

Simple Groovy datatypes

Java's type system: primitives and references

Java distinguishes between primitive types (such as `boolean` , `short` , `int` , `double` , `float` , `char` , and `byte`) and reference types (such as `Object` and `String`).

You cannot call methods on values of primitive types, and you cannot use them where Java expects objects of type `java.lang.Object` . For each primitive type, Java has a wrapper type a reference type that stores a value of the primitive type in an object.

The wrapper for `int` , for example, is `java.lang.Integer` . Conversely, operators such as `*` in `3 * 2` or `a * b` aren't supported for arbitrary 1 reference types in Java, but only for primitive types (with the notable exception of `+` , which is also supported for strings).

```
(60 * 60 * 24 * 365).toString(); // invalid Java
int secondsPerYear = 60 * 60 * 24 * 365;
secondsPerYear.toString(); // invalid Java
new Integer(secondsPerYear).toString();
assert "abc" - "a" == "bc" // invalid Java
```

Groovy's answer: everything's an object

To make Groovy fully object-oriented, and because at the JVM level Java doesn't support object-oriented operations such as method calls on primitive types, the Groovy designers decided to do away with primitive types. When Groovy needs to store values that would have used Java's primitive types, Groovy uses the wrapper classes already provided by the Java platform.

Table 1. Java's primitive datatypes and their wrappers

Primitive type	Wrapper type	Description
<code>byte</code>	<code>java.lang.Byte</code>	8-bit signed integer
<code>short</code>	<code>java.lang.Short</code>	16-bit signed integer
<code>int</code>	<code>java.lang.Integer</code>	32-bit signed integer
<code>long</code>	<code>java.lang.Long</code>	64-bit signed integer
<code>float</code>	<code>java.lang.Float</code>	Single-precision (32-bit) floating-point value
<code>double</code>	<code>java.lang.Double</code>	Double-precision (64-bit) floating-point value
<code>char</code>	<code>java.lang.Character</code>	16-bit Unicode character
<code>boolean</code>	<code>java.lang.Boolean</code>	Boolean value (<code>true</code> or <code>false</code>)

Any time you see what looks like a primitive literal value (the number 5 , for example, or the Boolean value true) in Groovy source code, that's a reference to an instance of the appropriate wrapper class.

Table 2. Numeric literals in Groovy

Type	Example literals
java.lang.Integer	15, 0x1234ffff, 0b00110011, 100_000_000
java.lang.Long	100L, 200l
java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	123g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

Interoperating with Java: automatic boxing and unboxing

Groovy performs these operations automatically for you where necessary. This is primarily the case when you call a Java method from Groovy. This automatic boxing and unboxing is known as autoboxing.

```
assert 'ABCDE'.indexOf(67) == 2
```

From Groovy's point of view, you're passing an Integer containing the value 67 (the Unicode value for the letter C), even though the method expects a parameter of primitive type int .

Groovy takes care of the unboxing. The method returns a primitive type int that's boxed into an Integer as soon as it enters the world of Groovy. That way, you can compare it to the Integer with value 2 back in the Groovy script.

No intermediate unboxing

If in 1+1 both numbers are objects of type Integer , you may be wondering whether those Integer objects are unboxed to execute the plus operation on primitive types.

The answer is no: Groovy is more object-oriented than Java. It executes this expression as 1.plus(1) , calling the plus() method of the first Integer object, and passing 2 the second Integer object as an argument. The method call returns an Integer object of value 2 .

This is a powerful model. Calling methods on objects is what object-oriented languages should do. It opens the door for applying the full range of object-oriented capabilities to those operators

Optional typing

Assigning types

Groovy offers the choice of explicitly specifying variable types just as you do in Java, And also offer the choice of implicit typing which is done by using The `def` keyword.

```
def a = 1      //java.lang.Integer
def b = 1.0f   //java.lang.Float
int c = 1      //java.lang.Integer
float d = 1    //java.lang.Float
Integer e = 1  //java.lang.Integer
String f = '1' //java.lang.String
```

Regardless of whether a variable's type is explicitly declared, the system is type safe.

Unlike untyped languages, Groovy doesn't allow you to treat an object of one type as an instance of a different type without a well-defined conversion being available.

You could never assign a `java.util.Date` to a reference of type `java.lang.Number`, in the hope that you'd end up with an object that you could use for calculation. That sort of behavior would be dangerous, which is why

Groovy doesn't allow it any more than Java does.

