



DevelopIntelligence
Developing Developers

A Quick Look at Static Imports

Written By: Kelby B. Zorgdrager

Static Imports

Version 1.1

Publication date: Feb 29, 2008

This material is licensed by DevelopIntelligence LLC and shall not be reproduced, edited, or distributed, in hard or soft copy format, without the written consent of DevelopIntelligence.

Information in this document is subject to change without notice. Companies, names, and data used in the examples herein are fictitious unless otherwise noted.

Static Imports is a publication of DevelopIntelligence, LLC. For more information regarding this publication or others, please contact us via email info@DevelopIntelligence.com, or by phone (303) 395-5340.

Java, Java Standard Edition, Java SE, Java Enterprise Edition, Java EE, Enterprise JavaBeans, and all other Java-based trademarks and logo trademarks are registered trademarks of Sun Microsystems, Inc., in the United States and other countries. All other products referenced herein are trademarks of their respective holders.

Copyright 2008 DevelopIntelligence LLC. All rights reserved.

Static Imports

Quick Facts

- Introduced in Java SE 5.0
- Part of JSR 201
- Expanded import facility to include static members
- Support single and on-demand import styles

Purpose

- Simplify reference to static members in source code
- Restore cohesion

Key Syntax

- `import static type.member>;`
- `import static <type.*>;`

Example Syntax

- `import static java.lang.Math.PI;`
- `import static java.lang.Math.*;`

Best Practices

- Be aware
- Be specific
- Refactor old code

Overview

Have you ever written a Java program that relies on constant variables (static variables), like `Math.PI`? Have you ever written a Java program that relies on utility functions (static methods) like `Math.abs()`? Have you ever created the oh-so-convenient, global variables in Java? If you have, you probably used some form of import functionality.

Import Facility

The import facility found within the Java language functions as a convenience method, used within source code, to reference a type by its simple name (class name) instead of its fully-qualified class name.

Import Features

The import facility supports two types of uses, a single-type import and an on-demand-type import. The single-type provides simplified reference to a single type within a package. The on-demand-type import provides simplified reference to all of the types within a package.

Static Import Features

In Java SE 5.0, the import facility was expanded to support simplified reference to static members. Like the standard facility, the expanded facility supports both single and on-demand imports.

Differences

The import facility is focused on simplifying references to types in source code. The static import facility is focused on simplifying references to static members, found within types, in source code.

Purpose

The primary purpose of the static import facility is to simplify references to static members in source code. The second purpose is to restore cohesion to class hierarchies. Some developers adopted work around to simplify working with static variables. This work-around commonly broke cohesive object oriented design.

Best Practices

When using static imports, be aware of any name space collisions that might occur between imported static members and locally declared members. When they occur, consider renaming the local members to remove

the collision. Static imports, like regular imports, can make your code hard to read. Consider using only the single-static import syntax to prevent any possible confusion. If you find that static imports simplify your code, consider refactoring your old code.

Related Concepts

- Import facility
- Type
- Fully-qualified class name
- Package
- Static members
- Class hierarchy
- Cohesion

APIs Used

- `java.lang.Math`
- `java.lang.Math.PI`
- `java.lang.Math.round()`

Source Code Examples

- `SingleStaticImport`
- `OnDemandStaticImport`

Usage Scenario

Assume we wanted to calculate the diameter of a circle and then round it to the nearest number.

The class, `java.lang.Math`, provides static members that can help us achieve this. The `Math` class contains a static variable for `PI`, `Math.PI`, and a static method for number rounding, `Math.round()`.

Static Import Syntax

The static import facility supports two syntaxes: single-static import and on-demand-static import. The single-static import simplifies referencing a single static member in a class. The on-demand-static import simplifies referencing all static members in a class.

Single-Static Import Example

This examples illustrates the use of the single-static import syntax. Lines 3 and 4 contain example usages of the single-static import mechanism. Lines 15 and 16 illustrate the simplification the static import provides. Notice each static member used within the program has its own static import statement.

