Inspections

JetGroovy offers many code inspections which can appear as warnings or errors in the IDE. Many of the inspections have quick fixes associated with them. This list is complete as of version 1.6.20679.

Assignment issues

Assignment replaceable with operator assignment		This inspection reports instances of assignment operations in Groovy which can be replaced by operator-assignment. Code using operator assignment may be clearer, and theoretically more performant.Use the check box below to ignore the conditional operators &&and ||. Replacing conditional operators with operator assignment modifies the semantics of the expression.	
Assignment to for-loop parameter		This inspection reports any instances of assignment a variable declared in a Groovy for statement in the body of that statement. It also reports any attempt to increment or decrement the variable. While occasionally intended, this construct can be extremely confusing, and is often the result of a typo.	
Assignment to method parameter		This inspection reports any instances of assignment to a variable declared as a Groovy method parameter. It also reports any attempt to increment or decrement the variable. While occasionally intended, this construct can be extremely confusing, and is often the result of a typo.	
Nested assignment		This inspection reports any instances of Groovy assignment expressions nested inside other expressions. While admirably terse, such expressions may be confusing, and violate the general design priniciple that a given construct should do precisely one thing.	
Result of assignment used		This inspection reports any Groovy assignment expressions nested inside other expressions, so as to use the assigned value immediately. While admirably terse, such expressions may be confusing, and violate the general design principle that a given construct should do precisely one thing.	
Silly assignment		This inspection reports any assignment of a Groovy variable to itself.	

Potentially confusing code constructs

Conditional expression		This inspection reports any instances of the ternary condition operator. Some coding standards prohibit the use of the condition operator, in favor of if-else statements.	
Double negation		This inspection reports any instances of double negation in Groovy code, like if (!!functionCall()).	
Statement with empty body		This inspection reports any instances of if,while, do or forstatements in Groovy code having empty bodies. While occasionally intended, this construction is confusing, and often the result of a typo.	
Negated conditional expression		This inspection reports any instances of Groovy conditional expressions whose conditions are negated. Flipping the order of the conditional expression branches will usually increase the clarity of such statements.	
Negated if condition expression		This inspection reports any instances of if statements which contain else branches and whose conditions are negated. Flipping the order of the if and elsebranches will usually increase the clarity of such statements.	
Nested conditional expression		This inspection reports any ternary conditional expressions in Groovy script file that are nested inside other conditional expressions. Such nested conditionals may be very confusing. "Elvis" expressions are counted as conditionals for purpose of this inspection.	
Nested switch statement		This inspection reports anyswitch statements in Groovy script file that are nested inside otherswitch statem ents. Such nested switch statements are confusing, and may result in unexpected behaviour.	
Octal integer		This inspection reports any instances of Groovy octal integer literals. Some coding standards prohibit the use of octal literals, as they may be easily confused with decimal literals.	
Overly complex arithmetic expression		This inspection reports reports any instances of Groovy arithmetic expressions with too many terms. Such expressions may be confusing and bug-prone.Use the field provided below to specify the maximum number of terms allowed in an arithmetic expression.	

Overly complex boolean expression		This inspection reports reports any instances of Groovy boolean expressions with too many terms. Such expressions may be confusing and bug-prone.Use the field provided below to specify the maximum number of terms allowed in an boolean expression.	
Pointless arithmetic expression		This inspection reports any instances of pointless arithmetic expressions in Groovy code. Such expressions include adding or subtracting zero, multiplying by zero or one, division by one, and shift by zero. Such expressions may be the result of automated refactorings not completely followed through to completion, and in any case are unlikely to be what the developer intended to do.	
Pointless boolean expression		This inspection reports any instances of pointless or pointlessly complicated boolean expressions in Groovy code. Such expressions include anding with true, oring with false, equality comparison with a boolean literal, or negation of a boolean literal. Such expressions may be the result of automated refactorings not completely followed through to completion, and in any case are unlikely to be what the developer intended to do.	
Result of increment or decrement used		This inspection reports any instances of Groovy increment or decrement expressions nested inside other expressions. While admirably terse, such expressions may be confusing, and violate the general design principle that a given construct should do precisely one thing.	

