Java 17 Features

**Date 03/05/2023**

# Overview of Java 17 (Sept 14, 2021)

1. java.time.InstantSource
2. JEP 406 - Pattern Matching for Switch Expression
3. JEP 306 - Restore Always Strict Floating-Point Semantics
4. JEP 356 - Enhanced Pseudo-Random Number Generators
5. JEP 415 - Context-Specific Deserialization Filters

# java.time.InstantSource

<https://stackoverflow.com/questions/69199722/what-is-the-main-purpose-of-the-new-interface-java-time-instantsource-over-exist>

<https://www.baeldung.com/java-instantsource>

The java.time.Clock class is handy for writing tests that check time-dependent functionality.

For example, when Clock is injected into the application classes via dependency injection, it can be mocked into tests, or a fixed time for test execution can be set using Clock.fixed().

Since Clock provides the getZone() method, you always have to think about which concrete time zone to instantiate a Clock object with.

To allow alternative, time zone-independent time sources, the interface java.time.InstantSource was extracted from Clock in **Java 17**. The new interface only provides the methods instant() and millis() for querying the time, where millis() is already implemented as a default method.

The Timer class in the following example uses InstantSource to determine the start and end times of a Runnable execution and uses those times to calculate the duration of execution:

**public class Timer {**

**private final InstantSource instantSource;**

**public Timer(InstantSource instantSource) {**

**this.instantSource = instantSource;**

**}**

**public Duration measure(Runnable runnable) {**

**Instant start = instantSource.instant();**

**runnable.run();**

**Instant end = instantSource.instant();**

**return Duration.between(start, end);**

**}**

**}**

In production, we can instantiate Timer with the system clock (where, for lack of alternative InstantSource implementations, we have to worry about the time zone - let's take the system's default time zone).

**Timer timer = new Timer(Clock.systemDefaultZone());**

We can test the measure() method by mocking InstantSource, having its instant() method return two fixed values, and comparing the return value of measure() with the difference of these values:

**@Test**

**void shouldReturnDurationBetweenStartAndEnd() {**

**InstantSource instantSource = mock(InstantSource.class);**

**when(instantSource.instant())**

**.thenReturn(Instant.ofEpochMilli(1\_640\_033\_566\_000L))**

**.thenReturn(Instant.ofEpochMilli(1\_640\_033\_567\_750L));**

**Timer timer = new Timer(instantSource);**

**Duration duration = timer.measure(() -> {});**

**assertThat(duration, is(Duration.ofMillis(1\_750)));**

**}**

public class \_01\_Java\_Time\_InstanceSource\_Demo1 {

public static void main(String[] args) {

var timeTest = new TimerTest(InstantSource.system());

System.out.println(timeTest.getInstant());

}

}

class TimerTest {

InstantSource instantSource;

public TimerTest(InstantSource instantSource) {

this.instantSource = instantSource;

}

Instant getInstant() {

return instantSource.instant();

}

}

# JEP 406 - Pattern Matching for Switch Expression

Through **JDK Enhancement Proposal 406**, checking whether an object is an instance of a particular class can also be written as a switch statement (or expression). This is another step towards pattern matching by enhancing pattern matching for switch expressions and statements. It reduces the boilerplate necessary to define those expressions and improves the expressiveness of the language. The switch statement is enhanced to match expressions with multiple patterns. Each pattern has a specific operation, so complex data patterns can be expressed concisely and safely.

public class \_01\_Switch\_Exp\_InstanceOf\_Demo1 {  
  
 public static void main(String[] args) {  
 System.*out*.println(*formatString*("Java 17"));  
 System.*out*.println(*formatString*(17));  
 }  
  
 private static String formatString(Object obj){  
 return switch(obj){  
 case Integer i -> String.*format*("Int %d", i);  
 case String s -> String.*format*("String %s", s);  
 case Long l -> String.*format*("Long %d", l);  
 case Double d -> String.*format*("Double %f", d);  
 default -> obj.toString();  
 };  
 }  
}

# JEP 306 - Restore Always Strict Floating-Point Semantics

An almost unknown Java keyword is strictfp. It is used in class definitions to make floating-point operations within a class "strict". This means that they lead to predictable results on all architectures.

