DELPHIX MASKING · TECHNICAL REFERENCE

Masking Algorithms Guide

How each algorithm transforms data, with which parameters, and what to expect in the output.

31	9	62
ALGORITHMS	CATEGORIES	BUILT-IN INSTANCES

Plugin `delphix-algorithm-plugin 2026.3.0`

Outputs verified with AlgorithmRunner

Contents

Algorithms are grouped using the same categories shown in the Algorithm Tester sidebar. Every input → output pair in this guide was produced by actually running the algorithm — none of them are illustrative.

PART 1

Personal Data

Algorithms that understand the semantics of the field — email, phone, name, CPF/CNPJ — and generate synthetic substitutes that still look like (and validate as) real data.

1.1 Email

`algorithm.plugin.email.Email` **DETERMINISTIC**

Masks an email address by treating each half separately: the part before the `@` (the name) and the domain after it. Each half has its own strategy, which lets you fully anonymize the user while keeping the domain — preserving the per-company distribution your reports rely on.

INPUT FORMAT

> A valid email address, with `@` and a domain. E.g. `john.smith@company.com`. Malformed input falls through to `errorHandlingAction`.

INPUT → OUTPUT

INPUT		OUTPUT
`john.smith@company.com`	→	`TEQVTCLYKX6Z5EGR5LYA6E44BVUOSA7ITPUJTPGZCYDNNF5MMDYA@example.com`
`john.smith@company.com`	→	`TEQVTCLYKX6Z5EGR5LYA6E44BVUOSA7ITPUJTPGZCYDNNF5MMDYA@company.com`

First row uses `domainAction: REPLACEMENT`; the second uses `domainAction: PRESERVE`. Note the generated name is identical in both — it depends only on the input and the key.

PARAMETERS

`nameAction` *	How to mask the name part. **UNIQUE** — generates a long unique identifier per input (guarantees two different emails never collide). **LOOKUP** — draws a name from a lookup file. **APPLY_ALGORITHM** — passes the name through another algorithm. **APPLY_FIRSTNAME_AND_LASTNAME_ALGORITHMS** — splits the name and applies first-name and last-name algorithms separately, producing something like `ana.silva@…`.
`domainAction` *	How to handle the domain. **REPLACEMENT** — swaps it for `domainReplacementString`. **APPLY_ALGORITHM** — passes the domain through another algorithm. **PRESERVE** — keeps the original domain intact.
`domainReplacementString`	The substitute domain, used only when `domainAction = REPLACEMENT`. E.g. `example.com`.
`errorHandlingAction`	What to do when the input is not a valid email. **EXCEPTION** — fails with an error. **CHARACTER_MAPPING** — applies generic character-by-character masking. **PRESERVE** — returns the original value unchanged.

> **Choosing between them.** If the goal is simply to prevent accidental email delivery, `domainAction: REPLACEMENT` pointing at a domain you control is safest. If analysts still need to segment by company, use `PRESERVE` — but remember the domain can itself be identifying in small datasets.

1.2 Phone

`algorithm.plugin.phone.Phone` **DETERMINISTIC**

Generates a synthetic phone number preserving the original formatting exactly: parentheses, spaces, hyphens and digit count all stay in place. Only the digits change — and not all of them: the algorithm tends to preserve prefixes that identify the format (the area code).

INPUT FORMAT

> A phone number in any formatting. Punctuation is kept positionally in the output. E.g. `(555) 123–4567`, `5551234567`.

INPUT → OUTPUT

INPUT		OUTPUT
(555) 123–4567	→	**(555) 418–3796**
5551234567	→	**5554183796**

PARAMETERS

isPhoneUnique *	When true, the algorithm guarantees that two different source numbers never produce the same masked number. This matters when the phone is used as a deduplication or join key. When false, collisions are possible but the generated numbers look more natural.

1.3 Name

`algorithm.plugin.name.Name` **DETERMINISTIC** **REQUIRES FILE**

Replaces a simple name — just a first name, or just a surname — with another drawn from a lookup file. The draw is a hash of (value + key), so the same name always maps to the same substitute: "John" will be "Thomas" across every table in the project, which keeps name-based joins working. For compound names use **Full Name** (§1.4).

INPUT FORMAT

> A single word, no separators. E.g. `John`, `Smith`, `Mary`.

INPUT → OUTPUT

INPUT		OUTPUT
John	→	**Thomas**
MARY	→	**ROBERT**

With `maskedValueCase: PRESERVE_INPUT` — uppercase input produced uppercase output, even though the file contains "Robert".

PARAMETERS

`lookupFile` *	A text file with one name per line. It is the source of substitutes. More lines means a lower chance that two different names receive the same substitute. It can hold any list — first names, surnames, male or female names separately.
`maskedValueCase`	Capitalization of the result. **PRESERVE_LOOKUP_FILE** — exactly as it appears in the file. **PRESERVE_INPUT** — copies the input's case pattern. **ALL_LOWER** / **ALL_UPPER** — forces lower or upper case.
`particlesToRemoveFile`	A file of particles (one per line) stripped from the input *before* hashing. This makes "van Dyke" and "Dyke" yield the same substitute instead of being treated as distinct names.
`particlesToPreserveFile`	A file of particles that must reappear in the result. The generated substitute gets back the particles the original input had.
`maxLengthOfMaskedName`	Maximum length in characters. Names in the file exceeding the limit are excluded from the draw. Use this when the target column has a size constraint and you don't want truncation.
`maxNumberNames`	Maximum names returned in multi-value mode. For a simple single column, leave it empty.
`filterAccent`	When true, strips accents before hashing — making "José" and "Jose" produce the same substitute. Recommended for Brazilian datasets where accentuation is inconsistent.
`inputCaseSensitive`	When true, "John" and "JOHN" are distinct inputs and may receive different substitutes. The default (false) ignores case when comparing.

1.4 Full Name

`algorithm.plugin.name.FullName` **DETERMINISTIC**

Splits a full name into first name(s) and surname, then applies a different algorithm to each part. It is an orchestrator: it masks nothing itself, it only decides where the given name ends and the surname begins, delegating each piece to the algorithm you point it at.

INPUT FORMAT

> A full name separated by spaces. E.g. `John Smith`, `Mary Anne Johnson`.

INPUT → OUTPUT

INPUT		OUTPUT
`John Michael Smith`	→	**`Pink Deepak Luella`**

PARAMETERS

`lastNameAtTheEnd`	Where the surname sits. **true** — it is the last word ("John Michael Smith" → surname "Smith"). **false** — it is the first (for datasets storing "Smith, John").
`ifSingleWordConsiderAsLastName`	Tie-breaker for single-word input. **true** — treat it as a surname. **false** — treat it as a first name. This decides which of the two algorithms gets called.
`firstNameAlgorithmRef`	Algorithm applied to the first name(s). E.g. `dlpx-core:FirstName`. If omitted, first names are left *unmasked*.
`lastNameAlgorithmRef`	Algorithm applied to the surname. E.g. `dlpx-core:LastName`. If omitted, the surname is left unmasked.
`lastNameSeparators`	Additional separators beyond the space. E.g. `["-"]` makes "Mary-Smith" read as name "Mary" + surname "Smith".
`maxLengthOfMaskedName`	Character limit for the masked full name, to respect the width of the target column.
`maxNumberFirstNames`	How many first names to mask. With `1`, "John Peter Charles Smith" has only "John" masked — "Peter Charles" passes through untouched.

