

Dhanya Kumar K.V

Jan 07, 2008



(Graham Hamilton, Jonathan Schwartz, and James Gosling)

Overview

Open Source has further evolved to simplify the programming and to improve the performance. The Java 2 Platform Standard Edition (J2SE) 5.0 ("Tiger") is the next major revision to the Java platform and language. This article briefly explains the new language features of Java 5.0 and JUnit 4. These exciting new features you've probably heard about but may not yet be using.

Acronyms

Acronyms which are repeatedly used in this article:

CTE - Compile-time error NCTE - No Compile-time error
RTE- Runtime error

System requirements

To follow along and try out the code for this tutorial, you need a working installation of Sun's JDK 1.5.x, Eclipse Europa (3.3) and Ant 1.7 or later versions of these tools.

Why Eclipse 3.3 and Ant 1.7?

- Eclipse 3.3 comes with JUnit 4 plug-in.
- If you want to run your JUnit 4 tests on a version of Ant prior to 1.7, you must retrofit the test case with a suite() method that returns an instance of JUnit4TestAdapter, as shown below:

```
public static junit.framework.Test suite() {  
    return new JUnit4TestAdapter(RegularExpressionTest.class);  
}
```

If you are using Ant 1.6 or older versions, upgrade it to 1.7 or later version's since these new releases support annotations and also provides backward compatibility. **Ant 1.6 or older versions for JUnit 4.x is NOT recommended.**



New Language Features

The features listed below are introduced in 5.0 since the previous major release (1.4.0)

- Generics
- Autoboxing/Unboxing
- Enhanced for Loop
- Static Import
- StringBuilder
- Varargs
- Typesafe Enums
- Covariant Returns
- Metadata (Annotations)

Generics

Prior to Java5, when you take an element out of a Collection, you must cast it to the type of element that is stored in the collection. Generic eliminates an unsafe cast and a number of extra parentheses. Generics are implemented by type erasure: generic type information is present only at compile time, after which it is erased by the compiler. This allows you to mix the generic code with legacy code. If you mix Generic and Non-generic Collections, compiler gives a warning message. The code using generics is clearer and safer.

```
public class GenericsExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<String>(); // Generics
        names.add("Tiger"); // takes only String objects
        names.add(100); // CTE, type-safety!
        names.get(0); // Cast not required, No RTE
    }
}
```

Polymorphism does not work the same way for generics as it does with arrays. For generics, the type of the variable declaration must match the type you pass to the actual object type.

```
Number[] numbers = new Integer[5]; // Polymorphism YES
List<Number> num = new ArrayList<Integer> (); //CTE, NO Polymorphism
List<Integer> num = new ArrayList<Integer> (); // No problem!
```

The wildcard `<?>` stops adding elements to the Collection parameter. By saying `<? extends Animal>`, we're saying, "I can be assigned a collection that is a subtype of List and typed for `<Animal>` or anything that extends Animal.

The keyword `super` and `<?>` allows to add objects to the collection in safe way. "Hey compiler, please accept any List with a generic type that is of type Dog, or a supertype of Dog. Nothing lower in the inheritance tree can come in, but anything higher than Dog is OK".

“The keyword `extends` / `super` in the context of a wildcard represents BOTH subclasses and interface implementations”.

```
class Animal { }

class Dog extends Animal { }

public class GenericsExample {
    public static void main(String[] args) {
        // polymorphism doesnt supports here
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Dog()); // NCTE !!!

        List<Dog> dogs = new ArrayList<Dog>();
        add(dogs); // NCTE, Dog extends Animal
        add(dogs); // If Dog doesnt extends Animal, then expect CTE
    }

    static void add(List<? extends Animal> list) {
        list.add(new Animal()); // CTE, you cannot add anything to list
    }

    // This method doesn't overloads add(), both takes List param. Thus CTE
    static void add(List<? super Dog> list) {
        list.add(new Dog()); // NCTE, super used
    }
}
```