A Quick Look at Static Imports

```
1. package examples.staticimport;
2. import static java.lang.Math.PI;
3. import static java.lang.Math.round;
4. /**
5.  * SingleStaticImport is a sample illustration of using the
6.  * single-static-import declaration to simplify access to a static member
7.  * within the java.lang.Math class.
8.  */
9. class SingleStaticImport {
10.
11.     public static void main(String [] args) {
12.         double circumference = 7.7;
13.         double diameter = circumference * PI;
14.         double roundedDiameter = round(circumference);
15.         System.out.println("The diameter of the circle is: " + diameter);
16.         System.out.println("The rounded diameter of the circle is: " + diameter);
17.     }
18. }
```

On-Demand-Static Import Example

This examples illustrates the use of the on-demand-static import syntax. Line 3 contains an example usage of the on-demand static import. Notice the reference to the static members does not change from the previous example.

```
1. package examples.staticimport;
2. import static java.lang.Math.*;
3. /**
4.  * OnDemandStaticImport is an example illustration of using the
5.  * ondemand-static-import declaration to simplify access to
6.  * static members within the java.lang.Math class.
7.  */
8. class OnDemandStaticImport {
9.
10.     public static void main(String [] args) {
11.         double circumference = 7.7;
12.         double diameter = circumference * PI;
13.         double roundedDiameter = round(circumference);
14.         System.out.println("The diameter of the circle is: " +
15.                             diameter);
15.         System.out.println("The rounded diameter is: " +
16.                             roundedDiameter);
16.     }
17. }
```

Purpose

The purpose of this lab is to experiment with the static import syntax, noticing how the syntax has a different implication from the traditional import statement syntax.

Static Import Lab

Description

In this lab, you will create a stand-alone Java application called Mixer. The Mixer application's functionality is dependent upon command-line arguments.

There are two basic business rules that govern the functionality of the Mixer:

If Mixer receives 3 or less arguments, Mixer should sort the arguments using `Arrays.sort` and print the results.

If Mixer receives more than 3 arguments, Mixer should sort the arguments, count the frequency

of each argument, and print the argument and its frequency in sorted order.

From a source perspective, Mixer should be written to leverage the static import facility.

Example Execution

3 or less arguments:

```
> java
labs.solutions.staticimport.Mixer
one two three
[one, three, two]
```

3 or more arguments

```
> java
labs.solutions.staticimport.Mixer
one two three four one
four=1
one=2
three=1
two=1
```

Duration

This lab should take you no more than 15 minutes.

DevelopIntelligence

About DevelopIntelligence

DevelopIntelligence is a software development productivity company. We have helped over 400 software development teams increase their productivity through the delivery of training, employee development programs, and educational consulting.



Our Services

Training

- 🕒 Seminars and Workshops
- 🕒 Hands-on and eLearning
- 🕒 Workshops and Boot camps

Employee Development

- 🕒 Coaching and Mentoring
- 🕒 New Hire Development
- 🕒 Team Migration planning

Educational Consulting

- 🕒 Curriculum design
- 🕒 Content creation
- 🕒 Program development

Our Methodology

Our proven **evaluate.architect.construct.** methodology examines the business drivers, the skill gaps, technology roadmap, and productivity objectives to create a client-focused, project-centric, customized productivity program. This program maps out the best continuous learning approach, processes, job aides, and team structuring required to achieve all of your goals.

Our Expertise

- 🕒 40,000+ Developers Trained
- 🕒 40+ Years of Software Development
- 🕒 35+ Years of Technical Training
- 🕒 12 Java certifications
- 🕒 4 Best-selling Java Books

Our History

Founded in 2003 with a single vision:

"help software teams develop better software"

DevelopIntelligence's team represents more than 40 years of software development experience coupled with over 35 years of technical training. Over the years, we have leveraged our experiences in business, training and software development to help clients of all sizes create productivity within their software development teams.