> **Mind the defaults.** Both `…AlgorithmRef` parameters are optional, and when left empty the corresponding part is **not masked**. It is easy to configure only the surname and let first names leak into production — always check both.

1.5 Financial ID BR (CPF/CNPJ)

`algorithm.plugin.financialId.FinancialIdBr` **DETERMINISTIC**

Masks Brazilian CPF and CNPJ numbers, generating synthetic values that remain **mathematically valid** — the check digits are recalculated. The type is detected automatically from the digit count: 11 is a CPF, 14 is a CNPJ. Input formatting (dots, slash, hyphen) is preserved in the output.

INPUT FORMAT

> A CPF or CNPJ, formatted or not. E.g. `123.456.789-09`, `12345678909`, `11.222.333/0001-81`, `11222333000181`.

INPUT → OUTPUT

INPUT		OUTPUT
`123.456.789-09`	→	`976.201.860-50`
`11.222.333/0001-81`	→	`50.009.982/0001-50`
`11222333000181`	→	`50009982000150`
`12345678000190`	→	`Error: invalid check digits (maskInvalidInput = false)`

Rows 2 and 3 show the same CNPJ with and without punctuation: the generated digits are identical, only the formatting differs.

PARAMETERS

`maskInvalidInput`	What to do when the incoming CPF/CNPJ fails check-digit validation. **false** (default) — raises an error and stops. **true** — masks it anyway, ignoring the invalidity. Legacy datasets commonly contain invalid documents; if that is your case, enable this or configure `fallbackAlgorithm`.
`cmNumericRef`	Algorithm used internally to shuffle the digits before the check digit is recalculated. If empty, uses the built-in numeric Character Mapping. Supply a catalog instance, e.g. `dlpx-core:CM Digits`.
`fallbackAlgorithm`	Algorithm invoked when the value is recognized as neither CPF nor CNPJ — blank field, free text, unexpected length. Without it, those cases become errors.

PART 2

Dates

Four distinct strategies for dates: shift while preserving intervals, shift by discrete values, replace outright, or merely clamp to a range.

2.1 Date Shift

`algorithm.plugin.dateAlgorithms.DateShift` **DETERMINISTIC**

Shifts the date by an amount drawn from `[minRange, maxRange]`. The draw is deterministic by key, so the same date always gets the same shift — which **preserves chronological order** across records. It is the algorithm of choice when analyses depend on intervals ("days between order and delivery").

INPUT FORMAT

> ISO 8601: `yyyy-MM-dd` or `yyyy-MM-ddTHH:mm:ss`. The output always includes a time component.

INPUT → OUTPUT

INPUT		OUTPUT
`2024-03-15`	→	**`2024-10-20T00:00`**
`2024-03-15T14:30:00`	→	**`2024-03-26T14:30`**
`2024-01-31`	→	**`2023-04-30T00:00`**

The first two use `±365 DAYS`; the third uses `±12 MONTHS`. Note the original time-of-day is always preserved.

PARAMETERS

`minRange`	Minimum shift, in units of `unit`. Negative allows moving backwards. E.g. `-365` with `DAYS` allows going back a year.
`maxRange`	Maximum shift. Positive allows moving forward. A narrow range better preserves data plausibility; a wide range increases protection.
`unit`	Unit of the shift: `SECONDS`, `MINUTES`, `HOURS`, `DAYS`, `MONTHS`, `YEARS`.
`roll`	Only relevant with `MONTHS` or `YEARS`, when the result would land on a non-existent date (January 31 + 1 month = February 31). **true** — shrinks to the last valid day of the month (Feb 28). **false** — overflows into the next month (Mar 3). When unsure, use `false`.

> **On interval preservation.** Because the shift depends on the source date, two records with different dates get different shifts — the distance between them is **not** preserved exactly. If keeping distances intact is a requirement, the shift must be fixed per entity, which is what `dlpx-core:Date Shift Fixed` provides.

2.2 Date Shift Discrete

`algorithm.plugin.dateAlgorithms.DateShiftDiscrete` **DETERMINISTIC**

A variant of Date Shift where the shift comes not from a continuous range but from a **finite set of allowed values** defined in a configuration file. Useful when the business requires standardized shifts — for instance, only multiples of 7 days, to preserve the day of the week.

INPUT FORMAT

> ISO 8601: `yyyy-MM-dd` or `yyyy-MM-ddTHH:mm:ss`.

INPUT → OUTPUT

INPUT		OUTPUT
`2024-03-15`	→	**`2024-03-29T00:00`**

PARAMETERS

No parameters exposed in the schema — the set of shifts comes from the plugin's built-in configuration.

2.3 Date Replacement

`algorithm.plugin.dateAlgorithms.DateReplacement` **DETERMINISTIC**

Discards the original date and draws an **absolute** date within `[minDate, maxDate]`. Unlike Date Shift, there is no relationship between input and output beyond determinism — the original chronological order is destroyed. Use it when the date itself is sensitive and no temporal analysis depends on it.

INPUT FORMAT

> ISO 8601 with time: `yyyy-MM-ddTHH:mm:ss`. E.g. `1985-07-20T00:00:00`.

INPUT → OUTPUT

INPUT		OUTPUT
`1985-07-20T00:00:00`	→	**`1988-08-13T00:00`**

PARAMETERS

`minDate`	Earliest possible output date, formatted `yyyy-MM-ddTHH:mm:ss`. Sets the floor of the draw window.
`maxDate`	Latest possible date, same format. Pick a window plausible for the domain — birth dates in 1970–2000, for example.
`unit`	Granularity of the draw: `SECONDS`, `MINUTES`, `HOURS` or `DAYS`. With `DAYS`, the time component is zeroed.

2.4 Min/Max Date/Time

`algorithm.plugin.minMax.MinMaxLocalDateTime`

Not really masking, but **range clamping**: dates inside the range pass through untouched; dates outside are pulled to the nearest bound. The typical use is suppressing outliers that identify individuals — the centenarian customer, the 1900 sentinel date.

INPUT FORMAT

> ISO 8601. E.g. `2024-03-15T14:30:00`.

INPUT → OUTPUT

INPUT		OUTPUT
`1985-06-15T08:00:00`	→	**`1985-06-15T08:00`**
`1950-01-01T00:00:00`	→	**`1970-01-01T00:00`**
`2024-05-10T12:00:00`	→	**`2000-12-31T23:59:59`**

Range `1970-01-01` to `2000-12-31T23:59:59`. The first date is inside and passes unchanged; the second is below the floor; the third is above the ceiling.

PARAMETERS

`minDate` *	Floor of the range. Any earlier date is replaced by this value.
`maxDate` *	Ceiling of the range. Any later date is replaced by this value.
`nonConformingDataDefaultValue`	Value returned when the input is not a valid date/time. Left empty, the algorithm raises a non-conforming data error and stops the job.

> **This does not anonymize.** Values inside the range come out **identical** to the input. Min/Max is a tool for suppressing outliers, not for masking — never use it alone on a sensitive column.