Autoboxing/Unboxing

Autoboxing converts primitives to wrapper objects automatically and unboxing convert's wrapper objects to primitives. These auto conversions have performance issues if used improperly. So when to use boxing? Use them only when there is an impedance mismatch between reference types and primitives, for example, when you have to put numerical values into a collection.

```
public class AutoBox {
    public static void main(String args[]) {
        Integer num1 = 100; // autobox an int
        int num2 = num1; // auto-unbox
        System.out.println(num1 + " " + num2); // displays 100 100
        List numbers = new ArrayList();
        numbers.add(num2); // autobox : int to Integer then Object
        num2 = numbers.get(0); // CTE, Wrapper to primitive possible
    }
}
```

Enhanced for Loop

Aka for-each loop. The for-each simplifies looping through an array or a collection and also beautifies your code. The for-each construct hides the index variable for arrays and the iterator for the collections.

Drawbacks of for-each loop: The for-each loop hides the iterator, so you cannot call remove. Therefore, the for-each loop is not usable for filtering. Similarly it is not usable for loops where you need to replace elements in a list or array as you traverse it. Finally, it is not usable for loops that must iterate over multiple collections in parallel.

```
public class ForeachArray {
    public static void main(String args[]) {
        String[] data = { "JavaTiger", "Ant1.7", "JUnit4", "Eclipse3.3" };
        for (String s : data) {
            System.out.println(s);
        }
    }
}
```

Static Import

Read as 'static import' but write as 'import static'. In order to access static members, it is necessary to qualify references with the class they came from. For example, one must say: `double r = Math.cos(Math.PI * 20);`

The static import construct allows unqualified access to static members without inheriting from the type containing the static members. Instead, the program imports the members, either individually:

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

Once the static members have been imported, they may be used without qualification: `double r = cos(PI * 20);`

Used appropriately, static import can make your program more readable, by removing the boilerplate of repetition of class names.

This feature has more drawbacks than benefits. DON'T OVERUSE IT IN YOUR PROGRAM.

Drawbacks of Static import:

- If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all the static members you import.
- All of the static members from a class can be particularly harmful to readability; if you need only one or two members, import them individually.
- You can't import same static member from two different classes

```
import static java.lang.Long.MAX_VALUE;  
import static java.lang.Integer.MAX_VALUE; // CTE
```

StringBuilder

String objects are immutable. StringBuilder and StringBuffer are like a String, but can be modified. StringBuilder class is same as StringBuffer class,

except that its methods are non-synchronized. Use `StringBuffer` for thread-safe operations and `StringBuilder` for better performance.

```
public class StringFamily {
    public static void main(String[] args) {
        StringBuffer buffer = new StringBuffer();
        StringBuilder builder = new StringBuilder();
        long bufferTime = Calendar.getInstance().getTimeInMillis();
        for (int i = 0; i < 1000000; i++) {
            buffer.append("JavaTiger");
        }
        System.out.println("Buffer manipulation time (Approx) : "
            + (Calendar.getInstance().getTimeInMillis() - bufferTime));

        long builderTime = Calendar.getInstance().getTimeInMillis();
        for (int i = 0; i < 1000000; i++) {
            builder.append("JavaTiger");
        }
        System.out.println("Builder manipulation time (Approx) : "
            + (Calendar.getInstance().getTimeInMillis() - builderTime));
    }
}
```

Result:

```
Buffer manipulation time (Approx) : 390
Builder manipulation time (Approx) : 266
```

Var-args

To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space, and then the name of the array that will hold the parameters received. This var-args will be the last parameter in the method. And only one var-args is allowed per method.

Generally speaking, you should not overload a var-args method, or it will be difficult for programmers to figure out which overloading gets called.

```
public class VarArgs {
    public static void main(String[] args) {
        print("Maths test marks : ", 99, 98, 97, 35, 50);
    }

    public static void print(String subject, int... marks) {
        System.out.print(subject);
        for (int i : marks) {
            System.out.print(i + " ");
        }
    }
}
```

Output:

```
Maths test marks : 99 98 97 35 50
```

Enum

Enum's represents a fixed set of type-safe constants. Enum overrides toString(), hashCode() and equals() methods, so it's a good candidate for Hashtable, HashMap and other Hash collections. As of Java 5, enum can be used with switch statements.

