void component(int index, String comp) throws InvalidIndexException;
public void prepend(String comp);
public void append(String comp);
public void insert(int index, String comp) throws InvalidIndexException;
public void remove(int index) throws InvalidIndexException;
```

Listing 3: Mutation methods from the Name interface.

We can distinguish between set methods and command methods.

- *Set method* (or *setter* for short). A set method is a mutation method that sets a (logical) field of an object to some value. The field may be a single technical field or it may be mapped on several fields. An example is the “void component(int i, String c)” method that sets the name component at index i to the value c. Setters are typically directed at exactly one field rather than many.
- *Command method.* A command method is a mutation method that executes a complex change to the object’s state, typically involving several fields and affecting further related objects. Examples of command methods are “void append(String)” and “void insert(int, String)”.

The appended catalog of method types lists aliases for the names of these method types and describes the naming conventions associated with them. For example, setters either have a set prefix (or none), followed by the name of the field to be updated. Command methods typically have an active verb form to indicate what they are doing.

In addition to setter and command methods, we need to take a look at the VectorName class to see a third mutation method type. Listing 4 shows the constructors of VectorName and the “void initialize(String, char, char)” method for initializing a VectorName object from the constructor parameters.

```
package org.riehle.Name.Impl;

import java.util.*;
import org.riehle.Name.*;
import org.riehle.Name.Util.*;

public class VectorName extends AbstractName {

protected Vector fComponents;

/**
 * Public convenience constructor
 */
public VectorName() {
   this("", DELIMITER_CHAR, ESCAPE_CHAR);
}
```

```
/**
 * Public convenience constructor, expects formally correct string.
 */
public VectorName(String name) {
    this(name, DELIMITER_CHAR, ESCAPE_CHAR);
}

/**
 * Public constructor, expects formally correct string.
 */
public VectorName(String name, char delimiter, char escape) {
    super();
    initialize(name, delimiter, escape);
}

/**
 * Initializes VectorName object.
 * Expects name argument to be formally correct string with delimiter and escape
chars.
 */
protected void initialize(String name, char delimiter, char escape) {
    fComponents= new Vector();
    int length= name.length();
    if (length != 0) {
        for (IntPair ip= new IntPair(0, 0); ip.y<length; ip.y++) {
            String component= StringHelper.nextString(name, delimiter, escape, ip);
            fComponents.addElement(component);
        }
    }
}

...

}
```

Listing 4: The "initialize(String, char, char)" method of VectorName and its uses.

The "void initialize(String, char, char)" method is a protected method used to initialize a VectorName object from a set of parameters. The first parameter is a string that contains all name components, properly separated by delimiter chars. The second parameter is the delimiter char itself. In the string, regular occurrences of the delimiter char are masked through an escape char, which is the third parameter.

For example, a name object for "java.lang.String" requires you to call initialize with the following parameters: "java.lang.String", '.', '\' (the escape char isn't really needed in this specific case). You could also initialize the name object with the following initialize parameters: "java#lang#String", '#', '\'. The general delimiter + escape char scheme lets you input any type of string without running into problems with reserved chars.

The "initialize" method parses the name parameter according to the delimiter and escape chars and initializes the name object accordingly. The algorithm in Listing 4 breaks up the name parameter into its name components and puts them into a Vector. The "initialize" method is called directly or indirectly from each of the three constructors "VectorName()", "VectorName(String)", and "VectorName(String, char, char)". The first two constructors are convenience constructors that assume default parameters for the "initialize" method.

Hence, we have a third type of method: initialization methods.

- *Initialization method.* An initialization method is a mutation method that sets some or all fields of an object to an initial value. Initialization methods are typically protected methods in support of public constructors, but they may also be opened up to special clients for reinitializing an existing object.

A field is initialized lazily (lazy initialization), if its initialization is deferred until it is first accessed. If this idiom is used a lot, the main initialization methods tend to initialize only a few fields, but not all.

Initialization methods are typically prefixed with "init" or called "initialize".

3.3 Helper methods

The third main category of method types is helper methods. Helper methods perform some helper task for the calling client, but typically do not change the state of the object they are part of (at least, this is not their main duty). Frequently, helper methods are static methods found in helper packages. Helper methods are also called utility methods, and helper packages are also called utility ("util") packages.

For an example, consider Listing 5 and 6.

```
public VectorName(Vector components) {
    fComponents= new Vector();
    VectorHelper.copy(components, fComponents);
}
```

Listing 5: The "VectorName(Vector)" constructor.