PART 3

Numeric

Transformations over numeric values: deterministic mapping into a range, range clamping, arbitrary Java expressions, and one fixed digit-repetition rule.

3.1 Numeric Mapping

`algorithm.plugin.characterMapping.NumericMapping` **DETERMINISTIC**

Maps a number to another one within `[minValue, maxValue]`, deterministically. Each source value consistently receives the same destination, which preserves joins and distinct counts — but neither order nor magnitude.

INPUT FORMAT

> A positive integer, no punctuation or decimals. E.g. `12345`.

INPUT → OUTPUT

INPUT		OUTPUT
12345	→	**67253**
98765	→	**53534**
11111	→	**50708**

PARAMETERS

minValue	Smallest value the output may take. Choose a floor that respects the column's semantics (e.g. `10000` to always guarantee 5 digits).
maxValue	Largest output value. The width of the range determines collision probability: narrow ranges make distinct values collide onto the same destination.

3.2 Min/Max BigDecimal

`algorithm.plugin.minMax.MinMaxBigDecimal`

The numeric counterpart of Min/Max Date/Time: clamps the value to a range, letting anything already inside pass through untouched. It serves to flatten outliers — very high salaries, extreme negative balances — that on their own identify a person.

INPUT FORMAT

> A decimal or integer number. E.g. `5000.50`, `3.14`.

INPUT → OUTPUT

INPUT		OUTPUT
5000.50	→	**5000.50**
250	→	**1000**
15000	→	**9999**

Range `1000` – `9999`. Note the decimal scale of the in-range value is preserved.

PARAMETERS

`minValue` *	Lower bound. Smaller values are raised to this number.
`maxValue` *	Upper bound. Larger values are lowered to this number.
`nonConformingDataDefaultValue`	Value returned when the input is not numeric. Empty makes the algorithm raise a non-conforming data error.

> **This does not anonymize.** Just like the date Min/Max, values inside the range come out identical. Combine it with another algorithm if the column is sensitive.

3.3 Numeric Expression

`algorithm.plugin.expression.NumericExpression`

The most open-ended algorithm in the set: it evaluates a **one-line Java expression** over the input value. The expression receives two variables — `input` (the value as a `BigDecimal`) and `seed` (a `long` derived from the key, for deterministic masking) — plus any constants you declare.

INPUT FORMAT

> An integer or decimal number. E.g. `1000`, `3.14`.

INPUT → OUTPUT

INPUT · EXPRESSION		OUTPUT
`1234.56` `input.setScale(0, HALF_UP)`	→	**`1235`**
`99.4` `input.setScale(0, HALF_UP)`	→	**`99`**
`1234.56` `new BigDecimal(seed % 9000 + 1000)`	→	**`7648`**
`1000` `input.multiply(new BigDecimal(rate))`	→	**`800.000000000000044408920985006261616945266723632812500`**

PARAMETERS

`expression` *	A Java expression returning `BigDecimal`. Available variables: `input` and `seed`. Classes need fully qualified names — `new java.math.BigDecimal(...)`, `java.math.RoundingMode.HALF_UP`.
`inputType`	How to interpret the input before exposing it as `input`: `DOUBLE`, `LONG` or `BIG_DECIMAL` (default).
`constants`	List of named constants available in the expression. Each has a `name` and a `value` (always a string). E.g. `name="rate"`, `value="0.8"`, used as `new java.math.BigDecimal(rate)`.
`nonConformingDataDefaultValue`	Value returned when the input is not numeric. Empty raises an error.

> **Watch out for floating point.** The fourth row above returned `800.000000000000044…` instead of `800`: the constant `"0.8"` went through a `double` before becoming a `BigDecimal`. For exact arithmetic, use the string constructor — `new java.math.BigDecimal("0.8")` — and finish with `.setScale(2, java.math.RoundingMode.HALF_UP)`.

3.4 Repeat First Digit

`algorithm.plugin.repeatFirstDigit.RepeatFirstDigit`

A fixed rule with no configuration: it takes the **last 4 digits** of the number and replaces them all with the first of those digits, repeated four times. The rest of the number stays intact. This is weak masking, suitable for low-sensitivity fields where you only want to break the exact value.

INPUT FORMAT

> A numeric string of exactly **4, 9, 10 or 14 digits**, without punctuation. Other lengths are rejected.

INPUT → OUTPUT

INPUT		OUTPUT
1234	→	**1111**
123456789	→	**123456666**

In the second case the first 5 digits are preserved and the last 4 (`6789`) become `6666` .

PARAMETERS

None. The behavior is fixed.

PART 4

String

The general-purpose algorithms for structured text. These are the plugin's most-used tools — and its most configurable.

4.1 Character Mapping

`algorithm.plugin.characterMapping.CharacterMapping` **DETERMINISTIC**

Swaps each character for another **from the same group**. You define the groups (lowercase letters, uppercase, digits…) and the algorithm guarantees a letter becomes a letter and a digit becomes a digit, preserving length and the "visual shape" of the data. Characters belonging to no group — spaces, accents, punctuation — pass through untouched.

INPUT FORMAT

> Any string. E.g. `Renée Smith`, `ABC123`.

INPUT → OUTPUT

INPUT		OUTPUT
`Renée Smith123`	→	**`Kyxéf Zawtv529`**
`ABC123`	→	**`YHE638`**
`Mary`	→	**`Rtut`**
`Renée Smith123 (different key)`	→	**`Qvtéi Dfnju271`**

The `é` and the space survived — neither belongs to a configured group. The last row shows the effect of the key: same input, different key, different result.

PARAMETERS

`characterGroups`	A list of strings; each string is a set of characters interchangeable with one another. E.g. `["abcdefghijklmnopqrstuvwxyz", "0123456789"]` creates two isolated groups — letters only become letters, digits only digits. Characters absent from every group are never masked.
`caseSensitive`	When true, uppercase and lowercase are treated as separate groups, preserving the original capitalization. This requires you to declare both sets separately.
`minMaskedPositions`	Minimum number of positions that must actually change. Prevents the algorithm from returning a nearly identical value by coincidence of the draw.
`preserveRanges`	List of stretches to keep intact, each with `start`, `length` and `direction`. Use it to preserve meaningful prefixes or suffixes — branch code, control digit.
`preserveLeadingZeros`	When true, leading zeros are kept. Essential for numeric codes stored as text, where `007` and `7` are not equivalent.

> **Why accents don't change.** If accented characters need masking, include them explicitly in a group: `"aáàâãeéêiíoóôõuú"`. Otherwise they remain visible in the masked data and can help re-identify the record.

4.2 Character Replacement

`algorithm.plugin.characterReplacement.CharacterReplacement`

Applies character-by-character substitution rules: each rule declares *which* characters to detect and *what* to replace them with. Unlike Character Mapping, the substitution here is fixed (not drawn) — every filtered character becomes the same symbol. It can optionally run after another algorithm, via `stringAlgorithm`.

INPUT FORMAT

> Any string. E.g. `Some example text`.

