# Package 'LifeInsuranceContracts'

September 1, 2021

**Type** Package

**Version** 0.0.3

**Date** 2021-08-24

**Title** Framework for Traditional Life Insurance Contracts

**Description** R6 classes to model traditional life insurance contracts like annuities, whole life insurances or endowments. Such life insurance contracts provide a guaranteed interest and are not directly linked to the performance of a particular investment vehicle. However, they typically provide (discretionary) profit participation. This package provides a framework to model such contracts in a very generic (cash-flow-based) way and includes modelling profit participation schemes, dynamic increases or more general contract layers, as well as contract changes (like sum increases or premium waivers). All relevant quantities like premium decomposition, reserves and benefits over the whole contract period are calculated and potentially exported to excel. Mortalities are given using the 'MortalityTables' package.

**Author** Reinhold Kainhofer [aut, cre]

**Maintainer** Reinhold Kainhofer <reinhold@kainhofer.com>

**Encoding** UTF-8

**Imports** R6, MortalityTables, objectProperties, lubridate, openxlsx, dplyr, scales, abind, stringr, methods, rlang, kableExtra, pander, tidyr

**License** GPL (>= 2)

**RoxygenNote** 7.1.1

**Collate** 'HelperFunctions.R' 'InsuranceParameters.R' 'ProfitParticipation_Functions.R' 'ProfitParticipation.R' 'InsuranceTarif.R' 'InsuranceContract.R' 'addDataTableWorksheet.R' 'contractGrid.R' 'exportInsuranceContract_xlsx.R' 'showVmGlgExamples.R' 'exportInsuranceContractExample.R'

**Suggests** knitr, rmarkdown, magrittr, tibble, testthat

**VignetteBuilder** knitr

**URL** https://gitlab.open-tools.net/R/r-life-insurance-contracts

**BugReports** https://gitlab.open-tools.net/R/r-life-insurance-contracts/-/issues

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2021-09-01 21:50:09 UTC

# R topics documented:

---

`applyHook` *If* `hook` *is a function, apply it to* `val`*, otherwise return* `val` *unchanged*

---

## Description

If `hook` is a function, apply it to `val`, otherwise return `val` unchanged

## Usage

```
applyHook(hook, val, ...)
```

## Arguments

| | |
|---|---|
| `hook` | (optional) function to apply to `val` and the other parameters |
| `val` | The value to which the hook is applied (ifgiven) |
| `...` | optional parameters passed to the hook function (if it is a function) |

## Examples

```
applyHook(NULL, 3) # returns 3 unchanged
applyHook(function(x) 2*x, 3) # applies the function, returns 6
applyHook(`+`, 3, 1) # returns 4
```

---

`CalculationSingleEnum-class` *Enum to define how much of a contract needs to be calculated automatically.*

---

## Description

Enum to define how much of a contract needs to be calculated automatically.

## Details

When an InsuranceContract object is created, all time series are immediately calculated. However, sometimes, one only needs part of the values, so it would be a waste of resources to calculate e.g. all future reserves and profit participation, if only premiums are of interest.

Possible values are:

- "all"
- "probabilities"
- "cashflows"
- "presentvalues"

- "premiums"
- "absvalues"
- "reserves"
- "premiumcomposition"
- "profitparticipation"
- "history"

---

`contractGrid` *Create a grid of InsuranceContract objects or premiums with each axis representing one varying parameter*

---

## Description

The function `contractGrid` creates a (two- or multi-dimensional) grid of InsuranceContract objects, where each axis represents one of the insurance parameters varying as given in the `axes` argument (as a named list).

The function `contractGridPremium` returns a grid of premiums as requested in the `premium` parameter rather than the full InsuranceContract objects. It is a convenience wrapper around `contractGrid` and is recommended if one is only interested in a grid of one particular value (typically some kind of premium). The function `contractGridPremium` can also be used on an existing `contractGrid`-generated grid of contracts to extract grid of numerical values of the specified premiums. If no contract grid is passed to `contractGridPremium`, `contractGrid` will be called to create it.

## Usage

```
contractGrid(
  axes = list(age = seq(20, 60, 10), policyPeriod = seq(5, 35, 5)),
  YOB = NULL,
  observationYear = NULL,
  ...
)

contractGridPremium(
  contractGrid = NULL,
  premium = "written",
  .fun = function(cntr) {     cntr$Values$premiums[[premium]] },
  ...
)
```

## Arguments

| | |
|---|---|
| `axes` | List of paramters spanning the dimensions of the grid. |
| `YOB` | optional year of bith. If missing, the `observationYear` and the contract's age |

observationYear
: The observation year, for which the grid shall be calculated. If given, the YOB is calculated from it, otherwise the contract's YOB is used

...
: In `contractGrid`: Additional parameters to be passed to `InsuranceContract$new()()`; In `contractGridPremium`: Additional parameters to be passed to `contractGrid`.

contractGrid
: (optional) existing contract grid from which to derive premiums. If not given, contractGrid is called with all parameters, so `...` should contain an `axes` argument in that case.

premium
: The type of premium to derive (key of the `contract$Values$premiums` list.

.fun
: The function to extract the desired premium from a contract object. By default it accesses the premium vector and extracts the type of premium given in the `premium` parameter. One can, however pass any other extractor function to access e.g. reserves, cash flows etc. at any desired time.

## Details

The function `contractGrid` will return the full InsuranceContract objects, so apply can later be used to extract premiums, reserves and other values to display in a grid. For this feature, one can also use the convenience function `contractGridPremium`.

The `axes` list describing the parameters changing along the axes of the resulting grid is internally expanded with `expand.grid()`. The resulting flat list of parameter (together with the fixed parameters passed as `...`) is then passed to the `InsuranceContract$new()()` call to create the corresponding contract object.

To create the human-readable row-/columnnames of the resulting array, the function `makeContractGridDimname()` for each value of the axes, allowing human-readable representations e.g. of a tariff or a mortality table as the dimension label.

## Examples

```
# TODO
```

---

# costs.baseAlpha *Helper function to define base costs with base alpha, but otherwise unchanged costs*

---

## Description

Returns a function that sets base alpha (and Zillmer) costs to the given value, but otherwise uses the full costs defined by the Costs parameter.

## Usage

```
costs.baseAlpha(alpha)
```

### Arguments

| | |
|---|---|
| alpha | The minimum alpha / Zillmer cost that cannot be waived |

### Details

This function can be set as minCosts parameter for a tariff and makes sure that only alpha costs are modified / waived, but no other costs.

---

## costsDisplayTable *Helper function to display all cost definitions in a concise table*

---

### Description

Returns a data.frame with columns

### Usage

```
costsDisplayTable(costs)
```

### Arguments

| | |
|---|---|
| costs | The cost structure to be displayed in a concise table style. |

---

## costValuesAsDF *Convert the multi-dimensional costs array to a data.frame for output to a file*

---

### Description

Convert the cost values array to a tx15 matrix

### Usage

```
costValuesAsDF(costValues)
```

### Arguments

| | |
|---|---|
| costValues | Cost definition data structure |

### Details

Not to be called directly, but implicitly by the InsuranceContract object. Convert the array containing cost values like cashflows, present values, etc. (objects of dimension tx5x3) to a matrix with dimensions (tx15)

## deathBenefit.annuityDecreasing

*Describes the death benefit of a decreasing whole life insurance (after a possible deferall period)*

### Description

The death benefit will be the full sumInsured for the first year after the deferral period and then decrease like an annuity to 0 at the end of the policyPeriod. This can be used with the `deathBenefit` parameter for insurance contracts, but should not be called directly.

### Usage

```
deathBenefit.annuityDecreasing(interest)
```

### Arguments

| | |
|---|---|
| `interest` | The interest rate of the loan, which is underlying the insurance. |

### Details

This function is a mere generator function, which takes the interest rate and generates a function that describes a decreasing annuity.

The generated function has the following parameters:

**len** The desired length of the Cash flow vector (can be shorter than the policyPeriod, if q_x=1 before the end of the contract, e.g. for life-long insurances)

**params** The full parameter set of the insurance contract (including all inherited values from the tariff and the profit participation)

**values** The values calculated from the insurance contract so far

## deathBenefit.linearDecreasing

*Describes the death benefit of a linearly decreasing whole life insurance (after a possible deferall period)*

### Description

The death benefit will be the full sumInsured for the first year after the deferral period and then decrease linearly to 0 at the end of the policyPeriod. This can be used with the `deathBenefit` parameter for insurance contracts, but should not be called directly.

### Usage

```
deathBenefit.linearDecreasing(len, params, values)
```

### Arguments

| | |
|---|---|
| len | The desired length of the Cash flow vector (can be shorter than the policyPeriod, if q_x=1 before the end of the contract, e.g. for life-long insurances) |
| params | The full parameter set of the insurance contract (including all inherited values from the tariff and the profit participation) |
| values | The values calculated from the insurance contract so far |

---

## exportInsuranceContract.xlsx

*Export an insurance act object tocontract (object of class InsuranceContract) to an Excel file*

---

### Description

Export an insurance act object tocontract (object of class InsuranceContract) to an Excel file

### Usage

```
exportInsuranceContract.xlsx(contract, filename)
```

### Arguments

| | |
|---|---|
| contract | The insurance contract to export |
| filename | Target Excel filename for export |

### Details

The function `exportInsuranceContract.xlsx` exports an object of class InsuranceContract to an Excel file. All basic data, as well as the time series of (absolute and unit) cash flows, reserves, premiums, premium composition and all profit participation scenarios are exported to the file in nicely looking tables.

No new calculations are done in this function. It only prints out the values stored in `contract$Values`.

### Examples

```
library("MortalityTables")
mortalityTables.load("Austria_Annuities_AVOe2005R")
# A trivial deferred annuity tariff with no costs:
tariff = InsuranceTarif$new(name = "Test Annuity", type = "annuity", tarif = "Annuity 1A",
    mortalityTable = AVOe2005R.unisex, i=0.01)
contract = InsuranceContract$new(
    tariff,
    age = 35, YOB = 1981,
    policyPeriod = 30, premiumPeriod = 15, deferralPeriod = 15,
    sumInsured = 1000,
    contractClosing = as.Date("2016-10-01")
);
## Not run: exportInsuranceContract.xlsx(contract, "Example_annuity_contract.xlsx")
```

`exportInsuranceContractExample` *Export the example calculations of an insurance contract*

## Description

Export the given contract to excel (full history/timeseries of all cash flows, reserves, premiums, etc.) and to a text file (sample calculation required by the Austrian regulation).