Enums can be declared as their own separate class, or as a class member, however they must not be declared within a method. If declared in separate class, only public or default visibility modifier is allowed.

What gets created when you make an enum?

The most important thing to remember is that enums are not Strings or ints! Each of the enumerated Day types are actually instances of Day. In other words, SUNDAY is of type Day

```
public class EnumExample {
    private enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    }

    public static void main(String[] args) {
        for (Day day : Day.values()) {
            System.out.println(day);
        }
    }
}
```

Covariant Returns

Prior to Java 5, when you override a method, the return type should be the same as overridden method. As of Java 5, you can return same type or the subclass of the type. For example,

```
class Animal {
    Animal getAnimal() {
        return new Animal();
    }
}

class Dog extends Animal {
    Dog getAnimal() {
        return new Dog();
    }
}
```

Annotations

Annotation (also known as metadata) facility that permits you to define and use your own annotation types. Annotation Processing Tool (apt) is a command-line utility for annotation processing. It includes a set of reflective APIs and supporting infrastructure to process program annotations. apt first runs annotation processors that can produce new source code and other files. Next, apt can cause compilation of both original and generated source files, thus easing the development cycle.

Annotation type declarations are similar to normal interface declarations. An at-sign (@) precedes the **interface** keyword. Each no-arg method declaration defines an element of the annotation type. Methods can have compile-time constant values. Method Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types. Void not acceptable!

```
@RequestForEnhancement(  
    id      = 2868724,  
    synopsis = "Enable time-travel",  
    engineer = "Mr. Tiger",  
    date    = "01/01/2008"  
)
```

Meta-annotations, which are actually known as the annotations of annotations, contain four types. These are: Target, Retention, Documented and Inherited. Only Target and Retention are explained here since these two meta-annotations are frequently used.

Target: The target annotation indicates the targeted elements of a class in which the annotation type will be applicable. It contains the following enumerated types as its value:

`@Target(ElementType.TYPE)` —can be applied to any element of a class
`@Target(ElementType.FIELD)` —can be applied to a field or property
`@Target(ElementType.METHOD)` —can be applied to a method level annotation
`@Target(ElementType.PARAMETER)` —can be applied to the parameters of a method
`@Target(ElementType.CONSTRUCTOR)` —can be applied to constructors
`@Target(ElementType.LOCAL_VARIABLE)` —can be applied to local variables
`@Target(ElementType.ANNOTATION_TYPE)` —indicates that the declared type itself is an annotation type

Retention: The retention annotation indicates where and how long annotations with this type are to be retained. There are three values:

- `RetentionPolicy.SOURCE`—Annotations with this type will be retained only at the source level and will be ignored by the compiler
- `RetentionPolicy.CLASS`—Annotations with this type will be retained by the compiler at compile time, but will be ignored by the VM
- `RetentionPolicy.RUNTIME`—Annotations with this type will be retained by the VM so they can be read only at run-time

Example:

```
// Test.java
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Indicates that the annotated method is a test method. This
 * annotation should be used only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { } // marker annotation, no elements
```

```

// Foo.java

public class Foo {
    @Test
    public static void m1() {
        System.out.println("m1");
    }

    @Test
    public static void m3() {
        throw new RuntimeException("Boom");
    }

    public static void m4() {
    }

    @Test
    public static void m5() {
    }
}

```

```

// RunTests.java
import java.lang.reflect.Method;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int passed = 0, failed = 0;
        for (Method m : Class.forName("Foo").getMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                try {
                    m.invoke(null);
                    passed++;
                } catch (Throwable ex) {
                    System.out.printf("Test %s failed: %s %n", m, ex.getCause());
                    failed++;
                }
            }
        }
        System.out.printf("Passed: %d, Failed %d%n", passed, failed);
    }
}

```

Output:

```

m1
Test public static void Foo.m3() failed: java.lang.RuntimeException: Boom
Passed: 2, Failed 1

```



For JUnit 3.x to recognize a class object containing tests, the class itself was required to extend from JUnit's `TestCase` (or some derivation thereof). And to recognize the test methods, the method name need to be prefixed with `test`.

