classes are always objects

Java makes a firm distinction between type names and variable names. With a few ambiguous exceptions (such as `classOrRef.member`), each Java syntax operates on entities of just one type of namespace.

This means that class-oriented syntaxes have evolved a set of reflective analogues: 'new' and 'instanceof' operate on classes, while the methods 'newInstance'' and 'isInstance' operate on objects. Moving back and forth between type-oriented syntaxes and their reflective analogus is cumbersome and error-prone. (It is even more complex with Java Generic Types.)

Groovy simplifies these matters, for most common usages, by making reflective class access be the main usage, instead of an arcane back door.

Advantages:

- Just one set of foundation syntaxes (i.e., method calls) to specify and implement.
- Just one syntax to learn, if you don't care about Java compatibility.
- Just one kind of cut-point (methods) to customize on, for API builders and language extenders.
- Special Java syntaxes ('new', 'instanceof') can be viewed as mere sugar over methods.
- Easy to use Java reflective APIs.
- _(More here?)

Disadvantages:

- Not the same as Java (though lexically simpler).
- More namespaces potentially affecting most identifiers, hence more disambiguation.
- _(More here?)

Object allocation, viewed as a method on a class, is discussed under the heading of object allocation.

Here is a sketch of class-oriented syntaxes and the underlying method-based semantics:

Syntax	Semantics	Comment
x instanceof C	C.isInstance(x)	Java reflection
(C)x	C.cast(x)	Java reflection (1.5)
new C()	C.newInstance()	see object allocation
new C(a,b)	C.getConstructor(...).newInstance(a,b)	hairy reflection API
make C(a,b)	C.make(a,b)	maybe a syntax extension?
C cvar = x	let cvar = C.cast(y)	(advisory type checking)
cvar = y	cvar = C.cast(y)	(advisory type checking)

Of course, a Groovy compiler is free to constant-fold method-based expressions that are compile-time constants, leading to the same bytecodes as a Java compiler would produce.

About Namspaces

Java has a many namespaces (class, method, field, package, label, etc.) with some syntaxes makiing ambiguous references to two or more namespaces. (See naming ambiguities.) Because Groovy's basic syntaxes have fewer tokens, there are more ambiguities, and so Groovy needs a strategy for helping programmers live with ambiguities, especially of type names vs. member names.

For example, which are the method names and which are the class names:

```
println message
String message
Wotsit message
```

The language needs to guess right most of the time, give good (preferably static) diagnostics when it guesses wrong, and make it easy to disambiguate the doubtful cases. (For a direct but clumsy example of a disambiguation hook, see 'typename' in C++.) This idea of 'good guessing' is an instance of the **Parse how I mean** tactic in design tactics.

About Static Methods

In Java, a static method call is superficially the same syntax as an object method call, but after the head token is disambiguated (class vs. variable), the deep syntax trees diverge.

In Groovy, the syntaxes are the same, and so a static method is called on a Class object. This requires an enhanced way of dispatching methods on Class.

Advisory Types on Declarations

If Classes are always specified (at least potentially) by variables, there are some places where Java absolutely needs to hardwire a type but Groovy will find it difficult. For example, if a declaration `C x = y` is allowed to have its head symbol `C` be a variable, it follows that the declaration's type may vary from run to run. This is both a strength and a weakness. One way to handle it is to forbid non-constant types in this position. Another is to allow them, with an advisory interpretation: The declaration actually declares two variables: a reference `x` and a meta-variable (of type Class) associated with `x`. All assignments to `x` go through a cast to x's type, if necessary. All uses of `x` can rely on the typing information in x's type, if the compiler can figure out a way to use it.

Here's an example that highlights this advisory interpretation in the presence of non-constant type names:

```
Class c = String
c var1 = "bar"  //var1.class == String
c = int
c var2 = 2  //var2.class == int
println var1[var2]  // prints 'r'
var1 = 123  //casts to String, even though c==int now
```

Perhaps useless, but is it a natural consequence of classes being objects always?