```
public static void copy(Vector from, Vector to) {
    for (Enumeration i= from.elements(); i.hasMoreElements(); ) {
        to.addElement(i.nextElement());
    }
}
```

Listing 6: The "void copy(Vector, Vector)" method of VectorHelper.

The "VectorName(Vector)" constructor creates a new Vector and asks the helper method "copy(Vector, Vector)" of the VectorHelper class to copy the constructor parameter into the "fComponents" Vector. Such a task is a typical duty of a helper method.

Another example is the static "String nextString(String name, char delimiter, char escape, IntPair ip)" method of the StringHelper class hinted at in the initialize method of Listing 4. Its purpose is to parse the name parameter for the next name component using the delimiter and escape chars, and to advance the parse position in the IntPair ip to continue with the next name component when called next time.

Next to such general helper methods, there are two important types of helper methods that deserve special attention. We already mentioned the factory method "NameEnumeration components()" from the Name interface.

- *Factory method.* A factory method is a helper method that creates an object and returns it to the client [3].

Factory methods are sometimes prefixed with "create", "make", or "new", followed by the product name.

The second important type of helper methods is the assertion method type. To understand this type of method, consider Listing 7 and 8.

```
public void remove(int index) throws InvalidIndexException {
    assertIsValidIndex(index);
    fComponents.removeElementAt(index);
}
```

Listing 7: The "void remove(int)" method of VectorName.

```
protected void assertIsValidIndex(int i) throws InvalidIndexException {
    if ((i < 0) || (i >= noComponents())) {
        throw new InvalidIndexException(i, noComponents());
    }
}
```

Listing 8: The "void assertIsValidIndex(int)" method of VectorName.

Listing 7 shows how the “void remove(int i)” method of VectorName calls the “void assertIsValidIndex(int i)” method to ensure that the index parameter i is in the range of valid indices. Only if this precondition is fulfilled, does the remove method work properly. Otherwise, an exception is thrown back to the client that called the remove method. The “assertIsValidIndex” method is an assertion method.

- *Assertion method.* An assertion method is a helper method that checks whether a certain condition holds. If the condition holds, it returns silently. If it does not hold, an appropriate exception is thrown.

Assertion methods are part of the larger design-by-contract scheme of designing interfaces [4]. They are used to ensure at runtime that preconditions and postconditions of methods and invariants of classes are maintained.

3.4 Exceptions to the rule: methods with several purposes

You may wonder how well the distinction between method types works in practice. In my experience, it works about 90% of the time, which is good enough to think of methods as being of a specific type. The remaining 10% of methods mix different purposes.

Consider a get method that logs the access to the field it provides. For example, “String component(int)” can be implemented to increment a counter each time it is called, so that a user can ask the name object how often the component method has been called using some other get method. Is the “String component(int)” method a query or a mutation method?

The primary purpose of this method is to return information about the object, so it is a query method, with a mutation method aspect thrown in. It is obvious that the clean conceptual scheme discussed above does not work nicely in this situation. Still the query method aspect is dominating the mutation method aspect, which justifies calling it a query method. From a practical perspective, you simply need to add to the method documentation that this method is a query method that increments a counter as a side-effect.

Another well-known example is the “Object nextElement()” method of Enumeration for iterating over a collection. This method does three things: it first marks the current element, then advances the position pointer, and finally returns the former current element to the client. Again, this method has both a query and a mutation aspect to it. In my opinion, the query aspect is stronger than the mutation aspect, so I think it is a query method.

From a design perspective, it is cleaner to have two separate methods: one that returns the current element and another one that advances the position pointer. From a practical perspective, you can merge these two methods if you know what you are doing. And in fact, iterating over a collection is such a well-known idiom that every seasoned Java developer knows what he or she is doing when calling “nextElement()”.

Yet another common idiom is lazy initialization, as discussed above under initialization methods. Here, a get method is also a hidden initialization method. Again, the main purpose of the method is to return a field’s value, which is implemented by initializing the field in the first place before returning its value.

Whenever I face a situation of mixing different method types in one method, I think hard about whether it isn’t better to split up the method into two. Most of the time, it is. If not, I don’t worry too much either. Methods are there for us to make life easier, not to make it more complicated, because some abstract scheme tells us so.

4 What have we gained?

Armed with the vocabulary of method types, we can both better design and document the code. For example, we can introduce a Javadoc tag @methodtype and then provide the name of the method’s type. This tag helps concisely convey information about the method being documented, much more effectively than a verbose description could do.

Listing 9 presents some sample annotations.

```
/**
 * Gets name component.
 * @methodtype get
 */
public String component(int index) throws InvalidIndexException;

/**
 * Sets name component.
 * @methodtype set
 */
public void component(int index, String comp) throws InvalidIndexException;

/**
 * Returns a simple human readable description.
 * @methodtype conversion
 */
public String asString();

/**
 * Inserts a name component.
 * @methodtype command
 */
public void insert(int index, String comp) throws InvalidIndexException;

/**
 * Returns name component enumeration.
 * @methodtype factory
 */
public NameEnumeration components();