## Usage

```
exportInsuranceContractExample(
  contract,
  prf = 10,
  outdir = ".",
  basename = NULL,
  extraname = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| `contract` | The `InsuranceContract` object to be exported |
| `prf` | The time of the premium waiver |
| `outdir` | The output directory (the file names are not configurable) |
| `basename` | The base output filename (sans .xlsx). If missing, a name of the form 2020-08-01_TARIFNAME_EXTRANAME_RZ0.01_x35_YoB1977_LZ45_PrZ20_VS100000 is used. If given, the main contract without modification will be exported to basename.xlsx, while the example with premium waiver will be exported to basename_PremiumWaiver_t10.xlsx and the text file containing the examples required by the LV-VMGV is exported to basename_VmGlg.txt. |
| `extraname` | If basename is not given, this allows a suffix to distinguish multiple exports. |
| `...` | Further parameters (passed on to `showVmGlgExamples`) |

## Details

Three output files are generated:

- DATE_TARIFF_Example.xlsx: Full history/timeseries
- DATE_TARIFF_Example_PremiumWaiver_t10.xlsx: Full history/timeseries after a premium waiver at the given time `prf`
- DATE_TARIFF_Examples_VmGlg.txt: Example calculation required for the Austrian regulation (LV-VMGLV)

## Examples

```
library("MortalityTables")
mortalityTables.load("Austria_Annuities_AVOe2005R")
# A trivial deferred annuity tariff with no costs:
tariff = InsuranceTarif$new(name="Test Annuity", type="annuity", tarif = "Annuity 1A",
    mortalityTable = AVOe2005R.unisex, i=0.01)
contract = InsuranceContract$new(
    tariff,
    age = 35, YOB = 1981,
    policyPeriod = 30, premiumPeriod = 15, deferralPeriod = 15,
    sumInsured = 1000,
    contractClosing = as.Date("2016-10-01")
);
## Not run: exportInsuranceContractExample(contract, prf = 10)
```

---

`fallbackFields` *Replace all missing values in fields (either missing or NA) with their corresponding values from fallback. Members in fallback that are missing in fields are inserted*

---

## Description

Replace all missing values in fields (either missing or NA) with their corresponding values from fallback. Members in fallback that are missing in fields are inserted

## Usage

```
fallbackFields(fields, valuelist)
```

## Arguments

| | |
|---|---|
| `fields` | existing list |
| `valuelist` | list of fields to replace in `fields`. Only keys that are missing in `fields` are added, no existing fields in `fields` are overwritten |

---

`fillFields` *Overwrite all existing fields in the first argument with values given in valuelist. Members of valuelist that are not yet in fields are ignored. This allows a huge valuelist to be used to fill fields in multiple lists with given structure.*

---

## Description

Overwrite all existing fields in the first argument with values given in valuelist. Members of valuelist that are not yet in fields are ignored. This allows a huge valuelist to be used to fill fields in multiple lists with given structure.

### Usage

```
fillFields(fields, valuelist)
```

### Arguments

| | |
|---|---|
| fields | existing list |
| valuelist | list of fields to replace in fields. Only keys that exist in fields are overwritten, no new fields are added to fields |

---

## fillNAgaps *Replace all* NA *entries of a vector with the previous non-NA value*

---

### Description

Sometimes one has a vector with some gaps (NA) values, which cause problems for several numeric functions. This function fillNAgaps fills these missing values by inserting the last preceeding non-NA-value. Leading NA values (at the start of the vector will not be modified). If the argument firstBack = TRUE, leading NA-values are replaced by the first non-NA value. Trailing NAs are always replaced by the last previous NA-value.

### Usage

```
fillNAgaps(x, firstBack = FALSE)
```

### Arguments

| | |
|---|---|
| x | The vector where NA-values should be filled by repeating the last preceeding non-NA value |
| firstBack | if TRUE, leading NAs are replaced by the first non-NA value in the vector, otherwise leading NAs are left untouched. |

### Details

This code was taken from the R Cookbook: http://www.cookbook-r.com/Manipulating_data/Filling_in_NAs_with_last_non-NA_value/ LICENSE (from that page): The R code is freely available for use without any restrictions. In other words: you may reuse the R code for any purpose (and under any license).

## filterProfitRates *Filter the whole data.frame of profit rates for the given profit classes*

### Description

This is a rather trivial helper function, which just calls `dplyr::filter()`.

### Usage

```
filterProfitRates(rates, classes)
```

### Arguments

| | |
|---|---|
| `rates` | data.frame containing all profit rates for multiple profit classes |
| `classes` | the profit classes, for which rates should be extracted |

## freqCharge *Defines a frequency charge (surcharge for monthly/quarterly/semiannual) premium payments #' Tariffs are typically calculated with yearly premium installments. When premiums are paid more often then one a year (in advance), the insurance receives part of the premium later (or not at all in case of death), so a surcharge for premium payment frequencies higher than yearly is applied to the premium, typically in the form of a percentage of the premium.*

### Description

This function generates the internal data structure to define surcharges for monthly, quarterly and semiannual premium payments. The given surcharges can be either given as percentage points (e.g. 1.5 means 1.5% = 0.015) or as fractions of 1 (i.e. 0.015 also means 1.5% surcharge). The heuristics applied to distinguish percentage points and fractions is that all values larger than 0.1 are understood as percentage points and values 0.1 and lower are understood as fractions of 1. As a consequence, a frequency charge of 10% or more MUST be given as percentage points.

### Usage

```
freqCharge(monthly = 0, quarterly = 0, semiannually = 0, yearly = 0)
```

### Arguments

| | |
|---|---|
| `monthly` | Surcharge for monthly premium payments |
| `quarterly` | Surcharge for quarterly premium payments |
| `semiannually` | Surcharge for semi-annual premium payments |
| `yearly` | Surcharge for yearly premium payments (optiona, default is no surcharge) |

### Details

Currently, the frequency charges are internally represented as a named list, `list("1" = 0,"2" = 0.01,"4" = 0.02,"12" = 0.03)`, but that might change in the future, so it is advised to use this function rather than explicitly using the named list in your code.

---

## head0 *Set all entries of the given vector to 0 up until index 'start'*

---

### Description

Set all entries of the given vector to 0 up until index 'start'

### Usage

```
head0(v, start = 0)
```

### Arguments

| | |
|---|---|
| `v` | the vector to modify |
| `start` | how many leading elements to zero out |

### Value

the vector `v` with the first `start` elements replaced by 0.

### Examples

```
head0(1:10, 3)
```

---

## initializeCosts *Initialize or modify a data structure for the definition of InsuranceTarif costs*

---

### Description

Initialize a cost matrix with dimensions: CostType, Basis, Period, where:

**CostType:** alpha, Zillmer, beta, gamma, gamma_nopremiums, unitcosts

**Basis:** SumInsured, SumPremiums, GrossPremium, NetPremium, Constant

**Period:** once, PremiumPeriod, PremiumFree, PolicyPeriod

This cost structure can then be modified for non-standard costs using the `setCost()` function. The main purpose of this structure is to be passed to InsuranceContract or InsuranceTarif definitions.

## Usage

```
initializeCosts(
  costs,
  alpha,
  Zillmer,
  beta,
  gamma,
  gamma.paidUp,
  gamma.premiumfree,
  gamma.contract,
  gamma.afterdeath,
  gamma.fullcontract,
  unitcosts,
  unitcosts.PolicyPeriod
)
```

## Arguments

| | |
|---|---|
| costs | (optional) existing cost structure to duplicate / use as a starting point |
| alpha | Alpha costs (charged once, relative to sum of premiums) |
| Zillmer | Zillmer costs (charged once, relative to sum of premiums) |
| beta | Collection costs (charged on each gross premium, relative to gross premium) |
| gamma | Administration costs while premiums are paid (relative to sum insured) |
| gamma.paidUp | Administration costs for paid-up contracts (relative to sum insured) |
| gamma.premiumfree | Administration costs for planned premium-free period (reltaive to sum insured) |
| gamma.contract | Administration costs for the whole contract period (relative to sum insured) |
| gamma.afterdeath | Administration costs after the insured person has dies (for term-fix insurances) |
| gamma.fullcontract | Administration costs for the full contract period, even if the insured has already dies (for term-fix insurances) |
| unitcosts | Unit costs (absolute monetary amount, during premium period) |
| unitcosts.PolicyPeriod | Unit costs (absolute monetary amount, during full contract period) |

## Examples

```
# empty cost structure (only 0 costs)
initializeCosts()

# the most common cost types can be given in initializeCosts()
initializeCosts(alpha = 0.04, Zillmer = 0.025, beta = 0.05, gamma.contract = 0.001)

# The same cost structure manually
costs.Bsp = initializeCosts();
```

```
costs.Bsp[["alpha", "SumPremiums", "once"]] = 0.04;
costs.Bsp[["Zillmer", "SumPremiums", "once"]] = 0.025;
costs.Bsp[["beta", "GrossPremium", "PremiumPeriod"]] = 0.05;
costs.Bsp[["gamma", "SumInsured", "PolicyPeriod"]] = 0.001;