Dynamic Groovy is type safe

Static is often associated with the appearance of type markers in the code. For instance, code such as

```
String greeting = readFromConsole()
```

is often considered static because of the `String` type marker, while unmarked code like

```
def greeting = readFromConsole()
```

is usually deemed dynamic.

By default, Groovy is very much a dynamic language. You can safely leave out type markers (and also type casts) in most scenarios and know that Groovy will do the appropriate runtime checks to ensure type safety when required.

Because type markers are optional in Groovy, that concept is often called **optional typing**.

Groovy uses type markers to enforce the Java type system at runtime. But it only does so at runtime, where Java does so with a mixture of compile-time and runtime checks.

This explains why the Groovy compiler 6 takes no issue with

```
Integer myInt = new Object() // ClassCastException At runtime
println myInt
```

In fact, this is the same effect you see if you write a typecast on the right-hand side of the assignment in Java. Consider this Java code:

```
Integer myInt = (Integer) returnsObject(); // Java!
```

Groovy types aren't dynamic, they never change

The word “dynamic” doesn't mean that the type of a reference, once declared, can ever change. Once you've declared Integer `myInt`, you cannot execute `myInt = new Object()`. This will throw a `GroovyCastException`.

You can only assign a value, which Groovy can cast to an Integer.

Casting

Groovy actually applies convenience logic when casting, which is mainly concerned with casting primitive types to their wrapper classes and vice versa, arrays to lists, characters to integers, Java's type widening for numeric types, applying the “Groovy truth” for casts to boolean, calling `toString()` for casts to string, and so on. The exhaustive list can be looked up in `DefaultTypeTransformation.castToType`.

```
import java.awt.*

Point topLeft = new Point(0, 0) // classic

Point botRight = [100, 100] // List cast

Point center = [x:50, y:50] // Map cast

assert botRight instanceof Point
assert center instanceof Point

def rect = new Rectangle()
rect.location = [0, 0] // Point
rect.size = [width:100, height:100] // Dimension
```

Implicit runtime casting can lead to very readable code, especially in cases like property assignments where Groovy knows that `rect.size` is of type `java.awt.Dimension` and can cast your list or map of constructor arguments onto that.

You don't have to worry about it: Groovy infers the type for you.

Optional typing

Groovy is an “optionally” typed language, and that distinction is an important one to grasp when understanding the fundamentals of the language. Groovy’s nearest ancestor, Java, is said to be a “strongly” typed language, whereby the compiler knows all of the types for every variable and can understand and honor contracts at compile time. This means that method calls are able to be determined at compile time, and therefore take the onus of their resolution off of the runtime system.

```
class UserHibernateDAO {
    def sessionFactory

    def getByFirstName(String name) {
        List<User> users = sessionFactory.createQuery("select * from user where
firstName = :name")
                                .setParameter("name", name)
                                .list()
        users.size() == 1 ? users[0] : users
    }
}
```

Duck typing

If it walks like a duck and quacks like a duck, it must be a duck.

Duck typing

implies that as long as an object has a certain set of method signatures, it’s interchangeable with any other object that has the same set of methods, regardless of whether the two have a related inheritance hierarchy.

NOTE

Experienced Groovy programmers tend to follow this rule of thumb: as soon as you think about the type of a reference, declare it; if you’re thinking of it as “just an object,” leave the type out.

Overriding operators

When a language bases its operators on method calls and allows these methods to be overridden, the approach is called operator overriding.

Table 3. Method-based operators

Operator	Name
+	a.plus(b)
-	a.minus(b)
*	a.multiply(b)
/	a.div(b)
%	a.mod(b)
<<	a.leftShift(b)
>>	a.rightShift(b)
>>>	a.rightShiftUnsigned(b)
**	a.power(b)
	a.or(b)
&	a.and(b)
^	a.xor(b)
++	a.next()
—	a.previous()
~a	a.bitwiseNegate()
-a	a.negative()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a in b	b.isCase(a)
+a	a.positive()
as	a.asType(b)
a()	a.call()
switch(a){case b:}	b.isCase(a)
a in	b.isCase(a)
a ==	if (a implements Comparable) { a.compareTo(b) == 0 } else { a.equals(b) }
a != b	!(a==b)
a <⇒	a.compareTo(b)
a > b	a.compareTo(b)>0
a < b	a.compareTo(b)<0
a ⇐ b	a.compareTo(b) ⇐ 0
a ≥ b	a.compareTo(b)≥0
a as b	a.asType(b)