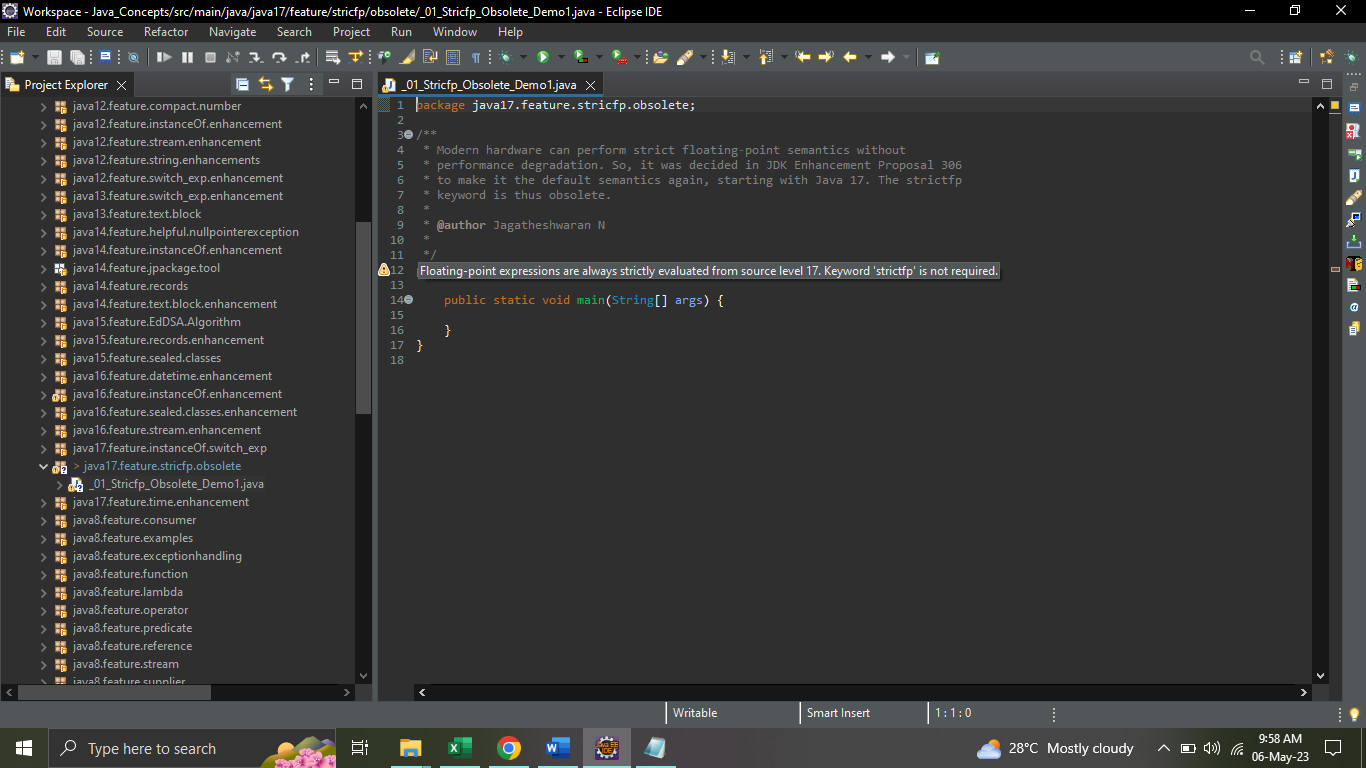
Strict floating-point semantics was the default behavior before Java 1.2 (i.e., more than 20 years ago).

Starting with Java 1.2, "standard floating-point semantics" was used by default, leading to slightly different results depending on the processor architecture. On the other hand, it was more performant, especially on the x87 floating-point coprocessor, which was widespread at that time, since it had to perform additional operations for the strict semantics.

Those who wanted to continue strict calculation from Java 1.2 had to indicate this by the strictfp keyword in the class definition.

Modern hardware can perform strict floating-point semantics without performance degradation. So, it was decided in JDK Enhancement Proposal 306 to make it the default semantics again, starting with Java 17.

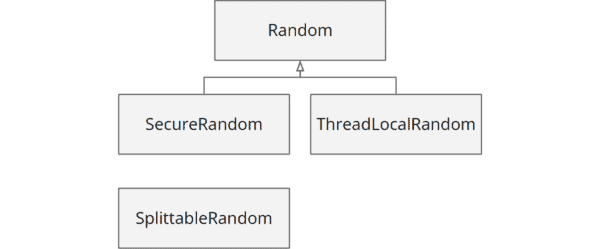
**The strictfp keyword is thus obsolete.** The usage leads to a compiler warning.



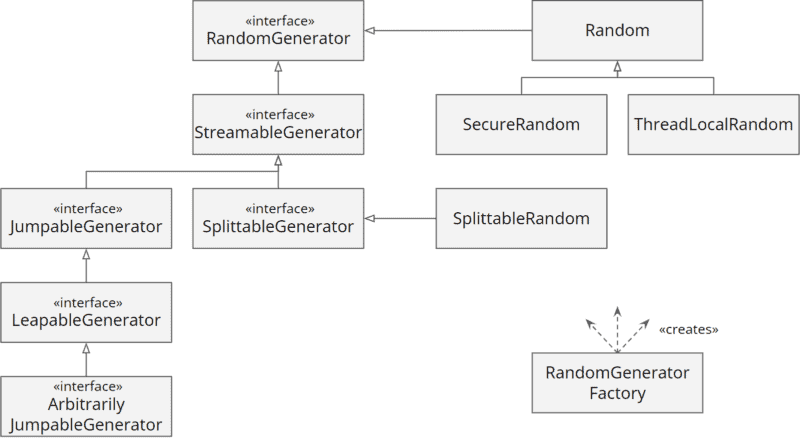
# JEP 356 - Enhanced Pseudo-Random Number Generators

Until now, it was cumbersome to exchange the random number-generating classes Random and SplittableRandom in an application (or even to replace them by other algorithms) although they offer a mostly matching set of methods (e.g. nextInt(), nextDouble(), and stream-generating methods like ints() and longs()).