# The same structure using the setCost() function:
library(magrittr)
costs.Bsp = initializeCosts() %>%
  setCost("alpha", "SumPremiums", "once", 0.04) %>%
  setCost("Zillmer", "SumPremiums", "once", 0.025) %>%
  setCost("beta", "GrossPremium", "PremiumPeriod", 0.05) %>%
  setCost("gamma", "SumInsured", "PolicyPeriod", 0.001)
```

---

`InsuranceContract` *Base Class for Insurance Contracts*

---

## Description

Base Class for Insurance Contracts

Base Class for Insurance Contracts

## Details

R6 class that models a complete, general insurance contract. The corresponding tariff and the profit participation scheme, as well as all other relevant contract parameters (if not defined by the tariff or explicitly overridden by the contract) can be given in the constructor.

## Usage

The typical usage of this class is to simply call `InsuranceContract$new()()`.

All parameters from the InsuranceContract.ParameterDefaults can be passed to the constructor of the class (i.e. the `InsuranceContract$new()()`-call). Parameters not explicitly given, will be taken from the tariff or as a fall-back mechanism from the InsuranceContract.ParameterDefaults defaults.

Immediately upon construction, all premiums, reserves and cash flows for the whole contract period are calculated and can be accessed via the `Values` field of the object.

## Calculation approach: Valuation

The calculation of all contract values is controlled by the function `InsuranceContract$calculateContract()()` (using methods of the InsuranceTarif object) and follows the following logic:

1. First the **contingent (unit) cash flows** and the **transition probbilities** are determined.

2. The **actuarial equivalence principle** states that at time of inception, the (net and gross) premium must be determined in a way that the present value of the future benefits and costs minus the present value of the future premiums must be equal, i.e. in expectation the future premiums ove the whole lifetime of the contract will exactly cover the benefits and costs. Similarly, at all later time steps, the difference between these two present values needs to be reserved (i.e. has already been paid by the customer by previous premiums).
3. This allows the premiums to be calculated by first calculating the **present values** for all of the **benefit and costs cash flow** vectors.
4. The formulas to calculate the gross, Zillmer and net **premiums** involve simple linear combinations of these present values, so the **coefficients of these formulas** are determined next.
5. With the coefficients of the premium formulas calculated, all **premiums can be calculated** (first the gross premium, because due to potential gross premium refunds in case of death, the formula for the net premium requires the gross premium, which the formula for the gross premium involves no other type of premuim).
6. With premiums determined, all unit cash flows and unit present values can now be expressed in monetary terms / as **absolute cash flows** (i.e. the actual Euro-amount that flows rather than a percentage).
7. As described above, the difference between the present values of premiums and present values of benefits and costs is defined as the required amount of reserves, so the **reserves (net, gross, administration cost, balance sheet)** and all values derived from them (i.e. surrender value, sum insured in case of premium waiver, etc.) are calculated.
8. The **decomposition of the premium** into parts dedicated to specific purposes (tax, rebates, net premium, gross premium, Zillmer premium, cost components, risk premium, savings premium, etc.) can be done once the reserves are ready (since e.g. the savings premium is defined as the difference of discounted reserves at times $t$ and $t+1$).
9. If the contract has **(discretionary or obligatory) profit sharing**B mechanisms included, the corresponding ProfitParticipation object can calculate that profit sharing amounts, once all guaranteed values are calculated. This can also be triggered manually (with custom profit sharing rates) by calling the methods `InsuranceContract$profitScenario()()` or `InsuranceContract$addProfitSce`

**Calculation approach: Cash Flows**

An insurance contract is basically defined by the (unit) cash flows it produces:

- **Premium payments** (in advance or in arrears) at each timestep
- **Survival payments** at each timestep
- **Guaranteed payments** at each timestep
- **Death benefits** at each timestep
- **Disease benefits** at each timestep

Together with the transition probabilities (mortalityTable parameter) the present values can be calculated, from which the premiums follow and finally the reserves and a potential profit sharing.

For example, a **term life insurance with regular premiums** would have the following cash flows:

- premium cash flows: 1, 1, 1, 1, 1, ...
- survival cash flows: 0, 0, 0, 0, 0, ...

- guaranteed cash flows: 0, 0, 0, 0, 0, ...
- death benefit cash flows: 1, 1, 1, 1, 1, ...

A **single-premium term life insurance** would look similar, except for the premiums:

- premium cash flows: 1, 0, 0, 0, 0, ...

A **pure endowment** has no death benefits, but a survival benefit of 1 at the maturity of the contract:

- premium cash flows: 1, 1, 1, 1, 1, ...
- survival cash flows: 0, 0, ..., 0, 1
- guaranteed cash flows: 0, 0, 0, 0, 0, ...
- death benefit cash flows: 0, 0, 0, 0, 0, ...

An **endowment** has also death benefits during the contract duration:

- premium cash flows: 1, 1, 1, 1, 1, ...
- survival cash flows: 0, 0, ..., 0, 1
- guaranteed cash flows: 0, 0, 0, 0, 0, ...
- death benefit cash flows: 1, 1, 1, 1, 1, ...

A **(deferred) annuity***B* has premium cash flows only during the deferral peroid and only survival cash flows during the annuity payment phase. Often, in case of death during the deferral period, all premiums paid are refunded as a death benefit.:

- premium cash flows: 1, 1, ..., 1, 0, 0, 0, ...
- survival cash flows: 0, 0, ..., 0, 1, 1, 1,...
- guaranteed cash flows: 0, 0, 0, 0, 0, ...
- death benefit cash flows: 1, 2, 3, 4, 5, ..., 0, 0, ...

A **terme-fix insurance** has a guaranteed payment at maturity, even if the insured has already died. The premiums, however, are only paid until death (which is not reflected in the contingent cash flows, but rather in the transition probabilities):

- premium cash flows: 1, 1, 1, 1, ..., 1
- survival cash flows: 0, 0, 0, 0, ..., 0
- guaranteed cash flows: 0, 0, 0, ..., 0, 1
- death benefit cash flows: 0, 0, 0, 0, ..., 0

The `InsuranceContract$new()` function creates a new insurance contract for the given tariff, using the parameters passed to the function (and the defaults specified in the tariff).

As soon as this function is called, the contract object calculates all time series (cash flows, premiums, reserves, profit participation) for the whole contract duration.

The most important parameters that are typically passed to the constructor are:

- `age` ... Age of the insured person (used to derive mortalities / transition probabilities)
- `policyPeriod` ... Maturity of the policy (in years)

- premiumPeriod ... How long premiums are paid (premiumPeriod = 1 for single-premium contracts, premiumPeriod equals policyPeriod for regular premium payments for the whole contract period, while other premium payment durations indicate premium payments only for shorter periods than the whole contract duration). Default is equal to policyPeriod
- sumInsured ... The sum insured (i.e. survival benefit for endowments, death benefit for whole/term life insurances, annuity payments for annuities)
- contractClosing ... Date of the contract beginning (typically created using something like as.Date("2020-08-01"))
- YOB ... Year of birth of the insured (for cohort mortality tables). If not given, YOB is derived from age and contractClosing.
- deferralPeriod ... Deferral period for deferred annuities (i.e. when annuity payments start at a future point in time). Default is 0.
- premiumFrequency ... How many premium payments per year are made (e.g. 1 for yearly premiums, 4 for quarterly premiumd, 12 for monthly premium payments). Default is 1 (yearly premiums).

While these are the most common and most important parameters, all parameters can be overwritten on a per-contract basis, even those that are defined by the tariff. For a full list and explanation of all parameters, see InsuranceContract.ParameterDefaults.

The InsuranceContract$addHistorySnapshot() function adds the current (or the explicitly given) state of the contract (parameters, calculated values, tarif, list of all contract blocks) to the history list (available in the history field of the contract, i.e. InsuranceContract$history).

Contracts with multiple contract blocks (typically either contracts with dynamic increases, sum increases or protection riders) are constructed by instantiating the child block (e.g. a single dynamic increase or the rider) independently with its own (shorter) duration and then inserting it into the parent contract with this function at the given time.

If no InsuranceContract object is passed as block, a copy of the parent is created with overriding parameters given in ....

This method adds a new contract block describing a dynamic or sum increase (increasing the sum insured at a later time $t$ than contract inception). This increase is modelled by a separate InsuranceContract object with the sum difference as its own sumInsured.

```
By default, all parameters are taken from the main contract,
with the maturity adjusted to match the original contract's
maturity.

The main contract holds all child blocks, controls their
valueation and aggregates all children's values to the
total values of the overall contract.
```

This method calculates all contract values (potentially starting from and preserving all values before a later time valuesFrom). This function is not meant to be called directly, but internally, whenever a contract is created or modified.

```
There is, hoever, a legitimate case to call this function
when a contract was initially created with a value of
```

```
\code{calculate} other than "all", so not all values of the
contract were calculated. When one later needs more values
than were initially calculated, this function can be called.
However, any contract changes might need to be rolled back
and reapplied again afterwards. So even in this case it is
probably easier to create the contract object from scratch
again.
```

This function is an internal function for contracts with multiple child blocks (dynamic increases, sum increases, riders). It takes the values from all child blocks and calculates the overall values from all child blocks aggregated.

```
This function should not be called manually.
```

This function modifies the contract at time $t$ so that no further premiums are paid (i.e. a paid-up contract) and the `sumInsured` is adjusted according to the existing reserves.

This function calculates one profit scenario with the provided profit participation parameters (all parameters not given in the call are taken from their values of the contract, profit participation scheme or tariff).

This function calculates one profit scenario with the provided profit participation parameters (all parameters not given in the call are taken from their values of the contract, profit participation scheme or tariff). The results are stored in a list of profit scenarios inside the contract.

```
This function can be chained to calculate and add multiple
profit scenarios.
```

## Public fields

`tarif` The InsuranceTarif underlying this contract. The tarif is the abstract product description (i.e. defining the type of insurance, fixing tpyes of benefits, specifying costs, guaranteed interest rate, mortality tables, potential profit sharing mechanisms, etc.), while the contract holds the individual parts like age, sum insured, contract duration, premium payment frequency, etc.

`parent` A pointer to the parent contract. Some contracts consist of multiple parts (e.g. a main savings contract with a dread-disease rider, or a contract with multiple dynamic increases). These are internally represented by one InsuranceContract object per contract part, plus one contract object combining them and deriving combined premiums, reserves and profit participation. The child contracts (i.e. the objects representing the individual parts) have a pointer to their parent, while the overall contract holds a list of all its child contract parts.

`ContractParameters` Insurance contract parameters explicitly specified in the contract (i.e. parameters that are NOT taken from the tariff of the defaults).

`Parameters` Full set of insurance contract parameters applying to this contract. The set of parameters is a combination of explicitly given (contract-specific) values, parameters determined by the tariff and default values.

`Values` List of all contract values (cash flows, present values, premiums, reserves, premium decomposition, profit participation, etc.). These values will be calculated and filled when the contract is created and updated whenever the contract is changed.

`blocks` For contracts with multiple contract parts: List of all tariff blocks (independently calculated InsuranceContract objects, that are combined to one contract, e.g. dynamic/sum increases). If this field is empty, this object describes a contract block (calculated as a stand-alone tariff), otherwise it will simply be the sum of its blocks (adjusted to span the same time periods)

`history` A list keeping track of all contract changes (including the whole contract state and its values before the change).

`dummy.public` dummy field to allow a trailing comma after the previous field/method

## Methods

### Public methods:

- `InsuranceContract$new()`
- `InsuranceContract$addHistorySnapshot()`
- `InsuranceContract$addBlock()`
- `InsuranceContract$addDynamics()`
- `InsuranceContract$calculateContract()`
- `InsuranceContract$consolidateBlocks()`
- `InsuranceContract$premiumWaiver()`
- `InsuranceContract$profitScenario()`
- `InsuranceContract$addProfitScenario()`
- `InsuranceContract$clone()`

**Method** `new()`**:** Create a new insurance contract (for the given tariff/product) and calculate all time series

*Usage:*