INPUT → OUTPUT

INPUT		OUTPUT
`Some example text`	→	**`R*v* *m*gyy* c*nz`**
`Mary Johnson`	→	**`Q*lv T*vb**h`**
`Renée François` `(filterAccents: INPUT)`	→	**`**n** Fp*ls**n`**

Rule: vowels → `*`. The consonants changed too because the algorithm applies its own masking before the rules. In the third row, `filterAccents: INPUT` converted `é` and `ç` to ASCII before processing.

PARAMETERS

`stringAlgorithm`	Algorithm applied to the input *before* the substitution rules. Its result is what passes through `replacementRules`. This lets you compose: mask with a strong algorithm, then normalize problematic characters.
`replacementRules`	List of rules applied in sequence. Each rule is a pair (set of characters to detect, replacement character).
↳ **`filteredCharacters`**	The characters this rule detects. Simple form, used when input and output share the same set.
↳ **`replacementCharacter`**	The single character that replaces each occurrence found.
↳ **`inputFilteredCharacters`**	Advanced form: characters detected in the input, when you want different sets for reading and writing.
↳ **`inputReplacementCharacter`**	Substitute applied to matches of `inputFilteredCharacters`.
↳ **`outputFilteredCharacters`**	Characters that, should they appear in the *output* of `stringAlgorithm`, must also be replaced.
↳ **`outputReplacementCharacter`**	Substitute applied to matches of `outputFilteredCharacters`.
`requireInputChange`	When true, requires at least one substitution to have happened — if no rule changed anything, the algorithm raises an error. This guards against configurations that silently mask nothing.
`filterAccents`	When to strip accents. **INPUT** — before applying the rules. **OUTPUT** — on the final result. **BOTH** — at both stages. **NONE** — never (default).

4.3 Segment Mapping

`algorithm.plugin.segmentMapping.SegmentMapping` **DETERMINISTIC**

Splits the string into fixed-length segments and handles each one independently — mask, preserve, or replace with a constant. This is the right algorithm for rigidly formatted documents where different parts carry different meaning: CPF, CNPJ, postal code, barcode, account number with an embedded branch code.

INPUT FORMAT

> A fixed-format string. With `autoIgnoreCharacters` enabled, separators (dots, hyphens, slashes) are detected and repositioned automatically. E.g. `123.456.789-09`.

INPUT → OUTPUT

INPUT		OUTPUT
123.456.789-09	→	**991.440.480-70**
987.654.321-00	→	**684.669.226-24**

Segments of 3-3-3-2 digits, all `MASK_NUMERIC`, with separators preserved in their original positions.

PARAMETERS

segments *	An ordered list of segments, in the sequence they appear in the value (ignoring ignored characters). The sum of the lengths must match the size of the data.
↳ **length** *	How many characters this segment occupies. Ignored characters do not count.
↳ **segmentType** *	**MASK_NUMERIC** — replaces with digits. **MASK_ALPHANUMERIC** — replaces with alphanumeric characters. **PRESERVE** — keeps the original. **CONSTANT** — replaces with the fixed value from `maskValues`.
↳ **inputValues**	Set of characters accepted in this segment on input, as a continuous string. E.g. `"0123456789"`. Empty accepts any character.
↳ **maskValues**	Pool of characters used for substitution. E.g. `"123456789"` (no zero) prevents the segment from starting with `0`. Empty uses the type's default pool.
ignoreCharacters	ASCII codes of the separators to skip when counting positions — `46` dot, `45` hyphen, `47` slash. They return to the output in their original positions.
autoIgnoreCharacters	When true, automatically detects all non-alphanumeric characters and ignores them. Removes the need to list ASCII codes by hand — recommended for CPF, CNPJ and postal codes.
allowShortSegments	When true, accepts input shorter than the sum of the segments, processing whatever is there. The default (false) raises an error.
processPreserveBeforeIgnore	Order of operations between removing ignored characters and computing `PRESERVE` segments. Enable it when a separator falls inside a preserved segment and positions come out shifted.

4.4 Regex Decompose

`algorithm.plugin.decompose.RegexDecompose`

The most flexible algorithm for structured text. You define regex patterns; the first one matching the **entire** string is applied, and each capture group (parentheses) receives an independent action — preserve, drop, redact, or pass through another algorithm. Text *between* groups is preserved automatically.

INPUT FORMAT

> Any string. Patterns are tested in declaration order until one matches. If none match, `fallbackAction` applies.

INPUT → OUTPUT

INPUT · REGEX		OUTPUT
`user@company.com` `([^@]+)(@[^@]+) → REDACT, PRESERVE`	→	`***@company.com`
`000001999191111` `(\d{6})(\d{9}) → PRESERVE, REDACT "X"`	→	`000001XXXXXXXXX`
`"000001999191111  "` `same pattern, trimInput: true`	→	`"000001XXXXXXXXX  "`
`123456789` `(\d{3})(\d+) → PRESERVE, TRUNCATE`	→	`123`
`ABC` `no fallbackAction`	→	`Error: No pattern matched and no fallbackAction is defined`
`ABC` `fallback REDACT "[INVALID]"`	→	`[INVALID]`

PARAMETERS

`maskPatterns`	List of patterns tested **in order**. The first whose regex matches the entire string wins — the rest are ignored. This ordering is what enables conditional routing (see the tip below).
↳ **`regex`**	A regular expression that must match the **entire** string (implicitly anchored). Capture groups delimit the parts that receive actions.
↳ **`actions`**	One action per capture group, in the same order as the parentheses. Two groups require exactly two actions.
↳ **`actions.type`**	**PRESERVE** — keeps the stretch. **TRUNCATE** — removes it (becomes an empty string). **REDACT** — replaces it with fixed text or a character. **APPLY_ALGORITHM** — passes it through another algorithm.
↳ **`actions.redactString`**	Fixed text replacing the *whole* group, regardless of original length. E.g. `"***"`. Only with `REDACT`.
↳ **`actions.redactCharacter`**	A single character replacing *each* character of the group, preserving length. E.g. `"X"` turns 9 digits into `XXXXXXXXX`. Mutually exclusive with `redactString`.
↳ **`actions.algorithm`**	Algorithm instance applied to the group when the type is `APPLY_ALGORITHM`. Accepts built-in catalog names (appendix B) or tests saved in the Tester.
`fallbackAction`	Action applied to the whole value when no pattern matches. It has the same four types, with its own `redactString`, `redactCharacter` and `algorithm`.

`trimInput`	Strips whitespace from both ends *before* attempting to match. The padding is restored in the output (see the third row of the table). Essential for `CHAR(n)` columns, which arrive right-padded with spaces.
`requireMask`	When enabled, raises an error if no pattern matches, instead of using the fallback. Use it when out-of-format values must be rejected rather than passed along.
`maxInputLength`	Maximum accepted input length. Longer values become errors. Empty imposes no limit.

Two common errors.
① `NullPointerException … action.type is null` — when adding an action through the form, the **Action type** field starts empty; select it explicitly.
② `Fallback action may not be PRESERVE when requireMask is set` — `requireMask` defaults to **true**; to use `fallbackAction: PRESERVE`, set `requireMask: false` explicitly.