Control Flow

Break statement		This inspection reports any instances of break statements in a Groovy script, other than in switch statements.	
Conditional expression can be conditional call		This inspection reports instances of Groovy ternary conditional expressions which can be replaced by the conditional call (?.) operation.	
Conditional expression can be elvis		This inspection reports any uses of the ternary condition operator in Groovy which can be replaced by the simpler "elvis" operator.	
Conditional expression with identical branches		This inspection reports any instances of Groovy conditional expressions with identical "then" and "else" branches. Such expressions are almost certainly programmer error.	
Constant conditional expression		This inspection reports any instances of Groovy conditional expressions of the formtrue?result1:result2 or false?result1:result2. These expressions sometimes occur as the result of automatic refactorings, and may obviously be simplified.	
Constant if statement		This inspection reports any instances of if statements of the formif(true)... or if(false).... These statements sometimes occur due to automatic refactorings, and may obviously be simplified.	
Continue statement		This inspection reports any instances of continue statements in a Groovy script.	
Fallthrough in switch statement		This inspection reports any instances of 'fallthrough' in a Groovy switch statement. While occasionally useful, fallthrough is often unintended, and may lead to surpising bugs.	
If statement with identical branches		This inspection reports any instances of Groovy ifstatements with identical "then" and else branches. Such statements are almost certainly programmer error.	
If statement with too many branches		This inspection reports instances of Groovy if statements with too many branches. Such statements may be confusing, and are often the sign of inadequate levels of design abstraction.Use the field provided below to specify the maximum number of branches expected.	
Loop statement that doesn't loop		This inspection reports any instance of Groovy for orwhile statements whose bodies are guaranteed to execute at most once. Normally, this is an indication of a bug.	
'return' statement can be implicit		This inspection reports any return statements at the end of Groovy closures which can be made implicit. Groovy closures implicitly return the value of the last statement in them.	
Switch statement with no default case		This inspection reports any instances of Groovy switch statements that do not containdefault labels.	

Redundant conditional expression		This inspection reports any instances of Groovy ternary conditional operators of the form x?true:false or similar, which can be trivially simplified.

Redundant 'if' statement		This inspection reports instances of Groovy if statements which can be simplified to single assignment orreturn statements. For example:
		`if(foo())`
		`{`
		`return true;`
		`}`
		`else`
		`{`
		`return false;`
		`}`
		can be simplified to
		`return foo();`

Unnecessary 'continue' statement		This inspection reports on any unnecessary Groovy continue statements at the end of loops. These may be safely removed.
Unnecessary 'return' statement		This inspection reports on any unnecessary Groovy return statements at the end of constructors and methods returningvoid. These may be safely removed.

Probable bugs

Divide by zero		This inspection reports any instances of division by zero or remainder by zero in Groovy code.
Infinite loop statement		This inspection reports any instances of Groovy for, while, or do statements which can only exit by throwing an exception. While such statements may be correct, they are often a symptom of coding errors.
Infinite recursion		This inspection reports any instances of Groovy methods which must either recurse infinitely or throw an exception. Methods reported by this inspection can not return normally.
Non short-circuit boolean		This inspection reports any uses of the non-short-circuit forms of boolean 'and' and 'or' (&and |) in Groovy code. The non-short-circuit versions are occasionally useful, but their presence is often due to typos of the short-circuit forms (&&and ||), and may lead to subtle bugs.
Result of array allocation ignored		This inspection reports any instances of Groovy array allocation where the array allocated ignored. Such allocation expressions are legal Groovy, but are usually either inadvertant, or evidence of a very odd object initialization strategy.
Result of object allocation ignored		This inspection reports any instances of Groovy object allocation where the object allocated ignored. Such allocation expressions are legal Groovy, but are usually either inadvertant, or evidence of a very odd object initialization strategy.

Error handling

'continue' or 'break' inside 'finally' block		This inspection reports any instances of break or continue statements inside of finallyblocks in Groovy code. While occasionally intended, such statements are very confusing, may mask exceptions thrown, and tremendously complicate debugging.