```
InsuranceContract$new(
  tarif,
  parent = NULL,
  calculate = "all",
  profitid = "default",
  ...
)
```

*Arguments:*

`tarif` The InsuranceTarif object describing the Tariff/Product and providing defaults for the parameters.

`parent` For contracts with multiple contract blocks (dynamic increases, sum increases, riders), each child is created with a pointer to its parent. NULL for single-block contracts or for the overall-contract of a multi-block contract. This parameter is used internally, but should not be used in user-written code.

`calculate` how much of the contract's time series need to be calculated. See CalculationEnum for all possible values. This is usefull to prevent calculation of e.g. reserves and profit participation, when one only wants to create a grid of premiums.

`profitid` The ID of the default profit participation scenario. The default profit participation scenario uses the default values passed, while further scenarios can be added by `InsuranceContract$addProfitSce`

... Further parameters (age, sum insured, contract closing / begin, premium payment details, etc.) of the contract, which can also override parameters defined at the tariff-level. Possible values are all sub-fields of the InsuranceContract.ParameterDefaults data structure.

**Method** `addHistorySnapshot()`**:** Add the current state of the contract to the history list

*Usage:*

```
InsuranceContract$addHistorySnapshot(
  time = 0,
  comment = "Initial contract values",
  type = "Contract",
  params = self$Parameters,
  values = self$Values,
  tarif = self$tarif,
  blocks = self$blocks
)
```

*Arguments:*

`time` the time described by the snapshot

`comment` a comment to store together with the contract state

`type` The type of action that caused a history snapshot to be stored. Typical values are "Contract" to describe the initial contract, "Premium Waiver" or "Dynamic Increase".

`params` The set of params to be stored in the history snapshot (default is `self$Parameters`, if not explicitly given)

`values` The calculated time series of all contract values calculated so far. Default is `self$Values`, if not explicitly given

`tarif` The underlying InsuranceTarif object describing the Product/Tariff. Default is `self$tarif`, if not explicitly given.

`blocks` The list of all contract children for contracts with multiple insurance blocks (e.g. dynamic increases, riders, etc.)

*Examples:*

```
# TODO
```

**Method** `addBlock()`**:** Add a child contract block (e.g. a dynamic increase or a rider) to an insurance contract

*Usage:*

```
InsuranceContract$addBlock(
  id = NULL,
  block = NULL,
  t = block$Values$int$blockStart,
  comment = comment,
  ...
)
```

*Arguments:*

`id` The identifier of the child block to be inserted

`block` The InsuranceContract object describing the child block. If NULL (or not given at all), a copy of the parent will be created.

`t` Then the child block starts, relative to the parent block. The child block is calculated independently (with time 0 describing its own start), so when aggregating all values from the individual blocks to overall values for the whole contract, the child's values need to be translated to the parent contracts's time frame using this parameter

`comment` The comment to use in the history snapshot.

`...` parameters to be passed to `InsuranceContract$new()()` when `block` is not given and a copy of the parent should be created with overrides.

*Examples:*

```
# TODO
```

**Method** `addDynamics()`**:** Add a dynamic increase with the same parameters as the main contract part

*Usage:*

```
InsuranceContract$addDynamics(t, NewSumInsured, SumInsuredDelta, id, ...)
```

*Arguments:*

`t` The time within the main contract when the sum increase happens. The InsuranceContract object describing the dynamic increase will still internally start at its own time 0, but the aggregation by the main contract will correctly offset to the time $t$ within the main contract.

`NewSumInsured` The over-all new sum insured (sum of original contract and all dynamica increaeses). The `sumInsured` of the new dynamic increase block will be determined as the difference of the old and new overall sum insured. Alternatively, it can directly be given as the `SumInsuredDelta` argument instead.

`SumInsuredDelta` The sum insured of only the dynamic increase, i.e. the sumInsured of the dynamic contract block only. The overall sum insured will increase by this amount. Only one of `NewSumInsured` and `SumInsuredDelta` is needed, the other one will be calculated accordingly. If both are given, the `SumInsuredDelta` will take precedence.

`id` The identifier of the contract block describing the dynamic increase. This is a free-form string that should be unique within the list of child blocks. It will be displayed in the Excel export feature and in the history snapshot list.

`...` Paramters to override in the dynamic block. By default, all parameters of the main contract block will be used, but they can be overridden per dynamic increase block.

*Examples:*

```
# TODO
```

**Method** `calculateContract()`**:** Calculate all time series of the contract from the parameters

*Usage:*

```
InsuranceContract$calculateContract(
  calculate = "all",
  valuesFrom = 0,
  premiumCalculationTime = 0,
  preservePastPV = TRUE,
  additionalCapital = 0,
  recalculatePremiums = TRUE,
  recalculatePremiumSum = TRUE,
  history_comment = NULL,
  history_type = "Contract"
)
```

*Arguments:*

`calculate` Which values to calculate. See CalculationEnum

`valuesFrom` Calculate only values starting from this time step on (all values before that time will be preserved). This is required when a contract is changed significantly (potentially even switching to a new tariff), so that the calculation bases for previous periods are no longer available.

`premiumCalculationTime` The time point when the premium should be re-calculated (including existing reserves) based on the actuarial equivalence principle. All reserves will be based on these new premiums.

`preservePastPV` Whether present value before the recalculation time `valuesFrom` should be preserved or recalculated. When they are recalculated, the present values are consistent to the new cash flows over the whole contract period, but they no longer represent the actual contract state at these times. If values are not recalculated, the reserves at each time step represent the proper state at that point in time.

`additionalCapital` The capital that is added to the contract (e.g. capital carried over from a previous contract) at the premium calculation time.

`recalculatePremiums` Whether the premiums should be recalculated at time `premiumCalculationTime` at all.

`recalculatePremiumSum` Whether to recalculate the overall premium sum when the premium is recalculated.

`history_comment` The comment for the history snapshot entyr

`history_type` The type (free-form string) to record in the history snapshot

**Method** `consolidateBlocks()`**:** Aggregate values from all child contract blocks (if any)

*Usage:*

```
InsuranceContract$consolidateBlocks(valuesFrom = 0)
```

*Arguments:*

`valuesFrom` The time from when to aggragate values. Values before that time will be left unchanged.

**Method** `premiumWaiver()`**:** Stop premium payments and re-calculate sumInsured of the paid-up contract

*Usage:*

```
InsuranceContract$premiumWaiver(t)
```

*Arguments:*

`t` Time of the premium waiver.

*Examples:*

```
# TODO
```

**Method** `profitScenario()`**:** Calculate one profit scenario and return all values

*Usage:*

```
InsuranceContract$profitScenario(...)
```

*Arguments:*

... Scenario-specific profit sharing parameters, overriding the default values. Typically, adjusted profit rates are required in a profitScenario.

*Returns:* a data.frame holding all profit participation values (rates, bases for the different profit types, profit allocations, terminal bonus funds, profit in case of death/surrender/premium waiver)

*Examples:*

```
# TODO
```

**Method** `addProfitScenario()`**:** Calculate one profit scenario and store it in the contract

*Usage:*

```
InsuranceContract$addProfitScenario(id, ...)
```

*Arguments:*

`id` The unique ID of the profit scenario. Will be used as key in the list of profit scenarios and printed out in the Excel export.

... Scenario-specific profit sharing parameters, overriding the default values. Typically, adjusted profit rates are required in a profitScenario.

*Examples:*

```
# TODO
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
InsuranceContract$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
# TODO


## ------------------------------------------------
## Method `InsuranceContract$addHistorySnapshot`
## ------------------------------------------------

# TODO

## ------------------------------------------------
## Method `InsuranceContract$addBlock`
## ------------------------------------------------

# TODO

## ------------------------------------------------
## Method `InsuranceContract$addDynamics`
## ------------------------------------------------

# TODO
```

```
## ------------------------------------------------
## Method `InsuranceContract$premiumWaiver`
## ------------------------------------------------

# TODO

## ------------------------------------------------
## Method `InsuranceContract$profitScenario`
## ------------------------------------------------

# TODO

## ------------------------------------------------
## Method `InsuranceContract$addProfitScenario`
## ------------------------------------------------

# TODO
```

---

`InsuranceContract.ParameterDefaults`

*Default parameters for the InsuranceContract class. A new contract will be pre-filled with these values, and values passed in the constructor (or with other setter functions) will override these values.*

---

## Description

Default parameters for the InsuranceContract class. A new contract will be pre-filled with these values, and values passed in the constructor (or with other setter functions) will override these values.

## Usage