JUnit 4 uses annotations to completely eliminate these two constraints.

Example: Pre JUnit 4 Test class

```
import junit.framework.Assert;
import junit.framework.TestCase;

public class SimpleTest extends TestCase {

    public void testCompare() { // Pre JUnit 4
        Assert.assertEquals(10, 10);
    }
}
```

1. What makes the framework to ignore `testCompare()`?

Remove or modify the prefix "test". But it intrudes a side effect since only one test method is available in `SimpleTest`. Framework reports `AssertionFailedError: No tests found in SimpleTest`

Example: JUnit 4 Test class

```
import static junit.framework.Assert.*; // Java 5 Feature
import org.junit.Test;

public class JUnit4Test { // No extends keyword

    @Test
    public void compare() { // no test prefix
        assertEquals(10, 10); // access thru static import
    }
}
```



```
import org.junit.Test;

public class ExceptionJUnit4Test {
    @Test(expected = NullPointerException.class)
    public void verifyNullException() throws Exception {
        String as = null;
        as.length();
    }
}

Result:
The test runs successfully since the test handles the exception
```

```
import org.junit.Test;

public class ExceptionJUnit4Test {
    @Test(expected = IllegalArgumentException.class)
    public void verifyNullException() throws Exception {
        String as = null;
        as.length();
    }
}

Result:
Un expected exception so the test fails.
```

Timeout test

In JUnit 4, a test case can take a **timeout** value as a parameter. The timeout value represents the maximum amount of time [in milliseconds] the test can take to run: if the time is exceeded, the test fails. Mainly used for performance test. For example,

```
import org.junit.Test;

public class StringOperationsJUnit4Test {
    @Test(timeout = 250)
    public void bufferOperationTimeTest() throws Exception {
        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < 1000000; i++) {
            buffer.append("JavaTiger");
        }
    }

    @Test(timeout = 160)
    public void builderOperationTimeTest() {
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < 1000000; i++) {
            builder.append("JavaTiger");
        }
    }
}
```

```
}  
Result:  
bufferOperationTimeTest() fails.  
Refer the output of StringFamily for time taken to append 'JavaTiger'  
1000000 times
```

Test fixtures

Test fixtures aren't new to JUnit 4, but the fixture model is improved. Fixtures foster reuse through a contract that ensures that particular logic is run either before or after a test. This logic, for example, could be initializing a class that you will test in multiple test cases.

Older versions of JUnit employed a somewhat inflexible fixture model with `setUp()` and `tearDown()` methods which are executed once for each test defined.

There are four fixture annotations: two for class-level fixtures and two for method-level ones.

- At the class level, you have `@BeforeClass` and `@AfterClass` and
- At the method (or test) level, you have `@Before` and `@After`.

For `@BeforeClass` and `@AfterClass`, the methods should be public and static. For `@Before` and `@After`, the methods should be public and non-static. You can specify more than one fixture for a test case in JUnit 4. Keep in mind, however, that as of the current version of JUnit 4, you cannot specify which fixture methods are to be run first, which can get tricky if you decide to use more than one.

```
import static org.junit.Assert.assertFalse;  
import static org.junit.Assert.assertTrue;  
  
import org.junit.AfterClass;  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
public class EmpNameJUnit4Test {  
    private static String name = null;  
}
```

```

@BeforeClass
public static void setUpBeforeClass() throws Exception {
    name = "James W Bond";
    System.out.println("Inside setUpBeforeClass");
}

@Test
public void firstName() throws Exception {
    int index = name.indexOf("James");
    assertTrue("First Name found", index == 0);
}

@Test
public void lastName() {
    int index = name.indexOf("Bond");
    assertFalse("Last Name not found", index == -1);
}

@AfterClass
public static void tearDownClass() throws Exception {
    System.out.println("Inside tearDownClass");
}
}

Result:
Both test methods succeeds. And prints the statement:
Inside setUpBeforeClass
Inside tearDownClass

```

Now, replace the `@BeforeClass` and `@AfterClass` with `@Before` and `@After` respectively and execute the program. The test fixture fails with an error: `tearDownClass()` and `setUpBeforeClass()` should be non-static.