Conditional routing by content. Because patterns are tested in order, you can route a value to different algorithms depending on what it contains. To apply one algorithm when there is an `@` and another when there isn't:

```
{"maskPatterns": [
  { "regex": "(.+@.+)",
    "actions": [{"type":"APPLY_ALGORITHM","algorithm":{"name":"dlpx-core:Email
SL"}}] },
  { "regex": "(.+)",
    "actions": [{"type":"APPLY_ALGORITHM","algorithm":{"name":"dlpx-
core:FirstName"}}] }
]}
```

The first pattern only matches values containing `@`; everything else falls to the second, which acts as a catch-all.

4.5 String Algorithm Chain

`algorithm.plugin.stringAlgorithmChain.StringAlgorithmChain`

Chains algorithms in sequence: the output of one becomes the input of the next. It requires at least two. Use it to compose transformations no single algorithm offers — mask then normalize, or apply two layers of substitution.

INPUT FORMAT

> A string compatible with the *first* algorithm in the chain. The rest receive whatever the previous one produced.

INPUT → OUTPUT

INPUT		OUTPUT
`John Smith`	→	**`Salomon`**

Chain `FirstName` → `LastName`: the first algorithm produced a name, and the second treated that result as if it were a surname.

PARAMETERS

`algorithmReferences`	An ordered list of instance references, **minimum 2**. E.g. `[{"name":"dlpx-core:FirstName"}, {"name":"dlpx-core:LastName"}]`. Names come from the built-in catalog (appendix B) or from tests saved in the Tester.

> **Order matters, and so does the fit.** Each algorithm must accept the format the previous one produced. Chaining a date algorithm after a name algorithm, for instance, fails — the second receives text it cannot parse.

4.6 Shuffle

`algorithm.plugin.shuffle.Shuffle` ⚠ NON-DETERMINISTIC BATCH MODE

Redistributes values **across** records: each row receives a value that belonged to another. No value is invented — the set stays identical, only the associations change. This perfectly preserves the column's statistical distribution, making it ideal for columns used in aggregations.

INPUT FORMAT

> Multiple values at once (batch mode). Each value must be at least `minimumShuffleSize` characters long.

INPUT → OUTPUT

INPUT BATCH		OUTPUT BATCH
Mary Johnson	→	**Peter Miller**
John Davis	→	**Anna Smith**
Peter Miller	→	**Charles Wilson**
Anna Smith	→	**Mary Johnson**
Charles Wilson	→	**John Davis**

PARAMETERS

`minimumShuffleSize`	Minimum length for a value to take part in the shuffle (default `3`). Shorter values are excluded and pass through unchanged.

> **Non-deterministic — by design.** The permutation changes on every run. This is a security decision: a reproducible permutation would let someone re-run the shuffle and undo the anonymization. The practical consequence is that Shuffle **does not preserve referential integrity** across tables or across runs.

PART 5

Free Text

For free-text fields — notes, reports, comments — where the sensitive data is embedded in prose that must stay readable.

5.1 Free Text Redaction

`algorithm.plugin.freeTextRedaction.FreeTextRedaction`

Scans free text for sensitive entities — by regular expression and/or by a list of terms in a file — and replaces only the matched stretches. Everything else is preserved, keeping the document readable and useful for analysis.

INPUT FORMAT

> Free text of any length. E.g. `Contact John by email at john@company.com or on 555-0100.`

INPUT → OUTPUT

INPUT		OUTPUT
`Contact John by email at john.smith@company.com.`	→	**`Contact John by email at [REDACTED].`**

PARAMETERS

`regularExpressions`	A list of `{patternString}` objects with the regexes identifying stretches to redact. Each pattern is applied across the whole text, to every occurrence.
`isDenyList`	**true** — deny list: redacts what *matches* the patterns. **false** — allow list: redacts everything that does *not* match. The second option is safer for unpredictable text, but usually destroys readability.
`regExRedactValue`	Text replacing the stretches captured by the regular expressions. E.g. `[REDACTED]`.
`lookupFile`	A file of literal terms, one per line, searched for in the text. Useful for proper nouns and terms that regex describes poorly.
`lookupFileRedactValue`	Text replacing occurrences found via the file. It is independent of `regExRedactValue`, letting you visually mark the origin of each redaction.

5.2 Redact

`algorithm.plugin.redact.Redact`

A simpler, more direct take on redaction: a map of `regex → replacement text`. Each map key is a pattern, and the corresponding value is what takes its place. With no regex configured, it returns the input value with no masking at all.

INPUT FORMAT

> Any string. Stretches matching the regexes are replaced.

INPUT → OUTPUT

INPUT		OUTPUT
`Contact John by email at john.smith@company.com.`	→	**`Contact John by email at [EMAIL].`**

PARAMETERS

`regexRedact`	A map of `{regex → replacement_text}`. It allows different markers per data type — `[EMAIL]`, `[CPF]`, `[PHONE]` — in a single configuration.

> **An empty configuration masks nothing.** With no regex in `regexRedact`, the algorithm returns the input untouched and issues **no** warning. Always confirm the map was filled in.

PART 6

Financial

Financial instruments with verifiable structure: cards with a Luhn check digit and IBANs with a country checksum.

6.1 Payment Card

`algorithm.plugin.characterMapping.PaymentCard` **DETERMINISTIC**

Masks card numbers while preserving the **BIN** (the leading digits identifying the network and issuer) and, optionally, the trailing digits used in reconciliation. The generated number keeps a **valid Luhn check digit**, so it still passes form and system validation.

INPUT FORMAT

> A card number with or without spaces/hyphens. E.g. `4111 1111 1111 1111`.

INPUT → OUTPUT

INPUT		OUTPUT
4111111111111111	→	**4111533479707760**
5500000000000004	→	**5500138521866444**
4111111111111111 `(preserve: 0)`	→	**8036550265772777**

In the first two, the BIN (`4111` , `5500`) was preserved automatically. In the third, with `preserve: 0` , even the BIN changed — the card network is no longer identifiable.

PARAMETERS

`preserve`	How many digits from the *end* of the card to preserve. Typically `4` , to keep the last four that appear on statements and confirmation screens.
`minMaskedPositions`	Minimum number of positions that must actually change. Ensures the masked card does not end up too similar to the original.

> **Preserving the last 4 has a cost.** Last-four plus BIN plus transaction date is often enough to re-identify a card in small datasets. Use `preserve: 4` only when the business process genuinely depends on it.

6.2 IBAN

`algorithm.plugin.iban.IBAN` **DETERMINISTIC**

Masks an IBAN while preserving the country code and recalculating the check digits, so the result remains a structurally valid IBAN. You control how many characters of the national part get shuffled.

INPUT FORMAT

> An IBAN in standard format, without spaces. E.g. `GB29NWBK60161331926819`.

INPUT → OUTPUT

INPUT		OUTPUT
`GB29NWBK60161331926819` (mask 20)	→	**`GB54EPWF99117911522013`**
`GB29NWBK60161331926819` (mask 8)	→	**`GB26NWBK60161384059911`**

With `numCharsToMask: 8` the bank code (`NWBK`) and branch survive; with `20`, practically only the country remains. `GB` is always preserved and the checksum recalculated.

PARAMETERS

`validateInput` *	When true, validates the IBAN checksum before masking and rejects invalid input. Turn it off only if the dataset holds knowingly malformed IBANs that still need processing.
`numCharsToMask` *	How many characters to shuffle, counted from the end of the national part. Low values preserve bank and branch; high values mask everything after the country code.