```
InsuranceContract.ParameterDefaults
```

## Format

The parameter list is a list of lists with the following structure:

Sublists:

- `$ContractData` ... Contract-specific data (policy period, closing, age, sum insured, premium payments, etc.)
- `$ContractState` ... Current contract state (paid-up, surrender penalty already applied, alpha costs already (partially) refunded)
- `$ActuarialBases` ... Actuarial bases for the contract calculation (mortality/invalidity table, guaranteed interest, surrender penalty, etc.)
- `$Costs`, `$minCosts` ... Expenses charged to the contract (see `initializeCosts()`)
- `$Loadings` ... Loadings, rebates and other charges of the tariff / contract (tax, unit costs, surcharge for no medial exam, premium/benefit frequency loading)

- `$Features` ... Peculiarities of the tariff (to enable non-standard formulas for certain company-specific historical "glitches" in the tariff definitions.)
- `$ProfitParticipation` ... Profit scheme and profit participation rates (default values, can be overwritten per profit scenario)
- `$Hooks` ... Hook functions to allow modification of various calculation aspects (e.g. modify the default cash flows after their setup)

**Elements of sublist** `InsuranceContract.ParameterDefault$ContractData`**:**

These values are typically set per contract and not by the tariff. Notable exceptions are the contract duration in some instances and the premiumPeriod=1 for single-premium contracts.

`$id` ID of the contract (to distinguish individual parts in contracts with multiple parts, e.g. dynamic increases), default = "Hauptvertrag"

`$sumInsured` Sum insured, default = 100,000

`$initialCapital` Reserve/Capital that is already available at contract inception, e.g. from a previous contract. No tax or acquisition costs are applied to this capital.

`$YOB` Year of birth of the insured, used to determine the age for the application of the mortality table

`$age` Age of the insured

`$technicalAge` Technical age of the insured (when the age for the application of the mortality table does not coincide with the real age)

`$ageDifferences` Vector of age differences to the first insured for contracts with multiple insured (i.e. joint-lives)

`$sex` Sex of the insured, to allow gender-specific prixing (e.g. different mortalities or age modification), default="unisex", Type is SexEnum

`$policyPeriod` Policy Duration (in years)

`$premiumPeriod` Premium payment period (in year), for single-premium contracts, `premiumPeriod = 1`. Default is `policyPeriod`, i.e. regular premiums during the whole contract period

`$deferralPeriod` deferral period for annuities, i.e. the period survival payments start only after this period, typically the retirement age. This applies mostly to tariffs of type annuity, although deferral periods are possible (but not common) for all other types of insurance, too.

`$guaranteedPeriod` guaranteed annuity payment period. The annuity pays out for this period, even if the insured dies. This applies only to tariffs of type annuity.

`$contractClosing` The date (variable of type Date) when the coverage of the contract starts (not neccessarily equal to the date when the contract was signed). Typically generated by a call to `as.Date()`. The year is relevant to derive the age of the insured, while month and day are relevant for the interpolation of the balance sheet reserves

`$blockStart` For contracts with multiple blocks (e.g. multiple dynamic increases, where each increase is modelled like a separate contract), this variable holds the offset of the current contract block relative to the main contract block. The main block starts a 0, dynamic increases start later! This value is only used by the parent block (i.e. $t=0$ of the child is aligned with $t=blockStart$ of the parent block.

`$premiumPayments` Whether premiums are paid in advance (default) or arrears. Value is of type PaymentTimeEnum with possible values "in advance" and "in arrears"

`$benefitPayments` Whether recurring benefits (e.g. annuities) are paid in advance (default) or arrears. Value is of type PaymentTimeEnum with possible values "in advance" and "in arrears"

`$premiumFrequency` Number of premium payments per year, default is 1.

`$benefitFrequency` Number of benefit payments per year, default is 1.

`$widowProportion` For annuities with a widow transition, this describes the factor of the widow benefits relative to the original benefit.

`$deathBenefitProportion` For endowments with a death and survival benefit, this describes the proportion of the death benefit relative to the survival benefit.

`$premiumRefund` Proportion of (gross) premiums refunded on death (including additional risk, e.g. 1.10 = 110% of paid premiums)

`$premiumIncrease` The yearly growth factor of the premium, i.e. 1.05 means +5% increase each year; a vector describes the premiums for all years

`$annuityIncrease` The yearly growth factor of the annuity payments, i.e. 1.05 means +5% increase each year; a vector describes the annuity unit payments for all years

`$deathBenefit` The yearly relative death benefit (relative to the initial sum insured); Can be set to a `function(len,params,values)`, e.g. `deathBenefit = deathBenefit.linearDecreasing`

`$costWaiver` The fraction of the costs that are waived (only those cost components that are defined to be waivable, i.e. by defining a corresponding `$minCosts`). Linearly interpolates between `$Costs` and `$minCosts`, if the latter is set. Otherwise is has no effect.

**Elements of sublist** `InsuranceContract.ParameterDefault$ContractState`**:**

Contract-specific status variables holding the status of the contract.

`$premiumWaiver` Whether the contract is paid-up.

`$surrenderPenalty` Whether a surrender penalty still applies (e.g. because it has already been applied during a contract change, or because due to legal reasons it can no longer be applied)

`$alphaRefunded` Whether alpha costs have (at least partially) been refunded (e.g. when a contract is changed or paid-up). Default is not yet refunded.

**Elements of sublist** `InsuranceContract.ParameterDefault$ActuarialBases`**:**

Tarif-specific actuarial calculation parameters of the contract. Typically, these values are set by the tariff, but can be overridden by contract (e.g. while prototyping a new product or a product change).

`$mortalityTable` The mortalityTable object describing the mortality of the insured

`$invalidityTable` For contracts with invalidity benefits, the mortalityTable object describing the probabilities of invalidity

`$invalidityEndsContract` For contracts with invalidity benefits, whether a payment of an invalidity benefit ends the contract.

`$i` Guaranteed yearly interest rate, default is 0.00, i.e. 0%

`$balanceSheetDate` The day/month when balance sheet reserves are calculated. Value of type Date, typically generated with `as.Date()`. The year is actually irrelevant, only the day and month are relevant.

`$balanceSheetMethod` How to interpolate the balance sheet reserves (at the balandeSheetDate) from the yearly contractual reserves. Either a string "30/360", "act/act", "act/360", "act/365" or a function with signature `balanceSheetMethod(params,contractDates,balanceDates)` that returns a vector of coefficients for each year to interpolate the reserves available at the given `contractDates` for the desired `balanceDates`

`$unearnedPremiumsMethod` How to calculate the unearned premiums (considering the balance sheet date and the premium frequency). A function with signature `unearnedPremiumsMethod(params,dates)`

`$surrenderValueCalculation` A function describing the surrender value calculation. If NULL, the full reserve will be used as surrender value. If given, it must be a function with signature `function(SurrenderReserve,params,values)`.

`$premiumWaiverValueCalculation` A function describing the reserve used to derive the premium-free sum insured. If NULL, the surrender value will be used. If given, it must be a function with signature `function(SurrenderReserve,params,values)`

`$premiumFrequencyOrder` Order of the approximation for payments within the year (unless an extra frequency loading is used => then leave this at 0)

`$benefitFrequencyOrder` Order of the approximation for payments within the year (unless an extra frequency loading is used => then leave this at 0)

**Elements of sublist** `InsuranceContract.ParameterDefault$Costs`**:**

Definition of contractual costs charged to the contract. See `initializeCosts()`.

`$Costs` The full cost defined for the contract / tariff, usually set with `initializeCosts()` and `setCost()`

`$minCosts` The minimum costs defined for the contract / tariff that cannot be waived. Either an explicit cost definition structure generated by `initializeCosts()` and `setCost()`, or a function(params, values, costs), where the full costs are passed as third parameter, so the function can modify only those cost parts that can be waived at all.

**Elements of sublist** `InsuranceContract.ParameterDefault$Loadings`**:**

`$ongoingAlphaGrossPremium` Acquisition cost that increase the gross premium

`$tax` insurance tax, factor on each premium paid, default is 4%, i.e. `i=0.04`

`$unitcosts` Annual unit cost for each policy, absolute value (can be a function)

`$security` Additional security loading on all benefit payments, factor on all benefits

`$noMedicalExam` Loading when no medicial exam is done, % of SumInsured

`$noMedicalExamRelative` Loading when no medicial exam is done, % of gross premium

`$sumRebate` gross premium reduction for large premiums, % of SumInsured

`$extraRebate` gross premium reduction for any reason, % of SumInsured

`$premiumRebate` gross premium reduction for large premiums, % of gross premium

`$partnerRebate` Rebate on premium with all surcharges and rebates when more than one contract is written with identical parameters. Sums with advanceBonusInclUnitCost and premiumRebate.

`$extraChargeGrossPremium` extra charges on gross premium (smoker, leisure activities, BMI too high, etc.)

`$benefitFrequencyLoading` Loading on the benefit for premium payment frequencies of more than once a year. Format is `list("1" = 0.0,"2" = 0.0,"4" = 0.0,"12" = 0.0)`

`$premiumFrequencyLoading` Loading on the premium for premium payment frequencies of more than once a year. Format is `list("1" = 0.0,"2" = 0.0,"4" = 0.0,"12" = 0.0)`

`$alphaRefundPeriod` How long the acquisition costs should be (partially) refunded in case of surrender or premium waiver.

**Elements of sublist** `InsuranceContract.ParameterDefault$Features`**:**

`$betaGammaInZillmer` Whether beta and gamma-costs should be included in the Zillmer premium calculation

`$alphaRefundLinear` Whether the refund of alpha-costs on surrender is linear in t or follows the NPV of an annuity

`$useUnearnedPremiums` Whether unearned premiums should be reported in the balance sheet reserves. Otherwise, a premium paid at the beginning of the period is added to the reserve at that time for balance-sheet purposes. For regular premiums, the default is TRUE, i.e. the balance-sheet reserve at time $t$ does not include the premium paid at time $t$, but unearned premiums are included in the balance sheet reserves. For single-premium contracts, there are no "unearned" premiums, but the initial single premium is added to the reserve immediately for balance-sheet purposes. In particular, the balance sheet reserve at time $t=0$ is not 0, but the premium paid. In turn, no unearned premiums are applied.

**Elements of sublist** `InsuranceContract.ParameterDefault$ProfitParticipation`**:**

Parameters describing the profit participation (instance of ProfitParticipation) Most element descrive some kind of profit rate (which can vary in time), while the bases, on which they are applied is defined in the profit scheme.

`$advanceProfitParticipation` Advance profit participation rate (percentage rebate of the gross premium)

`$advanceProfitParticipationInclUnitCost` Advance profit participation rate (percentage rebate on the gross premium after all surcharges and unit costs.

`$waitingPeriod` Waiting period of the profit sharing (e.g. no profit in the first two years of a contract, or similar)

`$guaranteedInterest` Individual contract-specific overrides of the guaranteed interest rate (i.e. not keyed by year)

`$interestProfitRate` Interest profit rate (guaranteed interest rate + interest profit rate = total credited rate)

`$totalInterest` Total credited rate (guarantee + interest profit)

`$mortalityProfitRate` Mortality Profit rate

`$expenseProfitRate` Expense profit rate

`$sumProfitRate` Sum profit rate (for high sumInsured)

`$terminalBonusRate` Terminal bonus rate (non-terminal-bonus fund, but "old" Austrian terminal bonus)

`$terminalBonusFundRate` Terminal bonus fund rate

`$profitParticipationScheme` Profit participation scheme (object of class ProfitParticipation)

`$profitComponents` Profit components of the profit scheme. List containing one or more of `c("interest","risk","expense","sum","terminal")`

`$profitClass` String describing the profit class the tariff is assigned to. Profit classes are used to bundle similar contracts (e.g. following similar risks) together. Profit participation rates are defined at the level of profit classes.

`$profitRates` General, company-wide profit rates, key columns are year and profitClass

`$scenarios` profit participation scenarios (list of overridden parameters for each scenario)

**Elements of sublist** `InsuranceContract.ParameterDefault$Hooks`**:**

`$adjustCashFlows` Function with signature `function(x,params,values,...)` to adjust the benefit/premium cash flows after their setup.

`$adjustCashFlowsCosts` Function with signature `function(x,params,values,...)` to adjust the costs cash flows after their setup.

`$adjustPremiumCoefficients` Function with signature `function(coeff,type,premiums,params,values,premiumCa` to adjust the coefficients for premium calculation after their default setup. Use cases are e.g. term-fix tariffs where the Zillmer premium term contains the administration cost over the whole contract, but not other gamma- or beta-costs.

## Examples