Example 1. Overridden operators

```
import groovy.transform.Immutable;

@Immutable
class Money {
    int value

    Money plus(Money other) {
        new Money(this.value + other.value) ①
    }
}

def tenDollar = new Money(10)
def fiveDollar = new Money(5)

assert (tenDollar + fiveDollar).value == 15
```

① Implicit return statement

NOTE

Our plus operation on the Money class returns Money objects in both cases. We describe this by saying that Money's plus operation is closed under its type. Whatever operation you perform on an instance of Money, you end up with another instance of Money.

coercion

```
1 + 1.0
```

What is the return type? the issue is more general.

One of the two arguments needs to be promoted to the more general type. This is called **coercion**.

When implementing operators, there are three main issues to consider as part of coercion:

Supported argument types

You need to decide which argument types and values will be allowed. If an operator must take a potentially inappropriate type, throw an `IllegalArgumentException` where necessary.

Promoting more specific arguments

If the argument type is a more specific one than your own type, promote it to your type and return an object of your type.

`Integer` is more specific than `BigDecimal`: every `Integer` value can be expressed as a `BigDecimal`, but the reverse isn't true. So for the `BigDecimal.plus(Integer)` operator, you'd consider promoting

the `Integer` to `BigDecimal` , performing the addition, and then returning another `BigDecimal` —even if the result could accurately be expressed as an `Integer` .

Handling more general arguments with double dispatch

If the argument type is more general, call its operator method with yourself as an argument. Let it promote you. This is also called double dispatch, and it helps to avoid duplicated , asymmetric, possibly inconsistent code.

NOTE	Groovy's general strategy of coercion is to return the most general type.
-------------	---

Strings

Table 4. String literal styles available in Groovy

Start/end characters	Example	Placeholder resolved?	Backslash escapes?
Single quote	'hello Dierk'	No	Yes
Double quote	"hello \$name"	Yes	Yes
Triple single quote	''' ===== Total: \$0.02 ===== '''	No	Yes
Triple double quote	"""first \$line second \$line third \$line"""	Yes	Yes
Forward slash	/x(d*)y/	Yes	Occasionally
Dollar slash	\$/x(d*)y/\$	Yes	Occasionally

The single-quoted

never pays any attention to placeholders. This is closely equivalent to Java string literals.

The double-quoted

is the equivalent of the single-quoted form, except that if the text contains unescaped dollar signs, the dollar sign introduces a placeholder, and the string will be treated as a GS tring instead of a plain string. GStrings are covered in more detail in the next section.

The triple-quoted (or multiline string literal)

allows the literal to span sev- eral lines. New lines are always treated as `\n` regardless of the platform, but all other whitespace is preserved as it appears in the text file. Multiline string literals may also be GS trings, depending on whether single quotes or double quotes are used. Multiline string literals act similar to Ruby or Perl.

The slashy

is also multiline but allows strings with back- slashes to be specified simply without having to escape all the backslashes. This is particularly useful with regular expressions, as you'll see later. There are only a few exceptions and limitations. Slashes are escaped with a backslash. A backslash can't appear as the last character of a slashy string. Dollar symbols that could

introduce a placeholder but aren't meant to also need to be escaped. If you want to create a string with a backslash followed by a u, the backslash needs to be escaped so as not to be interpreted as a Unicode character, which happens in the earliest stages of parsing.

The dollar slashy

allows strings with backslashes to be specified without having to escape all the backslashes. Only Unicode characters are escaped with a backslash. Dollar signs and slashes are escaped with a dollar sign. The other restrictions on backslashes you saw for normal slashy strings don't apply.

Table 5. Escaped characters as known to Groovy

Escaped special character	Meaning
<code>\b</code>	Backspace
<code>\t</code>	Tab
<code>\r</code>	Carriage return
<code>\n</code>	Linefeed
<code>\f</code>	Form feed
<code>\\</code>	Backslash
<code>\\$</code>	Dollar sign
<code>\uabcd</code>	Unicode character u + abcd (where a, b, c, and d are hex digits)
<code>\abc</code>	Unicode character u + abc (where a, b, and c are octal digits, and b and c are optional)
<code>\'</code>	Single quote
<code>\"</code>	Double quote

```
char a = 'x'
Character b = 'x'

'x' as char
'x'.toCharacter()
```

GStrings

GStrings are like strings with additional capabilities. They're literally declared in double quotes. What makes a double-quoted string literally a GString is the appearance of placeholders.