PART 7

Lookup & Mapping

Algorithms that substitute values by consulting an external source — a file of substitutes, a lookup table, or a persistent mapping database.

7.1 Secure Lookup

`algorithm.plugin.secureLookup.SecureLookup` DETERMINISTIC ⚠ EXCEPT WITH RANDOMIZE REQUIRES FILE

Replaces the value with a line from a lookup file, chosen by hashing (value + key). The same input always selects the same line, which gives referential consistency across tables without needing a mapping database. It is the plugin's most widely used lookup algorithm.

INPUT FORMAT

> Any non-null string. The file's contents determine the output domain.

INPUT → OUTPUT

INPUT		OUTPUT
`John Smith`	→	**`Sarah`**
`Mary Johnson`	→	**`William`**
`John Smith` `(ALL_UPPER)`	→	**`SARAH`**

The third row shows that `maskedValueCase` only affects capitalization — the line selected from the file is still the same.

PARAMETERS

`lookupFile` *	A file with one substitute value per line. The number of lines defines output diversity: a file of 20 names will make many distinct values collide onto the same substitute.
`hashMethod`	How the line index is derived. **SHA256** — cryptographic hash, deterministic by key (recommended). **LEGACY** — the old method, kept for compatibility with already-masked data. **RANDOMIZE** — random selection, *not* deterministic: every run changes the result.
`maskedValueCase`	Output capitalization. **PRESERVE_LOOKUP_FILE** — as in the file. **PRESERVE_INPUT** — copies the input's pattern. **ALL_LOWER** / **ALL_UPPER**.
`inputCaseSensitive`	When true, "John" and "JOHN" produce distinct hashes and may receive different substitutes. Leave it false to treat case variations as the same value.
`trimWhitespaceFromInput`	Strips whitespace from both ends of the input before hashing. Makes `"  John  "` and `"John"` land on the same substitute — important for `CHAR(n)` columns.
`trimWhitespaceInLookupFile`	Strips whitespace from both ends of each file line on load, preventing lines with trailing spaces from becoming distinct values.

7.2 Null-Safe Secure Lookup

`algorithm.plugin.nullSecureLookup.NullSecureLookup` **LIMITED IN THE RUNNER**

A variant of Secure Lookup with explicit null handling: null input returns null instead of raising an error. It exposes no parameters — it uses the lookup file embedded in the plugin.

INPUT FORMAT

> Any string, or null.

INPUT → OUTPUT

INPUT		OUTPUT
`sensitive-value`	→	`null` (in the standalone runner)

PARAMETERS

None.

> **In the standalone runner it always returns `null`.** The algorithm depends on the Masking Engine context to resolve its embedded lookup file. A `null` result here is expected and does not indicate a configuration error.

7.3 Data Cleansing

`algorithm.plugin.dataCleansing.DataCleansing` **REQUIRES FILE**

Substitutes values according to an **explicit** lookup table: you declare exactly which value becomes which. Unlike Secure Lookup, there is no hashing or drawing. In practice it is more a standardization tool than a masking one — normalizing abbreviations, unifying divergent spellings.

INPUT FORMAT

> A string with a match in the file. With no match, the original value passes through **unchanged**.

INPUT → OUTPUT

INPUT		OUTPUT
CA	→	**California**
ny	→	**New York**
XX	→	**XX** (no match)

The second row matched despite the different case because `caseSensitive` is false.

PARAMETERS

lookupFile *	A file with one mapping per line, formatted `original{delimiter}substitute`. E.g. `CA,California`.
delimiter	Separator between original and substitute. E.g. `,` for CSV, `;`, or a tab. Omitted, it uses a tab.
caseSensitive	When true, `SP` and `sp` need separate entries in the file. False is the more practical choice in most cases.
trimWhitespace	Strips whitespace from both ends of the input and of the file lines before comparing, avoiding match failures caused by database padding.

> **Unmatched values pass through intact.** This means an incomplete lookup table lets original data leak out **silently**. If the column is sensitive, validate the file's coverage before running in production.

7.4 Mapping

`algorithm.plugin.mapping.Mapping` **DOES NOT WORK IN THE RUNNER**

Maintains a persistent one-to-one mapping in a database: each original value receives a unique substitute, stored and reused across every table and every future run. It is the plugin's strongest guarantee of **referential consistency** — and the only one that survives between separate jobs.

INPUT FORMAT

> Any string. The algorithm queries the mapping set and, if the value is new, creates and persists a substitute.

INPUT → OUTPUT

INPUT		OUTPUT
`customer@email.com`	→	`Error: Mapping set references are not supported`

PARAMETERS

`mappingSet` *	Reference to the mapping set that stores the correspondences.
↳ **`algorithmName`** *	Name of the mapping set in the Masking Engine — it identifies which set to use. E.g. `client-name-mapping`.
↳ **`host`**	Host of the database holding the mapping set, when remote.
↳ **`port`**	Port of the remote mapping set database.
↳ **`database`**	Name of the remote database.
↳ **`schema`**	Schema of the remote database.
↳ **`isRemote`**	When true, uses the connection details above. False uses the Engine's local instance.
↳ **`mappingLookupKey`**	A partition key letting the same mapping set serve multiple isolated contexts — per client, per environment.
↳ **`propertiesRef`**	A properties file with the connection configuration, as an alternative to filling in host/port/database/schema individually.
`ignoreCharacters`	ASCII codes to strip from the input before the lookup. E.g. `45` hyphen, `46` dot — making a CPF with and without formatting point at the same mapping.

> **Unavailable in the Algorithm Tester.** The standalone runner has no mapping database; any test returns `Mapping set references are not supported`. This algorithm can only be validated in the Masking Engine.

PART 8

Multi-Column

Algorithms that receive the whole row rather than a single value — allowing the masking of one column to depend on the contents of another.

8.1 Multi-Column Condition

`algorithm.plugin.conditional.MultiColumnCondition` **DETERMINISTIC** **MULTI-COLUMN**

Chooses which algorithm to apply to each column **based on the value of a conditional column**. The classic case: if the gender column says "F", mask the name with a list of female names; if "M", with male names — keeping the record internally coherent.

INPUT FORMAT

> A row with named columns. Required: `key` (the conditional column). Optional: `string1` … `string10`, `numeric1` … `numeric3`, `date1` … `date3`, `binary1` … `binary3`.

INPUT → OUTPUT

INPUT ROW		OUTPUT ROW
`key = "F"` `string1 = "Mary"`	→	**`key = "F"`** **`string1 = "Karie"`**

The `key` column is not masked — it only decides which condition applies. Only `string1` had an algorithm configured.