...
```

Listing 9: Sample of methods annotated with method type names.

In the follow-up article, to the annotations of Listing 9, we add method properties like “convenience method” or “template method” that concisely document how a method is to be used and how it is implemented.

5 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 types and gives them a precise definition. If you feel that it leaves out some key method types, let me know. If you know further aliases for the names of the method types provided here, let me also know. At “http://www.riehle.org/java-report/” you can find a growing catalog of these and other method types as well as the source code of the examples.

6 Acknowledgments

I would like to thank my colleagues Dirk Bäumer, Daniel Megert, Wolf Siberski, and Hans Wegener for reading and commenting on this article.

The method type names are taken from several different sources:

- Arnold and Gosling's Java the Language [1],
- Kent Beck's book Smalltalk Best Practice Patterns [2],
- Gang-of-four's book Design Patterns [3],

and, most importantly, from common knowledge and use.

7 References

[1] Ken Arnold and James Gosling. Java, the Programming Language. Addison-Wesley, 1996.

[2] Kent Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1996.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, 1995.

[4] Bertrand Meyer. Object-oriented Software Construction, 2nd Edition. Prentice-Hall, 1998.

[5] 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.

[6] 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.

8 Catalog of method types

We distinguish three main categories of method types: query methods, mutation methods, and helper methods.

8.1 Query methods

A query method (aka accessing method) is a method that returns some information about the object being queried. It does not change the object's state. There are four main query method types: get method, boolean query method, comparison method, and conversion method.

8.1.1 Get method

Definition:	A get method is a query method that returns a (logical) field of the object.
Also known as:	Getter.
JDK example:	Class Object::getClass(), Object Enumeration::nextElement().
Name example:	String Name::component(int), NameEnumeration Name::components().
Prefixes:	(If any:) get.

Naming:	After the prefix (if any), the name of the field being queried follows.

8.1.2 Boolean query method

Definition:	A boolean query method is a query method that returns a boolean value.
Also known as:	–
JDK example:	boolean Object::equals().
Name example:	boolean Name::isEmpty().
Prefixes:	is, has, may, can, ...
Naming:	After the prefix, the aspect being queried follows.

8.1.3 Comparison method

Definition:	A comparison method is a query method that compares two objects and returns a value that tells you which object is "greater" or "lesser" than the other one according to an ordering criterion.
Also known as:	Comparing method.
JDK example:	boolean Object::equals(Object), int Comparable::compareTo(Object).
Name example:	boolean Name::isEqual(Object).
Prefix:	If also a boolean query method, see boolean query method.
Naming:	If also a boolean query method, see boolean query method.
Comment:	A comparison method is not a specialization of boolean query method, because many comparison methods return a ternary value, indicating less, equal, or greater.

8.1.4 Conversion method

Definition:	A conversion method is a query method that returns an object that represents the object being queried using some other (typically simpler) object.
Also known as:	Converter method, interpretation method.
JDK example:	String Object::toString(), int Integer::intValue().
Name example:	String Name::asString(), String Name::asStringWith(char).
Prefix:	as, to.
Naming	After the prefix, typically the class name being converted to follows.

8.2 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. There are three main mutation method types: set method, command method, and initialization method.

8.2.1 Set method

Definition:	A set method is a mutation method that sets a (logical) field of an object to some value.

Also known as:	Setter.
JDK example:	void Thread::setPriority(int), void Vector::addElement(Object).
Name example:	void Name::component(String).
Prefix:	(If any:) set.
Naming:	After the prefix (if any), the field being changed follows.

8.2.2 Command method

Definition:	A command method is a method that executes a complex change to the object's state, typically involving several fields and affecting further related objects.
Also known as:	–
JDK example:	void Object::notify(), void JComponent::repaint().
Name example:	void Name::insert(int i, String c), void Name::remove(int i).
Prefixes:	(If any:) handle, execute, make.
Naming:	–

8.2.3 Initialization method

Definition:	An initialization method is a mutation method that sets some or all fields of an object to an initial value.
Also known as:	–
JDK example:	void LookAndFeel::initialize().
Name example:	void StringName::initialize(), void VectorName::initialize().
Prefixes:	init, initialize.
Naming:	If prefixed with init, typically the name of the object part being initialized follows.

8.3 Helper methods

A helper method (aka utility method) is a method that performs some support task for the calling object. A helper method does not change the object's state, but only performs operations on the method arguments. Frequently, a helper method is a static method and provided by some helper (utility) class.

8.3.1 Factory method

Definition:	A factory method is a helper method that creates an object and returns it to the client.
Also known as:	Object creation method.
JDK example:	String String::valueOf(int), String String::valueOf(double).
Name example:	NameEnumeration Name::components(), Name StringName::newName(String).
Prefixes:	(If any:) new, create, make, build.
Naming:	After the prefix, the product name follows.
Comment:	See [3].

8.3.2 Assertion method

Definition:	An assertion method is a helper method that checks whether a certain condition holds. If the condition holds, it returns silently. If it does not hold, an appropriate exception is thrown.
Also known as:	–
JDK example:	–
Name example:	void Name::assertIsValidIndex(int) throws InvalidIndexException.
Prefixes:	assert, check, test.
Naming:	After the prefix, the condition being checked follows.
Comment:	Assertion methods are used to assert invariants about an object, for example preconditions upon method entry and post-conditions upon method completion. They are part of the larger design-by-contract scheme [4].