Placeholders may appear in a full `${expression}` syntax or an abbreviated `$reference` syntax.

```

import java.time.LocalDateTime

def name = "Jone"
def meetingName = "Simple"
def template = "Dear $name, $meetingName meeting starts at
${LocalDateTime.now().hour}pm tonight."
assert template == 'Dear Jone, Simple meeting starts at 14pm tonight.'

TimeZone.default = TimeZone.getTimeZone('GMT')
def date = new Date(0)
def dateMap = [y:date[YEAR]-1900
               , m:date[MONTH]
               , d:date[DAY_OF_MONTH]]

def out = "Year $dateMap.y Month $dateMap.m Day $dateMap.d"
assert out == 'Year 70 Month 0 Day 1'
Extended
def timeZone = TimeZone.getTimeZone('GMT')
def format = 'd MMM YYYY HH:mm:ss z'
out = "Date is ${date.format(format, timeZone)} !"
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'

def sql = """
SELECT FROM MyTable
WHERE Year = $dateMap.y
"""

assert sql == """
SELECT FROM MyTable
WHERE Year = 70
"""

out = "my 0.02\$"
assert out == 'my 0.02$'

```

Although GS trings behave like `java.lang.String` objects for all operations that a programmer is usually concerned with, they're implemented differently to capture the fixed and dynamic parts (the so-called values) separately. This is revealed by the following code:

```

def me = 'Tarzan'
def you = 'Jane'
def line = "me $me - you $you"

assert line == 'me Tarzan - you Jane'
assert line instanceof GString

assert line.strings[0] == 'me '
assert line.strings[1] == ' - you '

assert line.values[0] == 'Tarzan'
assert line.values[1] == 'Jane' 14

```

Placeholder evaluation time

Each placeholder inside a GString is evaluated at declaration time and the result is stored in the GString object.

By the time the GString value is converted into a `java.lang.String` (by calling its `toString` method or casting it to a string), each value gets written 14 to the string. Because the logic of how to write a value can be elaborate for certain types (most notably closures), this behavior can be used in advanced ways that make the evaluation of such placeholders appear to be lazy.

```

// Eager
def value=1
def gstring="$value"
value=2
assert gstring=="1"

//Lazy
def value=1
def gstring="${-> value}"
value=2
assert gstring=="2"

```

Extra Groovy Capability in String

A miscellany of string operations

```
String greeting = 'Hello Groovy!'

assert greeting.startsWith('Hello')

assert greeting.getAt(0) == 'H'

assert greeting[0] == 'H'

assert greeting.indexOf('Groovy') >= 0

assert greeting.contains('Groovy')

assert greeting[6..11] == 'Groovy'

assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'

assert greeting.count('o') == 3

assert 'x'.padLeft(3) == ' x'

assert 'x'.padRight(3, '_') == 'x__'

assert 'x'.center(3) == ' x '

assert 'x' * 3 == 'xxx'

def greeting = 'Hello'

greeting <= ' Groovy'
assert greeting instanceof java.lang.StringBuffer

greeting << '!'
assert greeting.toString() == 'Hello Groovy!'

greeting[1..4] = 'i'
assert greeting.toString() == 'Hi Groovy!'
```

NOTE

Although the expression `stringRef << string` returns a `StringBuffer`, note that `StringBuffer` isn't automatically assigned to the `stringRef`. When used on a `String`, it needs explicit assignment; on `StringBuffer` it doesn't. With a `StringBuffer`, the data in the existing object is changed with a `String` you can't change the existing data, so you have to return a new object instead. You might also note that a greeting was explicitly typed. It's effectively of type `Object` and can reference both `String` and `StringBuffer` values.

Regular expressions

Regular expressions are prominent in scripting languages and have also been available in the Java library since JDK 1.4.

Groovy relies on Java's `regex` (regular expression) support and adds three operators for convenience:

- The regex find operator, `=~`
- The regex match operator, `==~`
- The regex pattern operator, `~string`

```
import java.util.regex.Matcher
import java.util.regex.Pattern

def pattern = ~/\d+/
assert pattern instanceof Pattern

def matcher = 123 =~/\d+/
assert matcher instanceof Matcher

def result = 123==~/\d+/
assert result instanceof Boolean

assert result == matcher.matches()
```

Regular expression GStrings

```
def reference = "hello"
assert reference == /$reference/
```

TIP

Sometimes the slashy syntax interferes with other valid Groovy expressions such as line comments or numerical expressions with multiple slashes for division. When in doubt, put parentheses around your pattern like `(/pattern/)`. Parentheses force the parser to interpret the content as an expression.