Remove the modifier `static` from both the methods and execute the program, it prints "Inside setUpBeforeClass" and "Inside setUpBeforeClass" each twice.

TestSuite: logically group tests

Test Suite is not new to JUnit 4, but the semantics are improved with two annotations `@RunWith` and `@SuiteClasses`.

JUnit 4 bundles a suite runner, named `Suite` that you must specify in the `@RunWith` annotation. `@SuiteClasses`, which takes as a parameter a list of classes intended to represent the test suite.

Pre JUnit 4 Code Snippet:

```
public class JUnit3Suite {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(OneTimeRegularExpressionJUnit3Test.suite());
        suite.addTestSuite(RegularExpressionJUnit3Test.class);
        return suite;
    }
}
```

JUnit 4 TestSuite Example:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

import testfixture.EmpNameJUnit4Test;
import testingexception.ExceptionJUnit4Test;

@RunWith(Suite.class)
@SuiteClasses( { EmpNameJUnit4Test.class, ExceptionJUnit4Test.class } )
public class JUnit4Suite {

}
```

Parametric testing

JUnit 4 introduces an excellent new feature that lets you create generic tests that can be fed by parametric values. As a result, you can create a single test case and run it multiple times -- once for every parameter you've created.

It takes just five steps to create a parametric test in JUnit 4:

- 1) Create a generic test that takes no parameters.
- 2) Create a static feeder method that returns a Collection type and decorate it with the @Parameter annotation.
- 3) Create class members for the parameter types required in the generic method defined in Step 1.
- 4) Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3.

- 5) Specify the test case be run with the `Parameterized` class via the `@RunWith` annotation.

Example: Verifies the zip code

```
import static org.junit.Assert.assertEquals;

import java.util.Arrays;
import java.util.Collection;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class ParametricRegularExpressionTest {
    private static String zipRegex = "^\\d{5}([\\-]\\d{4})?$";

    private static Pattern pattern;
    private String phrase;
    private boolean match;

    public ParametricRegularExpressionTest(String phrase, boolean
                                           match) {
        this.phrase = phrase;
        this.match = match;
    }

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        pattern = Pattern.compile(zipRegex);
    }

    @Parameters
    public static Collection regExValues() {
        return Arrays.asList(new Object[][] { { "22101", true },
        { "221x1", false }, { "22101-5150", true },
        { "221015150", false } });
    }

    @Test
    public void verifyZipCode() throws Exception {
        System.out.println("Inside verifyGoodZipCode");
        Matcher mtcher = this.pattern.matcher(phrase);
        boolean isValid = mtcher.matches();
        assertEquals("Pattern did not validate zip code", isValid,
            match);
    }
}
```

A new assert

JUnit 4 has added a new assert method for comparing array contents. It isn't a big addition, but it does mean you'll never again have to iterate over the contents of an array and assert each individual item.

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CompareArrayElementsJUnit4Test {

    @Test
    public void verifyArrayContents() throws Exception {
        String[] actual = new String[] { "JUnit 3.8.x", "JUnit
                                         4.1", "TestNG" };
        String[] var = new String[] { "JUnit 3.8.x", "JUnit 4.1",
                                         "TestNG 5.5" };
        assertEquals("the two arrays should not be equal", actual,
                    var);
    }
}
```

Note: The `assertEquals` for comparing arrays is deprecated. Better use `ArrayUtils` class.

Related Resources:

Java 5:

- <http://java.sun.com/developer/technicalArticles/releases/j2se15/>
- <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html>

JUnit

- <http://www.junit.org/>
- <http://www.junit.org/taxonomy/term/12>
- <http://junit.sourceforge.net/javadoc/junit/framework/package-summary.html>

Queries/Feedback

Please post your queries or feedback to dhanya@dhanyakumar.com or info@dhanyakumar.com

PS: content of this article is compiled from various websites of the net.