PARAMETERS

`conditions`	A list of conditions evaluated against the value of `key`. Each condition declares the values that activate it and which algorithms to apply to which column slots.
↳ **`key`**	List of conditional-column values that activate this condition. E.g. `["F", "Female"]` activates for either.
↳ **`string1…string10`**	Algorithm applied to column `stringN` when the condition activates. E.g. `{"name": "dlpx-core:FirstName"}`. Slots without an algorithm pass through untouched.
↳ **`numeric1…numeric3`**	Algorithm applied to the numeric columns (`BigDecimal`).
↳ **`date1…date3`**	Algorithm applied to the date columns (`LocalDateTime`).
↳ **`binary1…binary3`**	Algorithm applied to the binary columns (`ByteBuffer`).
`fallbackAlgo`	Algorithm applied to the string columns when *no* condition matches. Without it, the values pass through unmasked.
`fallbackKey`	Value assumed as the key when the `key` column is null or absent from the row.
`keyCaseSensitive`	When true, comparing `key` against the condition lists distinguishes upper from lower case. Default: false.
`filterLength`	Uses only the first (or last) *N* characters of `key` in the comparison. Useful when the conditional column has a meaningful prefix and a variable suffix.
`filterDirection`	Where to count `filterLength` characters from: `LEFT` (the start) or `RIGHT` (the end).

> **Where the names `key`, `string1` … come from.** They are **fixed slots** of the algorithm, not column names from your database. The translation between the real column (`GENDER`, `NAME`) and the slot (`key`, `string1`) is done in the Masking Engine **inventory**. In the Algorithm Tester you simulate that mapping by filling in the input table by hand.

> **Columns without an algorithm are not masked.** If a condition declares only `string1`, every other column in the row passes through intact — even if it holds sensitive data. Configure `fallbackAlgo` or declare each slot explicitly.

8.2 Multi-Column Address

`algorithm.plugin.address.MultiColumnAddress` **REQUIRES ADDRESS FILE**

Masks address fields spread across several columns — street, number, district, city, state, postal code — generating a synthetic address that is **internally coherent**. Without it, masking each column in isolation would produce impossible combinations (a California street with a New York ZIP code).

INPUT FORMAT

> The row's address columns. Requires an address lookup file in Delphix's specific format.

INPUT → OUTPUT

INPUT		OUTPUT
`123 Flower Street`	→	`Error: null FileReference — address file not configured`

PARAMETERS

Not documented in this guide — the algorithm does not initialize without the address file, which prevents inspecting its schema in the runner.

> **Not testable in the standalone runner.** The algorithm fails during initialization without the address lookup file, which does not ship with the plugin. Configure it and validate directly in the Masking Engine.

PART 9

Other

Two specialized algorithms: generic check-digit recalculation and reversible tokenization.

9.1 Check Digit

`algorithm.plugin.checkdigit.Checkdigit` **DETERMINISTIC**

Masks numbers carrying a check digit and **recalculates that digit** afterwards, so the result still passes validation. It is generic: any weighted-sum-plus-modulus scheme can be described through its parameters — barcode, EAN, payment slip, registration number, state tax ID.

INPUT FORMAT

> A numeric string without punctuation, including the check digit at the configured position. E.g. `7891000315507` (EAN-13).

INPUT → OUTPUT

INPUT		OUTPUT
123456789012	→	**570382872686**

PARAMETERS

weightList *	Weights applied to each digit in the weighted sum, one per data digit. E.g. EAN-13 uses `[1,3,1,3,1,3,1,3,1,3,1,3]`. The list length must equal `numDigitsForCheckdigitCalculation`.
modulusNumber *	The modulus divisor. The check digit is normally `(modulus - remainder) % modulus`. EAN-13 uses `10`; CNPJ uses `11`.
checkDigitIndex *	Position of the check digit in the string, counted from `0` at the left. In a 13-digit code with the check digit last, enter `12`.
calculateChecksumRightToLeft *	Direction in which the weights are applied. **true** — right to left (payment slips, CNPJ). **false** — left to right (EAN, UPC).
numDigitsForCheckdigitCalculation *	How many digits enter the sum, excluding the check digit itself. EAN-13 has 12 data digits.
swapModulusForZeroRemainder	When the remainder is zero, uses `modulusNumber` itself as the check digit instead of `0`. Some standards (CNPJ among them) follow this rule.
checksumCalculationType	**STANDARD** — weighted sum modulo N (default). **TFN** — the Australian Tax File Number variant, with its own logic.
numericAlgorithm	Algorithm that shuffles the digits which are not the check digit. Omitted, it uses the built-in numeric Character Mapping.
alphaNumericAlgorithm	Algorithm used when the input contains letters as well as digits. Without it, alphanumeric input may fail depending on `characterHandling`.
fallbackAlgorithm	Algorithm invoked for invalid input when `invalidInputHandling = FALLBACK_MASK`. It receives the whole original value.
preserveRegex	A regex identifying parts to preserve — fixed prefixes, country codes. Matching stretches stay intact and only the remainder is masked.
inputHandlingConfig	Block controlling how the input is handled before masking.
↳ **characterHandling**	**NUMERIC_ONLY** — accepts only `0–9`. **STANDARD** — letters use their ASCII value. **ASCII_VALUE_MINUS_48** — letters use `(ASCII - 48)`, used by standards that interleave letters and digits.
↳ **invalidInputHandling**	**ERROR** — raises an exception (default). **FALLBACK_MASK** — delegates to `fallbackAlgorithm`.

↳ **`shortInputHandling`**	Input shorter than expected: **FALLBACK** delegates, **PAD_LEFT** / **PAD_RIGHT** pad with `padCharacter`.
↳ **`padCharacter`**	Padding character for short input. Usually `0`.
↳ **`trimWhitespace`**	Strips whitespace from both ends before processing. Necessary for padded `CHAR(n)` columns.

9.2 Tokenization

`algorithm.plugin.tokenization.Tokenization` **REVERSIBLE** **⚠ NON-DETERMINISTIC BY DEFAULT**

The only **reversible** algorithm in the set. It encrypts the value with AES and returns a token; with the same key and configuration, the original value can be recovered (in the Tester, via the *Detokenize* button — `REIDENTIFY` mode). It is the choice when the data must return to its original form somewhere in the pipeline.

INPUT FORMAT

> An alphanumeric string. Characters outside the tokenizable set — symbols, spaces, hyphens — trigger the `fallback`.

INPUT → OUTPUT

INPUT		OUTPUT
`4111111111111111`	→	**`ry5PjcFAfRYtmFWLGo7dCW59bs/Ulpuq`**
`ry5PjcFAfRYtmFWLGo7dCW59bs/Ulpuq` (REIDENTIFY)	→	**`4111111111111111`**
`4111-1111-1111-1111` (with hyphens)	→	**`idDL+Gi/KYr7yIm1pCnRjMFKhJvDYarNNyWV`**

The second row demonstrates exact reversal. Note the token is **longer** than the original — it carries the initialization vector.

PARAMETERS

`fallback`	What to do with non-tokenizable characters. **NONE** — raises an error. **CHARACTER_MAPPING** — applies generic masking to those characters and continues.
`ivLength`	Size of the AES initialization vector in bytes (default `8`). **This is the parameter that decides whether the algorithm is deterministic.** With any value above zero a fresh IV is drawn on every call, so the same input under the same key produces a different token each time. With `0` there is no IV and the token is always the same. Larger values increase randomness but consume more space in the result.
`cmCharacterGroups`	Only with `fallback = CHARACTER_MAPPING`. Defines which characters are interchangeable in the fallback. Empty uses the default groups.
`cmMinMaskedPositions`	Only with `fallback = CHARACTER_MAPPING`. Minimum positions the fallback must mask, avoiding tokens where almost nothing changed.