```
InsuranceContract.ParameterDefaults
```

---

`InsuranceContract.ParametersFallback` *InsuranceContract.ParametersFallback*

---

## Description

Provide default values for the insurance contract parameters if any of the parameters is not explicitly set.

## Usage

```
InsuranceContract.ParametersFallback(params, fallback, ppParameters = TRUE)
```

## Arguments

| | |
|---|---|
| `params` | Current, explicitly set contract parameters. All NULL values will be filled with the corresponding entry from `fallback`. |
| `fallback` | Fallback values that will be used when the corresponding entry in `params` is NULL. |
| `ppParameters` | Whether profit participation parameters should also be filled (default is TRUE) |

---

`InsuranceContract.ParametersFill` *InsuranceContract.ParametersFill*

---

## Description

Initialize the insurance contract parameters from the passed arguments. Arguments not given are left unchanged. If no existing parameter structure is given, an empty (i.e. all NULL entries) structure is used.

## Usage

```
InsuranceContract.ParametersFill(
  params = InsuranceContract.ParameterStructure,
  costs = NULL,
  minCosts = NULL,
  ...
)
```

### Arguments

| | |
|---|---|
| `params` | Initial values of the insurance contract parameters. (default: empty parameter structure) |
| `costs, minCosts, ...` | Values for any of the entries in the insurance contract parameter structure. These values take precedence over the initial parameters provided in `params`. |

---

## `InsuranceContract.ParameterStructure` *Full insurance contract parameter structure.*

---

### Description

All values are filled with NULL, so the functions `InsuranceContract.ParametersFill` and `InsuranceContract.Paramet` can be used to override existing parameters or to provide default values for unset (NULL) entries.

### Usage

```
InsuranceContract.ParameterStructure
```

### Format

An object of class `list` of length 9.

---

## `InsuranceContract.Values` *Data structure (filled only with NULL) for insurance contract class member values.*

---

### Description

Data structure (filled only with NULL) for insurance contract class member values.

### Usage

```
InsuranceContract.Values
```

### Format

An object of class `list` of length 16.

`InsuranceTarif` *Base class for traditional Insurance Tarifs (with fixed guarantee, profit sharing and no unit-linked component)*

## Description

The class `InsuranceTarif` provides the code and general framework to implement contract-independent functionality of a life insurance product.

## Details

This is a base class for holding contract-independent values and providing methods to calculate cash flows, premiums, etc. Objects of this class do NOT contain contract-specific values like age, death probabilities, premiums, reserves, etc. Rather, they are the calculation kernels that will be called by the `InsuranceContract` objects to make the actual, tariff-specific calculations.

Most methods of this class are not meant to be called manually, but are supposed to be called by the InsuranceContract object with contract-specific information. The only methods that are typically used for defining an insurance tariff are the constructor `InsuranceTarif$new()` and the cloning method `InsuranceTarif$createModification()`. All other methods should never be called manually.

However, as overriding private methods is not possible in an R6 class, all the methods need to be public to allow overriding them in derived classes.

## Public fields

`name` The tariff's unique name. Will also be used as the key for exported data.

`tarif` The tariff's public name (typically a product name), not necessarily unique.

`desc` A short human-readable description of the tariff and its main features.

`tariffType` An enum specifying the main characteristics of the tarif. Possible values are:

- **annuity** Whole life or term annuity (periodic survival benefits) with flexible payouts (constand, increasing, decreasing, arbitrary, etc.)
- **wholelife** A whole or term life insurance with only death benefits. The benefit can be constant, increasing, decreasing, described by a function, etc.
- **endowment** An endowment with death and survival benefits, potentially with different benefits.
- **pureendowment** A pure endowment with only a survival benefit at the end of the contract. Optionally, in case of death, all or part of the premiums paid may be refunded.
- **terme-fix** A terme-fix insurance with a fixed payout at the end of the contract, even if the insured dies before that time. Premiums are paid until death of the insured.
- **dread-disease** A dread-disease insurance, which pays in case of a severe illness (typically heart attacks, cancer, strokes, etc.), but not in case of death.
- **endowment + dread-disease** A combination of an endowment and a temporary dread-disease insurance. Benefits occur either on death, severe illness or survival, whichever comes first.

`Parameters` A data structure (nested list) containing all relevant parameters describing a contract, its underlying tariff, the profit participation scheme etc. See InsuranceContract.ParameterStructure for all fields.

`dummy` Dummy field to allow commas after the previous method

## Methods

### Public methods:

- `InsuranceTarif$new()`
- `InsuranceTarif$createModification()`
- `InsuranceTarif$getParameters()`
- `InsuranceTarif$getInternalValues()`
- `InsuranceTarif$getAges()`
- `InsuranceTarif$getTransitionProbabilities()`
- `InsuranceTarif$getCostValues()`
- `InsuranceTarif$getPremiumCF()`
- `InsuranceTarif$getAnnuityCF()`
- `InsuranceTarif$getDeathCF()`
- `InsuranceTarif$getBasicCashFlows()`
- `InsuranceTarif$getCashFlows()`
- `InsuranceTarif$getCashFlowsCosts()`
- `InsuranceTarif$presentValueCashFlows()`
- `InsuranceTarif$presentValueCashFlowsCosts()`
- `InsuranceTarif$getAbsCashFlows()`
- `InsuranceTarif$getAbsPresentValues()`
- `InsuranceTarif$presentValueBenefits()`
- `InsuranceTarif$getPremiumCoefficients()`
- `InsuranceTarif$premiumCalculation()`
- `InsuranceTarif$reserveCalculation()`
- `InsuranceTarif$getBalanceSheetReserveFactor()`
- `InsuranceTarif$reserveCalculationBalanceSheet()`
- `InsuranceTarif$calculateProfitParticipation()`
- `InsuranceTarif$reservesAfterProfit()`
- `InsuranceTarif$getBasicDataTimeseries()`
- `InsuranceTarif$premiumDecomposition()`
- `InsuranceTarif$calculateFutureSums()`
- `InsuranceTarif$calculatePresentValues()`
- `InsuranceTarif$clone()`

**Method** `new()`**:** Initialize a new tariff object

*Usage:*

```
InsuranceTarif$new(
  name = NULL,
  type = "wholelife",
  tarif = "Generic Tarif",
  desc = "Description of tarif",
  ...
)
```

*Arguments:*

`name` The unique name / ID of the tariff

`type` An enum specifying the main characteristics of the tarif. See TariffTypeEnum

`tarif` The tariff's public name to be stored in the `tarif` field.

`desc` A short human-readable description to be stored in the `desc` field.

`...` Parameters for the InsuranceContract.ParameterStructure, defining the characteristics of the tariff.

*Details:* The constructor function defines a tariff and generates the corresponding data structure, which can then be used with the InsuranceContract class to define an actual contract using the tariff.
The arguments passed to this function will be stored inside the `Parameters` field of the class, inside one of the lists sublists. The parameters are stacked from different layers (higher levels override default values from lower layers):

- InsuranceContract object (parameters passed directly to the individual contract)
- ProfitParticipation object (parameters for profit participation, passed to the definition of the profit plan, which is used for the tarif definition or the contract)
- InsuranceTarif object (parameters passed to the definition of the tariff that was used for the contract)
- Defaults taken from InsuranceContract.ParameterStructure

The general implementation of this parameter layering means that (a) a tariff can already provide default values for contracts (e.g. a default maturity, default sum insured, etc) and (b) individual contracts can override all parameters defined with the underlying tariff. In particular the latter feature has many use-cases in prototyping: E.g. when you have a tariff with a guaranteed interest rate of 1\ one can immediately instantiate a contract with an updated interest rate or mortality table for comparison. There is no need to re-implement a tariff for such comparisons, as long as only parameters are changed.

*Examples:*

```
MortalityTables::mortalityTables.load("Austria_Annuities_AVOe2005R")
tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)
```

**Method** `createModification()`**:** create a copy of a tariff with certain parameters changed

*Usage:*

```
InsuranceTarif$createModification(
  name = NULL,
  tarif = NULL,
  desc = NULL,
  tariffType = NULL,
  ...
)
```

*Arguments:*

`name` The unique name / ID of the tariff

`tarif` The tariff's public name to be stored in the `tarif` field.

`desc` A short human-readable description to be stored in the `desc` field.

`tariffType` An enum specifying the main characteristics of the tarif. See TariffTypeEnum

`...` Parameters for the InsuranceContract.ParameterStructure, defining the characteristics of the tariff.

*Details:* This method `createModification` returns a copy of the tariff with all given arguments changed in the tariff's `InsuranceTarif$Parameters` parameter list.
As InsuranceTarif is a R6 class with reference logic, simply assigning the object to a new variable does not create a copy, but references the original tariff object. To create an actual copy, one needs to call this method, which first clones the whole object and then adjusts all parameters to the values passed to this method.

*Examples:*

```
MortalityTables::mortalityTables.load("Austria_Annuities_AVOe2005R")
tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)
tarif.unisex = tarif.male$createModification(name = "Annuity unisex",
    mortalityTable = AVOe2005R.unisex)
```

**Method** `getParameters()`**:** Retrieve the parameters for this tariff (can be overridden for each contract)

*Usage:*

```
InsuranceTarif$getParameters()
```

*Examples:*

```
tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)
tarif.male$getParameters()
```

**Method** `getInternalValues()`**:** Get some internal parameters cached (length of data.frames, policy periods cut at max.age, etc.)

*Usage:*

```
InsuranceTarif$getInternalValues(params, ...)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`...` currently unused

*Details:* This methos is not meant to be called explicitly, but rather used by the InsuranceContract class. It returns a list of maturities and ages relevant for the contract-specific calculations

**Method** `getAges()`**:** Calculate the contract-relevant age(s) given a certain parameter data structure (contract-specific values)

*Usage:*

```
InsuranceTarif$getAges(params)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

*Details:* This method is not meant to be called explicitly, but rather used by the InsuranceContract class. It returns the relevant ages during the whole contract period

**Method** `getTransitionProbabilities()`**:** Calculate the transition probabilities from the contract-specific parameters passed as `params` and the already-calculated contract values `values`

*Usage:*

```
InsuranceTarif$getTransitionProbabilities(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getCostValues()`**:** Obtain the cost structure from the cost parameter and the given paremeter set

*Usage:*

```
InsuranceTarif$getCostValues(params)
```

*Arguments:*

`params` The parameters of the contract / tariff

*Details:* Not to be called directly, but implicitly by the InsuranceContract object. The cost parameter can be either an array of costs (generated by `initializeCosts()`) or a function with parameters `param` and `values`(=NULL) returning an array of the required dimensions. This function makes sures that the latter function is actually evaluated.

