MethodData

A *profile* is information which summarizes the behavior of a bytecode instruction at some *profile point* in the program being executed by the JVM. The summary is designed to allow the optimizing compiler to guess at behaviors at the same point in the program. The profiles are crucial to later optimizing compilation.
A profile point is a specific instance of a bytecode. Not all bytecodes perform profiling.

Some bytecodes that operate on references perform type profiling. These may also record a bit which tells if a null was encountered.

Profile counts

A common form of data collected at a profile point is an execution count. The execution count allows the optimizing compiler to estimate the frequency of future executions of the code. Branches record taken and (if conditional) untaken counts. Method invocations also collect counts, since exceptions can cause downstream code to become less frequent. Switches record taken counts for each switch case.

Profile data structure

A profile is a metadata structure of type `MethodData`. Each method has zero or one of them. The structure is laid out as a heterogeneous array which is sequenced in parallel with the bytecodes themselves. Only only a minority of bytecodes capture profile data, the overall profile block is often larger than the bytecodes themselves. Each element in the profile array captures information for one instance of a bytecode in the method. (These are the profile points referred to above.)

A `MethodData` block is not created when its method is first loaded, but rather when the method is somehow noticed as relevant to execution (e.g., warm enough). Each profile applies to one bytecode method, and is affected by all executions of that method, from whatever caller.

The interpreter and some compiled code (tier one) collect profiles. Tier one emulates the interpreter with respect to profiling.

Profile pollution

Profiles (especially type profiles) are subject to pollution if the profiled code is heavily reused in ways the diverge from each other.

As a simple example, if `ArrayList.contains` is used with lists that never contain nulls, some null checks will never be taken, and the profile can reflect this. But if this routine is also used with lists that occasionally contain nulls, then the "taken" count of the null check instruction may become non-zero. This in turn may influence the compiler to check operands more cautiously, with a loss of performance for *all* uses of the method.

As a more complex example, if `ArrayList.contains` is only ever used on arrays that contains strings, then the type profile will reflect this, and the virtual call to `Object.equals` on each element can be optimistically inlined as if it were an invokespecial of `String.equals`.

Polluted profiles can be mitigated by a number of means, including:

- inlining with type or value information flowing from caller context
- inlining more than one type case (UseBimorphicInlining)
- context-dependent split profiles (bug 8015416)
- hand inlining by the Java programmer (which is discouraged)
- generic type reification (when pollution comes from a type parameter; not implemented)
- other forms of online code splitting
- adding type-check bytecodes for profiled references *before* the call to the shared routine (including `instanceof Object` which is a no-op except for profile effects)