The class hierarchy used to look like this,



Through **JDK Enhancement Proposal 356**, Java 17 introduced a framework of interfaces inheriting from each other for the existing algorithms and new algorithms so that the concrete algorithms are easily interchangeable in the future.



The methods common to all random number generators like nextInt() and nextDouble() are defined in RandomGenerator. So if you only need these methods, you should always use this interface in the future.

The framework includes three new types of random number generators:

**JumpableGenerator:** provides methods to skip a large number of random numbers (e.g., 264).

**LeapableGenerator:** provides methods to skip a very large number of random numbers (e.g., 2128).

**ArbitrarilyJumpableGenerator:** offers additional methods to skip an arbitrary number of random numbers.

In addition, duplicated code was eliminated from the existing classes, and code was extracted into non-public abstract classes (not visible in the class diagram) to make it reusable for future implementations of random number generators.

In the future, new random number generators can be added via the Service Provider Interface (SPI) and be instantiated via RandomGeneratorFactory.

Java 17 also refactored the legacy random classes like **java.util.Random, SplittableRandom** and **SecureRandom** to extend the new **RandomGenerator** interface.

public class \_01\_Enhanced\_Random\_Number\_Generator\_Demo1 {

public static void main(String[] args) {

// Legacy

// RandomGeneratorFactory.of("Random").create(40);

// Default L32X64MixRandom

// RandomGenerator generator = RandomGeneratorFactory.getDefault().create();

// Passing the same seed to random, and then calling it will give you the same

// set of numbers for example, seed = 999

RandomGenerator generator = RandomGeneratorFactory.*of*("Xoshiro256PlusPlus").create(999);

System.***out***.println(generator.getClass());

int counter = 0;

while (counter <= 10) {

int number = generator.nextInt(12);

counter++;

System.***out***.println(number);

}

// The below code generates all the Java 17 PRNG algorithms

RandomGeneratorFactory.*all*().map(factory -> factory.group() + " : " + factory.name()).sorted()

.forEach(System.***out***::println);

}

}

# JEP 415 - Context-Specific Deserialization Filters

Deserialization of objects poses a significant security risk. Malicious attackers can construct objects via the data stream to be deserialized, via which they can ultimately execute arbitrary code in arbitrary classes available on the classpath.

Java 9 introduced deserialization filters, i.e., the ability to specify which classes may (or may not) be deserialized.

Until now, there were two ways to define deserialization filters,

1. Per ObjectInputStream.setObjectInputFilter() for each deserialization separately.
2. System-wide via system property jdk.serialFilter or security property of the same name in the file conf/security/java.properties.

These variants are not satisfactory for complex applications, especially those with third-party libraries that also contain deserialization code. For example, deserialization in third-party code cannot be configured via ObjectInputStream.setObjectInputFilter() (unless you change the third-party source code), but only globally.

JDK Enhancement Proposal 415 makes it possible to set deserialization filters context-specifically, e.g., for a specific thread or based on the call stack for a particular class, module, or third-party library.

The configuration of the filters is not easy and is beyond the scope of this article. You can find details in the JEP link below.

<https://openjdk.org/jeps/415>

In Java, deserializing untrusted data is dangerous, read the OWASP – Deserialization of untrusted data and Brian Goetz – Towards Better Serialization.

**Java 9**, JEP 290 introduced serialization filtering to help prevent deserialization vulnerabilities.

public class \_01\_Context\_Specific\_Deserialization\_Demo1 {

public static void main(String[] args) throws IOException {

byte[] bytes = *convertObjectToStream*(new DeserializeExample());

InputStream inputStream = new ByteArrayInputStream(bytes);

ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);

ObjectInputFilter filter = ObjectInputFilter.Config

.*createFilter*("maxbytes=1024;java17.feature.context.specific.deserialization.\*;java.base/\*;!\*");

// .createFilter("maxbytes=1024;!java17.feature.context.specific.deserialization.\*;java.base/\*;!\*"); - REJECT Scenario

objectInputStream.setObjectInputFilter(filter);

try {

Object object = objectInputStream.readObject();

System.***out***.println("Read Object ==> " + object);

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

private static byte[] convertObjectToStream(Object object) {

ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

try (ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteArrayOutputStream)) {

objectOutputStream.writeObject(object);

return byteArrayOutputStream.toByteArray();

} catch (IOException ex) {

ex.printStackTrace();

}

throw new RuntimeException();

}

}

class DeserializeExample implements Serializable {

private static final long ***serialVersionUID*** = 1L;

public String toString() {

return "Deserialization Demo";

}

}

# References

<https://www.happycoders.eu/java/java-17-features/>

<https://www.baeldung.com/java-17-new-features>

<https://examples.javacodegeeks.com/java-17-new-features-tutorial/>

<https://mkyong.com/java/what-is-new-in-java-17/>

<https://www.geeksforgeeks.org/jdk-17-new-features-in-java-17/>