**Method** `getPremiumCF()`**:** Returns the unit premium cash flow for the whole contract period.

- For constant premiums it will be rep(1, premiumPeriod),
- for single premiums it will be c(1, 0, 0, ...),
- for increasing premiums it will be (1+increase)^(0:(premiumPeriod-1)) and 0 after the premium period

*Usage:*

```
InsuranceTarif$getPremiumCF(len, params, values)
```

*Arguments:*

`len` The desired length of the returned data frame (the number of contract periods desire)

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getAnnuityCF()`**:** Returns the unit annuity cash flow (guaranteed and contingent) for the whole annuity payment period (after potential deferral period)

- For constant annuity it will be rep(1, annuityPeriod),
- for increasing annuities it will be (1+increase)^(0:(premiumPeriod-1)) and 0 after the premium period

*Usage:*

```
InsuranceTarif$getAnnuityCF(len, params, values)
```

*Arguments:*

`len` The desired length of the returned data frame (the number of contract periods desire)

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getDeathCF()`**:** Returns the unit death cash flow for the whole protection period (after potential deferral period!)

- For constant death benefit it will be rep(1, policyPeriod),
- for linearly decreasing sum insured it will be (policyPeriod:0)/policyPeriod

*Usage:*

```
InsuranceTarif$getDeathCF(len, params, values)
```

*Arguments:*

`len` The desired length of the returned data frame (the number of contract periods desire)

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getBasicCashFlows()`**:** Returns the basic (unit) cash flows associated with the type of insurance given in the InsuranceTarif's `tariffType` field

*Usage:*

```
InsuranceTarif$getBasicCashFlows(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getCashFlows()`**:** Returns the cash flows for the contract given the parameters

*Usage:*

```
InsuranceTarif$getCashFlows(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getCashFlowsCosts()`**:** Returns the cost cash flows of the contract given the contract and tariff parameters

*Usage:*

```
InsuranceTarif$getCashFlowsCosts(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `presentValueCashFlows()`**:** Returns the present values of the cash flows of the contract (cash flows already calculated and stored in the `cashFlows` data.frame)

*Usage:*

```
InsuranceTarif$presentValueCashFlows(cashFlows, params, values)
```

*Arguments:*

`cashFlows` data.frame of cash flows calculated by a call to `InsuranceTarif$getCashFlows()`

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `presentValueCashFlowsCosts()`**:** Calculates the present values of the cost cash flows of the contract (cost cash flows alreay calculated by `InsuranceTarif$getCashFlowsCosts()` and stored in the `values` list

*Usage:*

```
InsuranceTarif$presentValueCashFlowsCosts(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getAbsCashFlows()`: Calculate the cash flows in monetary terms of the insurance contract

*Usage:*

```
InsuranceTarif$getAbsCashFlows(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Once the premiums of the insurance contracts are calculated, all cash flows can also be expressed in absolute terms. This function calculates these time series in monetary terms, once the premiums are calculated by the previous functions of this class.
This method is NOT to be called directly, but implicitly by the InsuranceContract object.

**Method** `getAbsPresentValues()`: Calculate the absolute present value time series of the insurance contract

*Usage:*

```
InsuranceTarif$getAbsPresentValues(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Once the premiums of the insurance contracts are calculated, all present values can also be expressed in absolute terms. This function calculates these time series in monetary terms, once the premiums and the unit-benefit present values are calculated by the previous functions of this classe.
This method is NOT to be called directly, but implicitly by the InsuranceContract object.

**Method** `presentValueBenefits()`: Calculate the absolute present value time series of the benefits of the insurance contract

*Usage:*

```
InsuranceTarif$presentValueBenefits(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Once the premiums of the insurance contracts are calculated, all present values can also be expressed in absolute terms. This function calculates these time series of the benefits present values in monetary terms, once the premiums and the unit-benefit present values are calculated by the previous functions of this classe.
This method is NOT to be called directly, but implicitly by the InsuranceContract object.

**Method** `getPremiumCoefficients()`**:** Calculate the linear coefficients of the premium calculation formula for the insurance contract

*Usage:*

```
InsuranceTarif$getPremiumCoefficients(
  type = "gross",
  coeffBenefits,
  coeffCosts,
  premiums,
  params,
  values,
  premiumCalculationTime = values$int$premiumCalculationTime
)
```

*Arguments:*

`type` The premium that is supposed to be calculated ("gross", "Zillmer", "net")

`coeffBenefits` (empty) data structure of the benefit coefficients. The actual values have no meaning, this parameter is only used to derive the required dimensions

`coeffCosts` (empty) data structure of the cost coefficients. The actual values have no meaning, this parameter is only used to derive the required dimensions

`premiums` The premium components that have already been calculated (e.g. for net and Zillmer, the gross premium has already been calculated to allow modelling the premium refund)

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

`premiumCalculationTime` The time when the premiums should be (re-)calculated according to the equivalence principle. A time 0 means the initial premium calculation at contract closing, later premium calculation times can be used to re-calculate the new premium after a contract change (possibly including an existing reserve)

*Details:* Not to be called directly, but implicitly by the InsuranceContract object. When `getPremiumCoefficients` is called, the `values$premiums` array has NOT yet been filled! Instead, all premiums already calculated (and required for the premium coefficients) are passed in the `premiums` argument.

**Method** `premiumCalculation()`**:** Calculate the premiums of the InsuranceContract given the parameters, present values and premium cofficients already calculated and stored in the `params` and `values` lists.

*Usage:*

```
InsuranceTarif$premiumCalculation(
  params,
  values,
  premiumCalculationTime = values$int$premiumCalculationTime
)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

`premiumCalculationTime` The time when the premiums should be (re-)calculated according to the equivalence principle. A time 0 means the initial premium calculation at contract closing, later premium calculation times can be used to re-calculate the new premium after a contract change (possibly including an existing reserve)

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `reserveCalculation()`**:** Calculate the reserves of the InsuranceContract given the parameters, present values and premiums already calculated and stored in the `params` and `values` lists.

*Usage:*

```
InsuranceTarif$reserveCalculation(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getBalanceSheetReserveFactor()`**:** Calculate the (linear) interpolation factors for the balance sheet reserve (Dec. 31) between the yearly contract closing dates

*Usage:*

```
InsuranceTarif$getBalanceSheetReserveFactor(method, params, years = 1)
```

*Arguments:*

`method` The method for the balance sheet interpolation (30/360, act/act, act/360, act/365 or a function)

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`years` how many years to calculate (for some usances, the factor is different in leap years!)

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `reserveCalculationBalanceSheet()`**:** Calculate the reserves for the balance sheets at Dec. 31 of each year by interpolation from the contract values calculated for the yearly reference date of the contract

*Usage:*

```
InsuranceTarif$reserveCalculationBalanceSheet(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `calculateProfitParticipation()`**:** Calculate the profit participation given the contract parameters and the already calculated reserves of the contract.

*Usage:*

```
InsuranceTarif$calculateProfitParticipation(params, ...)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`...` Additional parameters for the profit participation calculation, passed through to the profit participation scheme's `ProfitParticipation$getProfitParticipation()`

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `reservesAfterProfit()`**:** Calculate the reserves after profit participation for the given profit scenario

*Usage:*

```
InsuranceTarif$reservesAfterProfit(profitScenario, params, values, ...)
```

*Arguments:*

`profitScenario` The ID of the profit scenario for which to calculate the reserves

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

`...` TODO

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `getBasicDataTimeseries()`**:** Return the time series of the basic contract

*Usage:*

```
InsuranceTarif$getBasicDataTimeseries(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object.