Empty 'catch' block		This inspection reports empty catch blocks. While occasionally intended, this empty catch blocks can make debugging extremely difficult.	
Empty 'finally' block		This inspection reports instances of empty finally blocks in Groovy code. Empty finally blocks usually indicate coding errors.	
Empty 'try' block		This inspection reports any instances of empty try blocks in Groovy code. Empty finally blocks usually indicate coding errors.	
'return' inside 'finally' block		This inspection reports any instances of Groovy return statements inside of finallyblocks. While occasionally intended, such return statements may mask exceptions thrown, and tremendously complicate debugging.	
'throw' inside 'finally' block		This inspection reports any instances of Groovy throw statements inside of finallyblocks. While occasionally intended, such throw statements may mask exceptions thrown, and tremendously complicate debugging.	
Unused catch parameter		This inspection reports any catch parameters that are unused in their corresponding blocks. This inspection will not report any catch parameters named "ignore" or "ignored".	

GPath inspections

Getter call can be property access		This inspection reports any calls to "getter" methods in Groovy which can be replaced by the equivalent property access form.	
Call to List.get can be keyed access		This inspection reports any calls in Groovy code to java.util.List.get() methods. Such calls can be replaced by the shorter and clearer keyed access form.	
Call to List.set can be keyed access		This inspection reports any calls in Groovy code to java.util.List.set() methods. Such calls can be replaced by the shorter and clearer keyed access form.	
Call to Map.get can be keyed access		This inspection reports any calls in Groovy code to java.util.Map.get() methods. Such calls can be replaced by the shorter and clearer keyed access form.	
Call to Map.put can be keyed access		This inspection reports any calls in Groovy code to java.util.Map.put() methods. Such calls can be replaced by the shorter and clearer keyed access form.	
Setter call can be property access		This inspection reports any calls to "setter" methods in Groovy which can be replaced by the equivalent property access form.	

Method Metrics

Method with too many parameters		This inspection reports any instances of methods with too many parameters. Methods with too many parameters can be a good sign that refactoring is necessary. Methods whose signatures are inherited from library classes are ignored by this inspection.Use the field provided below to specify the maximum acceptable number of parameters a method might have.	
Method with more than three negations		This inspection reports Groovy methods with three or more negation operations (! or !=). Such methods may be unnecessarily confusing.	
Method with multiple return points		This inspection reports any instances of Groovy methods with too many return points. Methods with too many return points may be confusing, and hard to refactor. Use the field provided below to specify the maximum acceptable number of return points a method might have.	
Overly complex method		This inspection reports any instances of Groovy methods that have too high a cyclomatic complexity. Cyclomatic complexity is basically a measurement of the number of branching points in a method. Methods with too high a cyclomatic complexity may be confusing and difficult to test.Use the field provided below to specify the maximum acceptable cyclomatic complexity a method might have.	
Overly long method		This inspection reports any instances of Groovy methods that are too long. Methods that are too long may be confusing, and are a good sign that refactoring is necessary. Use the field provided below to specify the maximum acceptable number of non-comment source statements a method might have.	

Overly nested method		This inspection reports any instances of Groovy methods whose bodies are too deeply nested. Methods with too much statement nesting may be confusing, and are a good sign that refactoring may be necessary.Use the field provided below to specify the maximum acceptable nesting depth a method might have.	