Table 6. Regular expression symbols

Symbol	Meaning
.	Any character
^	Start of line (or start of document, when in single-line mode)
\$	End of line (or end of document, when in single-line mode)
\d	Digit character
\D	Any character except digits
\s	Whitespace character
\S	Any character except whitespace
\w	Word character
\W	Any character except word characters
\b	Word boundary
()	Grouping
(x y)	x or y , as in (Groovy,Java,Ruby)
\1	Backmatch to group one; for example, find doubled characters with (.)\1
x *	Zero or more occurrences of x
x	One or more occurrences of x
x ?	Zero or one occurrence of x
x { m , n }	At least m and at most n occurrences of x
x { m }	Exactly m occurrences of x
[a-f]	Character class containing the characters a , b , c , d , e , f
[^a]	Character class containing any character except a
(?is:x)	Switches mode when evaluating x ; i turns on ignoreCase , s means single- line mode

TIP

Symbols tend to have the same first letter as what they represent; for example, digit, space, word, and boundary. Uppercase symbols define the complement; think of them as a warning sign for no.

```

def twister = 'she sells sea shells at the sea shore of seychelles'
// twister must contain a substring of size 3
// that starts with s and ends with a
assert twister =~ /s.a/

def finder = (twister =~ /s.a/)
assert finder instanceof java.util.regex.Matcher

// twister must contain only words delimited by single spaces
assert twister ==~ /(\w+ \w+)* /

def WORD = /\w+/
matches = (twister ==~ /($WORD $WORD)* /)
assert matches instanceof java.lang.Boolean
assert !(twister ==~ /s.e /)

def wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x x'

def words = twister.split(/ /)
assert words.size() == 10
assert words[0] == 'she'

```

TIP

To remember the difference between the `=~` find operator and the `==~` match operator (it looks like a burning match), recall that match is more restrictive, because the pattern needs to cover the whole string. The demanded coverage is “longer” just like the operator itself.

What is a match?

A match is the occurrence of a regular expression pattern in a string.

It's therefore a string: a substring of the original string. When the pattern contains groupings like in `/begin.(*?)end/`, you need to know more information: not just the string matching the whole pattern, but also what part of that string matched each group.

Therefore, the match becomes a list of strings, containing the whole match at position 0 with group matches being available as `match[n]` where `n` is group number `n`. Groups are numbered by the sequence of their opening parentheses.

Working on each match of a pattern

```
def myFairStringy = 'The rain in Spain stays mainly in the plain!'
// words that end with 'ain': \b\w*ain\b
def wordEnding = /\w*ain/
def rhyme = /\b${wordEnding}\b/
def found = ''
myFairStringy.eachMatch(rhyme) { match ->
found += match + ' '
}
assert found == 'rain Spain plain '
found = ''
(myFairStringy =~ rhyme).each { match ->
found += match + ' '
}
assert found == 'rain Spain plain '

def cloze = myFairStringy.replaceAll(rhyme){ it-'ain'+'___' }
assert cloze == 'The r___ in Sp___ stays mainly in the pl___!'
```

The GDK enhances the `Matcher` class with simplified array-like access to this information.

In Groovy, you can think about a matcher as if it was a list of all its matches.

matches all nonwhitespace characters

```
def matcher = 'a b c' =~ /\S/
assert matcher[0] == 'a'
assert matcher[1..2] == ['b','c']
assert matcher.size() == 3
```

This use case comes with an interesting variant that uses Groovy's parallel assignment feature that allows you to directly assign each match to its own reference.

parallel assignment

```
def (a,b,c) = 'a b c' =~ /\S/
assert a == 'a'
assert b == 'b'
assert c == 'c'
```


Example 2. groupings in the match.