> ⚠ **Non-deterministic with the default configuration.** With `ivLength` above zero — and the default is `8` — every call draws a fresh initialization vector, so the same input under the same key produces a different token each time. They all reverse correctly, but they **cannot preserve joins**: the same value in two tables becomes two different tokens. If you need the same value to always produce the same token, use `ivLength: 0` — the output is then deterministic and still reversible.

> **The token does not preserve the format.** Despite being described as *format-preserving*, the observed result is a Base64 string longer than the input — `4111111111111111` (16 characters) became 32. Size the target column with room to spare, or the job will fail on truncation.

> **The key is the secret.** Anyone holding the key can reverse any token. Treat it with the same rigor as a production key: out of the code, out of the repository, with controlled rotation — and remember that rotating invalidates every token already issued.

APPENDIX A

Cross-cutting concepts

Behaviors that span several algorithms and tend to cause confusion the first time you meet them.

DETERMINISM AND THE ROLE OF THE KEY

Most algorithms are **deterministic**: the same input value, with the same key, always produces the same output. That is what preserves joins — if `John` becomes `Thomas` in the customers table, it becomes `Thomas` in the orders table too.

The key is what makes the result unpredictable to anyone who lacks it. Changing it changes the entire output, as seen in §4.1. Two practical consequences: **the same key must be used across the whole project**, and **changing the key invalidates consistency with already-masked data**.

BEHAVIOR	ALGORITHMS
Deterministic by key	Character Mapping, Numeric Mapping, Segment Mapping, Payment Card, IBAN, Check Digit, Date Shift, Date Replacement, Secure Lookup, Name, Full Name, Email, Phone, Financial ID BR
⚠ **Non-deterministic**	**Shuffle** (§4.6) — always; a different permutation on every run. **Tokenization** (§9.2) — whenever `ivLength` is above zero, and the default is `8`; use `ivLength: 0` for deterministic output. **Secure Lookup** (§7.1) — only with `hashMethod: RANDOMIZE`.
Key-independent	Min/Max (both), Redact, Free Text Redaction, Data Cleansing, Repeat First Digit, Numeric Expression without `seed`

ALGORITHMS THAT MAY MASK NOTHING

Several algorithms silently return the original value under certain conditions. None of them issue a warning — verifying is on you.

ALGORITHM	WHEN DATA PASSES THROUGH INTACT
Min/Max BigDecimal · Date/Time	Whenever the value is already inside the range
Data Cleansing	Value with no match in the lookup file
Redact	`regexRedact` empty, or no pattern matching
Regex Decompose	`fallbackAction: PRESERVE` with a value matching no pattern
Full Name	`firstNameAlgorithmRef` or `lastNameAlgorithmRef` not configured
Multi-Column Condition	Column slot without an algorithm, or no condition matching and no `fallbackAlgo`
Character Mapping	Characters outside every `characterGroups` entry (accents, punctuation)

CHAR(N) COLUMNS AND INVISIBLE PADDING

`CHAR(n)` columns right-pad the value with spaces up to the declared size. An 11-digit CPF in a `CHAR(15)` column reaches the algorithm as `"12345678909    "`. Since `\d` does not match a space, regexes that work in the Tester fail in production with messages like `No pattern matched` or `did not conform to the expected format`.

The fix is to enable the algorithm's trim option: `trimInput` on Regex Decompose, `trimWhitespace` on Check Digit and Data Cleansing, `trimWhitespaceFromInput` on Secure Lookup. The padding is restored in the output.

REFERENCES BETWEEN ALGORITHMS

Many parameters ask for an "algorithm reference" — `firstNameAlgorithmRef`, `actions.algorithm`, `numericAlgorithm`, `fallbackAlgo`. The expected value is `{"name": "<instance>"}`, where the instance can be:

① a name from the built-in catalog, prefixed with `dlpx-core:` (appendix B); ② the name of a test saved in the Algorithm Tester itself; ③ in the Masking Engine, any configured algorithm instance. Non-existent names produce `Sub-algorithm not found`.

APPENDIX B

Built-in instance catalog

The 62 instances shipped with the plugin, usable in any reference parameter. Prefix them with `dlpx-core:` — for example `dlpx-core:FirstName`. Names containing spaces work fine.

- ABARoutingNumber SL
- Age SL
- BR - Financial ID
- BloodType SL
- Brazil CNPJ
- Brazil CPF
- CM Alpha-Numeric
- CM Digits
- CM Numeric
- Countries SL
- Credit Card
- Date Shift Discrete
- Date Shift Fixed
- Date Shift Variable
- Department SL
- Email SL
- Email Unique
- Ethnicity SL
- FirstName
- FullName
- Gender SL
- IBAN
- Japan Corporate Number
- Japan Drivers License
- Japan My Number
- JobTitle SL
- Language SL
- LastName
- Lat_Long Coordinates
- Legacy AccNoLookup
- Legacy AddrLine2Lookup
- Legacy AddrLookup
- Legacy BusinessLegalEntityLookup
- Legacy CommentLookup
- Legacy DrivingLicenseNoLookup
- Legacy DummyHospitalNameLookup
- Legacy EmailLookup
- Legacy FirstNameLookup
- Legacy FullNMLookup
- Legacy LastCommaFirstLookup
- Legacy LastNameLookup
- Legacy RandomValueLookup
- Legacy SchoolNameLookup
- Legacy USStateCodesLookup
- Legacy USStatesLookup
- Legacy UsCitiesLookup
- Legacy UsCountiesLookup
- Legacy WebURLsLookup
- MaritalStatus SL
- MedicalGenericDrug SL
- NationalOrigin SL
- Null Secure Lookup
- Phone US
- Phone Unique
- PoliticalOrientation SL
- Redact Digits-Zero
- ReligiousOrientation SL
- Repeat First Digit
- SexualOrientation SL
- Shuffle
- SwiftCode SL
- TimeRange

> **The `SL` suffix** stands for *Secure Lookup* — pre-configured instances with themed lookup files embedded (job titles, countries, marital status). They are the fastest way to mask categorical columns without assembling your own file.

APPENDIX C

Standalone runner limitations

The Algorithm Tester runs algorithms outside the Masking Engine. Some features depend on the Engine's context and do not work locally.

FEATURE	STATUS
`FileReference`	**Supported.** Use a `file:///absolute/path` URI. Requires `commons-codec` in `lib/`. Files are managed under *Settings → Files*.
`MappingSetReference`	**Not supported.** The *Mapping* algorithm (§7.4) fails with `Mapping set references are not supported` — it depends on a mapping database.
`EmbeddedGeneratorReference`	**Not supported.**
`MaskValueMetadata`	Always returns `null`.
Multi-Column Address (§8.2)	Does not initialize without Delphix's address lookup file.
Null-Safe Secure Lookup (§7.2)	Always returns `null` — it depends on the embedded file resolved by the Engine.

About the outputs in this guide. Every input → output pair was generated by running the algorithms in `AlgorithmRunner` against the `delphix-algorithm-plugin 2026.3.0-SNAPSHOT` plugin. Deterministic algorithms reproduce these exact values given the same key; Shuffle, by its nature, produces a different permutation on every run.