Threading issues

Access to static field locked on instance data		This inspection reports on any access to a static field of any non-threadsafe type specified below, which is accessed from an instance field or a non-synchronized block. It is possible that the static field is accessed from multiple threads, which can lead to unspecified side effects.	
Busy wait		This inspection reports instances of calls to java.lang.Thread.sleep() that occur inside loops. Such calls are indicative of "busy-waiting". Busy-waiting is often inefficient, and may result in unexpected deadlocks as busy-waiting threads do not release locked resources.	
Double-checked locking		This inspection reports any instances of the double-checked locking construct in Groovy code. For a discussion of double-checked locking and why it is unsafe, seehttp://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.htmlUse the checkbox below to ignore double-checked locking on volatile fields. Using a volatile field for double-checked locking works correctly on Java 5 virtual machines, but probably does not have any performance advantages over plain full synchronization of the accessor method.	
Empty 'synchronized' block		This inspection reports any instances of synchronized statements in Groovy code having empty bodies. While theoretically this may be the semantics intended, this construction is confusing, and often the result of a typo.	
Nested 'synchronized' statement		This inspection reports all instances of nested Groovy synchronized statements. Nested synchronized statements are either useless (if the lock objects are identical) or prone to deadlock.	
'notify()' or 'notifyAll()' while not synced		This inspection reports on any Groovy call to notify() not made inside a corresponding synchronized statement or synchronized method. Calling notify() on an object without holding a lock on that object will result in an IllegalMonitorStateException being thrown. Such a construct is not necessarily an error, as the necessary lock may be acquired before the containing method is called, but it's worth looking at.	
Non-private field accessed in synchronized context		This inspection reports any instances of non-final, non-private fields which are accessed in a synchronized context in Groovy code. A non-private field cannot be guaranteed to always be accessed in a synchronized manner, and such "partially synchronized" access may result in unexpectedly inconsistent data structures. Accesses in constructors an initializers are ignored for purposes of this inspection.	
Synchronization on non-final field		This inspection reports instances of Groovy synchronized statements where the lock expression is a non-final field. Such statements are unlikely to have useful semantics, as different threads may be locking on different objects even when operating on the same object.	
Synchronization on 'this'		This inspection reports any instances of synchronization in Groovy code which use this as their lock expression. Constructs reported include synchronizedblocks which lock this, and calls to wait()notify() or notifyAll() which target wait(). Such constructs, like synchronized methods, make it hard to track just who is locking on a given object, and make possible "denial of service" attacks on objects. As an alternative, consider locking on a private instance variable, access to which can be completely controlled.	
Synchronization on variable initialized with literal		This inspection reports any Groovy synchronized block which locks on an object which is initilized with a literal. String literals are interned and Number literals can be allocated from a cache. Because of this, it is possible that some other part of the system which uses an object initialized with the same literal, is actually holding a reference to the exact same object. This can create unexpected dead-lock situations, if the string was thought to be private.	
Synchronized method		This inspection reports any use of the synchronized modifier on Groovy methods. Some coding standards prohibit the use of the synchronized modifier, in favor of synchronized statements.	

Call to System.runFinalizersOnExit()		This inspection reports any calls to System.runFinalizersOnExity() from Groovy code. This call is one of the most dangerous in the Java language. It is inherently non-thread-safe, may result in data corruption, deadlock, and may effect parts of the program far removed from it's call point. It is deprecated, and it's use strongly discouraged.	
Call to Thread.stop(), Thread.suspend(), or Thread.resume()		This inspection reports any calls to Thread.stop(), Thread.suspend(), or Thread.resume() from Groovy code. These calls are inherently prone to data corruption and deadlock, and their use is strongly discouraged.	
Unconditional 'wait' call		This inspection reports any instances of .wait() being called unconditionally within a synchronized context in Groovy code. Normally, .wait() is used to block a thread until some condition is true. If .wait() is called unconditionally, that often indicates that the condition was checked before a lock was acquired. In that case a data race may occur, with the condition becoming true between the time it was checked and the time the lock was acquired. While constructs found by this inspection are not necessarily incorrect, they are certainly worth examining.	
Unsynchronized method overrides synchronized method		This inspection reports any instances of non-synchronized Groovy methods overriding sy nchronized methods.	
'wait()' not in loop		This inspection reports on any call to wait() from Groovy code not made inside a loop. w ait() is normally used to suspend a thread until a condition is true, and that condition should be checked after the wait()returns. A loop is the clearest way to achieve this.	
'wait()' while not synced		This inspection reports any call to wait() not made inside a corresponding synchronized statement or synchronized method within Groovy code. Calling wait() on an object without holding a lock on that object will result in an IllegalMonitorStateException being thrown. Such a construct is not necessarily an error, as the necessary lock may be acquired before the containing method is called, but its worth looking at.	
While loop spins on field		This inspection reports on any instances of Groovy while loops which spin on the value of a non-volatile field, waiting for it to be changed by another thread. In addition to being potentially extremely CPU intensive when little work is done inside the loop, such loops are likely have different semantics than intended, as the Java Memory Model allows such field accesses to be hoisted out of the loop, causing the loop to never complete even if another thread does change the field's value.	

Validity issues

Duplicate switch case		This inspection reports duplicated case expressions in Groovy switch statements.	
Method with inconsistent returns		This inspection reports any Groovy methods with both value-returning and non-value-returning return statements. While theoretically valid, such inconsistency is almost certainly due to coding error.	
Unreachable Statement		This inspection reports any statements in Groovy script file that are unreachable.	