If the pattern contains parentheses to define groups, then the result of asking for a particular match is an array of strings rather than a single one: the same behavior as we mentioned for `eachMatch`. Again, the first result (at index 0) is the match for the whole pattern. Consider this example, where each match finds pairs of strings that are separated by a colon. For later processing, the match is split into two groups, for the left and the right string:

```
def matcher = 'a:1 b:2 c:3' =~ /(\S+):(\S+)/
assert matcher.hasGroup()
assert matcher[0] == ['a:1', 'a', '1'] // 1st match
assert matcher[1][2] == '2' // 2nd match, 2nd group
```

This also applies to the matcher's `each` method

```
def matcher = 'a:1 b:2 c:3' =~ /(\S+):(\S+)/
matcher.each { full, key, value ->
    assert full.size() == 3
    assert key.size() == 1 // a,b,c
    assert value.size() == 1 // 1,2,3
}
```

IMPORTANT

Groovy internally stores the most recently used matcher (per thread). It can be retrieved with the static property `Matcher.lastMatcher`.

You can also set the index property of a matcher to make it look at the respective match with `matcher.index = x`. Both can be useful in some exotic corner cases.

Patterns and performance

The rationale behind this construction is that patterns are internally backed by a finite-state machine that does all the high-performance magic.

This machine is compiled when the pattern object is created.

The more complicated the pattern, the longer the creation takes. In contrast, the matching process as performed by the machine is extremely fast.

```
def twister = 'she sells sea shells at the sea shore of seychelles'
// some more complicated regex:
// word that starts and ends with same letter
def regex = /\b(\w)\w*\1\b/
def many = 100 * 1000

start = System.nanoTime()
many.times{
    twister =~ regex
}
timeImplicit = System.nanoTime() - start
start = System.nanoTime()
pattern = ~regex
many.times{
    pattern.matcher(twister)
}
timePredef = System.nanoTime() - start
assert timeImplicit > timePredef * 2
```

NOTE

To find words that start and end with the same character, the `\1` backmatch is used to refer to that character.

Patterns for classification

The Pattern object, as returned from the pattern operator, implements an `isCase(String)` method that's equivalent to a full match of that pattern with the string. This classification method is a prerequisite for using patterns conveniently with the `in` operator, the `grep` method, and `in` switch cases.

```
def fourLetters = ~/\w{4}/

assert fourLetters.isCase('work')

assert 'love' in fourLetters

switch('beer'){
    case fourLetters: assert true; break
    default: assert false
}

beasts = ['bear', 'wolf', 'tiger', 'regex']
assert beasts.grep(fourLetters) == ['bear', 'wolf']
```

TIP

Classifications read nicely with `in` , `switch` , and `grep` . It's rare to call `classifier.isCase(candidate)` directly, but when you see such a call, it's easiest to read it from right to left: “candidate is a case of classifier.”

Numbers

Table 7. Numerical coercion

+ - *	B	S	I	C	L	BI	BD	F	D
Byte	I	I	I	I	L	BI	BD	D	D
Short	I	I	I	I	L	BI	BD	D	D
Integer	I	I	I	I	L	BI	BD	D	D
Character	I	I	I	I	L	BI	BD	D	D
Long	L	L	L	L	L	BI	BD	D	D
BigInteger	BI	BI	BI	BI	BI	BI	BD	D	D
BigDecimal	BD	BD	BD	BD	BD	BD	BD	D	D
Float	D	D	D	D	D	D	D	D	D
Double	D	D	D	D	D	D	D	D	D

GDK methods for numbers

```
assert 1 == (-1).abs()
assert 2 == 2.5.toInteger() // conversion
assert 2 == 2.5 as Integer // enforced coercion
assert 2 == (int) 2.5 // cast
assert 3 == 2.5f.round()
assert 3.142 == Math.PI.round(3)
assert 4 == 4.5f.trunc()
assert 2.718 == Math.E.trunc(3)
assert '2.718'.isNumber() // String methods
assert 5 == '5'.toInteger()
assert 5 == '5' as Integer
assert 53 == (int) '5' // gotcha!
assert '6 times' == 6 + ' times' // Number + String
```

WARNING

Don't cast strings to numbers! In Groovy, you can cast a string of length 1 directly to a char . But char and int are essentially the same thing on the Java platform. This leads to the gotcha where '5' is cast to its Unicode value 53 . Instead, use the type conversion methods.

```
def store = ''
10.times{
  store += 'x'
}
assert store == 'xxxxxxxxxx'

store = ''
1.upto(5) { number ->
  store += number
}
assert store == '12345'

store = ''
2.downto(-2) { number ->
  store += number + ' '
}
assert store == '2 1 0 -1 -2 '

store = ''
0.step(0.5, 0.1){ number ->
  store += number + ' '
}
assert store == '0 0.1 0.2 0.3 0.4 '
```