**Method** `premiumDecomposition()`**:** Calculate the time series of the premium decomposition of the contract

*Usage:*

```
InsuranceTarif$premiumDecomposition(params, values)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (in the `contract$Values` list) then this method is called by the contract object

*Details:* Not to be called directly, but implicitly by the InsuranceContract object. All premiums, reserves and present values have already been calculated.

**Method** `calculateFutureSums()`**:** Generic function to calculate future sums of the values

*Usage:*

```
InsuranceTarif$calculateFutureSums(values, ...)
```

*Arguments:*

`values` The time series, for which future sums at all times are desired

`...` currently unused

**Method** `calculatePresentValues()`**:** Calculate all present values for a given time series. The mortalities are taken from the contract's parameters.

*Usage:*

```
InsuranceTarif$calculatePresentValues(values, params)
```

*Arguments:*

`values` The time series, for which future present values at all times are desired

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`...` currently unused

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
InsuranceTarif$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
# Define an insurance tariff for 10-year endowments, using a guaranteed interest
# rate of 1% and the Austrian population mortality table of the census 2011.
# Premiums are paid monthly in advance during the whole contract period.
MortalityTables::mortalityTables.load("Austria_Census")
# Cost structure:
#   - 4% up-front acquisition costs (of premium sum)
#   - 1% collection cost of each premium paid
#   - 1%o yearly administration cost (of the sum insured) as long as premiums are paid
#   - 2%o yearly administration cost for paid-up contracts
#   - 10 Euro yearly unit costs (as long as premiums are paid)
costs.endw = initializeCosts(alpha = 0.04, beta = 0.01, gamma = 0.001,
    gamma.paidUp = 0.002, gamma.premiumfree = 0.002, unitcosts = 10)

endowment.AT1 = InsuranceTarif$new(
    name = "Endow AT 1%", type = "endowment", tarif = "Austrian Endowment",
   desc = "An endowment for Austrian insured with 1% interest and no profit participation",
    ContractPeriod = 10,
    i = 0.01, mortalityTable = mort.AT.census.2011.unisex,
    costs = costs.endw, premiumFrequency = 12)
```

```
# The instantiation of the actual contract will provide the contract specific
# information and immediately calculate all further values:
ctr.end.AT1 = InsuranceContract$new(tarif = endowment.AT1,
    contractClosing = as.Date("2020-07-01"), age = 42)

# All values for the contract are already calculated during construction and
# stored in the ctr.end.AT1$Values list:
ctr.end.AT1$Values$basicData
ctr.end.AT1$Values$transitionProbabilities
ctr.end.AT1$Values$cashFlowsCosts
ctr.end.AT1$Values$presentValues
ctr.end.AT1$Values$premiums
ctr.end.AT1$Values$reserves
ctr.end.AT1$Values$premiumComposition
# etc.

## ------------------------------------------------
## Method `InsuranceTarif$new`
## ------------------------------------------------

MortalityTables::mortalityTables.load("Austria_Annuities_AVOe2005R")
tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)

## ------------------------------------------------
## Method `InsuranceTarif$createModification`
## ------------------------------------------------

MortalityTables::mortalityTables.load("Austria_Annuities_AVOe2005R")
tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)
tarif.unisex = tarif.male$createModification(name = "Annuity unisex",
    mortalityTable = AVOe2005R.unisex)

## ------------------------------------------------
## Method `InsuranceTarif$getParameters`
## ------------------------------------------------

tarif.male = InsuranceTarif$new(name = "Annuity Males", type = "annuity",
    i = 0.01, mortalityTable = AVOe2005R.male)
tarif.male$getParameters()
```

---

`isRegularPremiumContract` *Determine whether a contract (given all parameters) is a contract with regular premiums*

---

## Description

Regular premium contracts are identified by the parameter `premiumPeriod` > 1.

### Usage

```
isRegularPremiumContract(params, values)
```

### Arguments

| | |
|---|---|
| params | The parameters of the contract. |
| values | Unused by default (already calculated values of the contract) |

---

## isSinglePremiumContract

*Determine whether a contract (given all parameters) is a single-premium contract or with regular premiums*

---

### Description

Single premium contracts are identified by the parameter `premiumPeriod = 1`.

### Usage

```
isSinglePremiumContract(params, values)
```

### Arguments

| | |
|---|---|
| params | The parameters of the contract. |
| values | Unused by default (already calculated values of the contract) |

---

## makeContractGridDimname

*Create human-readable labels for the dimensions in a* `contractGrid()`

---

### Description

The function `makeContractGridDimname` generates a short, human-readable dimension label for the entries along the axes of a `contractGrid()`. The default is to use the `value` unchanged as the row-/columnname, but for some parameter values (like a InsuranceTarif or mortalityTable) a custom method of this function is needed to create the (short) human-readable representation for the axes in the grid.

The function `makeContractGridDimnames` generate proper dimnames for all entries of the axes of a `contractGrid()` by calling `makeContractGridDimname` on each of the axes' values

### Usage

```
makeContractGridDimname(value)

makeContractGridDimnames(axes)
```

### Arguments

| | |
|---|---|
| value | the value along the axis, for which a name should be generated |
| axes | the axes with all names, for which a name should be generated |

### Functions

- makeContractGridDimname: Create a short, human-readable dimensional name for an object (default S3 method)
- makeContractGridDimnames: Generate proper dimnames for all entries of the axes of a contractGrid()

### Examples

```
library(MortalityTables)
mortalityTables.load("Austria_Census")

makeContractGridDimname(mort.AT.census.2011.unisex)

makeContractGridDimnames(axes = list(
    age = seq(30,60,10),
    mortalityTable = c(mort.AT.census.2011.unisex, mort.AT.census.2011.male,
                       mort.AT.census.2011.female))
)
```

---

## pad0 *Pad a vector with 0 to a desired length*

---

### Description

Pad a vector with 0 to a desired length

### Usage

```
pad0(v, l, value = 0, start = 0)
```

### Arguments

| | |
|---|---|
| v | the vector to pad with 0 |
| l | the desired (resulting) length of the vector |
| value | the value to pad with (if padding is needed). Default to 0, but can be overridden to pad with any other value. |
| start | the first start values are always set to 0 (default is 0), the vector v starts only after these leading zeroes. The number of leading zeroes counts towards the desired length |

## Value

returns the vector v padded to length l with value value (default 0).

## Examples

```
pad0(1:5, 7)   # Pad to length 7 with zeroes
pad0(1:5, 3)   # no padding, but cut at length 3

# 3 leading zeroes, then the vector start (10 elements of vector, no additional padding needed):
pad0(1:10, 13, start = 3)

# padding with value other than zero:
pad0(1:5, 7, value = "pad")
```

---

# padLast *Pad the vector* v *to length* l *by repeating the last entry of the vector.*

---

## Description

This function callc pad0() with the last element of the vector as padding value

## Usage

```
padLast(v, l, start = 0)
```

## Arguments

| | |
|---|---|
| v | the vector to pad by repeating the last element |
| l | the desired (resulting) length of the vector |
| start | the first start values are always set to 0 (default is 0), the vector v starts only after these leading zeroes. The number of leading zeroes counts towards the desired length |

## Examples

```
padLast(1:5, 7) # 5 is repeated twice
padLast(1:5, 3) # no padding needed
```

`PaymentTimeSingleEnum-class` *Enum to describe when a benefit or premium payment is due (in advance or in arrears)*

## Description

Enum to describe when a benefit or premium payment is due (in advance or in arrears)

## Details

Currently, only two values are allowed;

- "in advance"
- "in arrears"

`ProfitComponentsMultipleEnum-class` *Enum to define the different components of profit participation.*

## Description

Enum to define the different components of profit participation.

## Details

Profit participation schemes typically consist of different components, which are calculated independently. Typical components are interest profit to distribute investment gains to the customer, risk profit and expense profit to return security margins in the biometric risk and the expenses to the customer and sum profit, which aplies to contracts with higher sums insured, where charged expenses are calculated from the sum insured, while the actual expenses are more or less constant. Thus, high contracts are charged more, which causes profits that are returned as sum profit.

As a special case, part of the profits can be stored in a terminal bonus reserve and only distributed on maturity (or potentially on death), but not on surrender. Some (older) profit participation schemes add an independently calculated bonus on maturity (e.g. twice the total profit assignment of the last year) at maturity to give customers an additional incentive not to surrender a contract.

Possible values are (multiple can be given):

- "interest"
- "risk"
- "expense"
- "sum"
- "terminal"
- "TBF"

`ProfitParticipation` *Base Class for Profit Participation Schemes*

## Description

Base Class for Profit Participation Schemes

Base Class for Profit Participation Schemes

## Details

Base class for Profit Participation schemes (holding contract-independent values and providing methods to calculate the profit participation values from the given reserves).

The profit participation object is typically not used directly, but rather defined once and then passed on to an InsuranceTarif or InsuranceContract object, where it will be used internally when profit participation is calculated.

This class provides the technical implementation of a profit plan for traditional life insurance contracts with a guaranteed component (calculated before the profit scheme comes into play) and a discretionary profit on top.

This function is called when a new profit participation scheme is created with a call to `ProfitParticipation$new(...)`. Possible parameters to the `new`-Call are all parameters from the `ProfitParticipation` sublist of the InsuranceContract.ParameterStructure parameter structure (which are understood as template values that can be overridden per contract or even per profit participation scenario) and the components of the `Functions` field defining the functions to calculate the individual components of the profit participation (rates, calculation bases, calculation, benefits)

This method `createModification` returns a copy of the profit scheme with all given arguments changed in the schmes's `Parameters` parameter list.

As ProfitParticipation is a R6 class with reference logic, simply assigning the object to a new variable does not create a copy, but references the original profit scheme object. To create an actual copy, one needs to call this method, which first clones the whole object and then adjusts all parameters to the values passed to this method.

The InsuranceContract's param structure InsuranceContract.ParameterStructure contains the field `params$ProfitParticipation$advanceProfitParticipation`, which can either be numeric rate for advance profit participation, or a function with signature `function(params,values,...)` that returns the advance profit participation rate when called with the contract's parameters and the values calculated so far (cash flows and premiums)

The InsuranceContract's param structure InsuranceContract.ParameterStructure contains the field `params$ProfitParticipation$advanceProfitParticipationInclUnitCost`, which can either be numeric rate for advance profit participation, or a function with signature `function(params,values,...)` that returns the advance profit participation rate when called with the contract's parameters and the values calculated so far (cash flows and premiums)

This function provides an easy way to modify the whole set of profit rates after their initial setup. Possible applications are waiting periods, which can be implemented once for all rates rather than inside each individual calculation period.

## Public fields

`name` The human-readable name of the profit plan.

`Parameters` Parameter template for profit-participation-specific parameters, i.e. the `ProfitParticipation` element of the InsuranceContract.ParameterStructure data structure.

All elements defined in the profit scheme can be overriden per contract in the call to `[InsuranceContract]$new` or even in the explicit call to `InsuranceContract$profitScenario()()` or `InsuranceContract$addProfitScenari`

`Functions` list of functions defined to calculate the individual components. For each of the profit components

- interest profit
- risk profit
- expense profit
- sum profit
- terminal bonus
- terminal bonus fund

a rate, a profit base and a calculation function can be defined, by assigning one of the predefined ProfitParticipationFunctions or proving your own function with signature `function(rates,params,values,..`. Additionally, for each of the benefit types (survival, death, surrender, premium waiver) a function can be provided to calculate the benefit stemming from profit participation.

`dummy` Dummy to allow commas in the previous method

## Methods

### Public methods:

- `ProfitParticipation$new()`
- `ProfitParticipation$setParameters()`
- `ProfitParticipation$setFunctions()`
- `ProfitParticipation$setFallbackParameters()`
- `ProfitParticipation$createModification()`
- `ProfitParticipation$getAdvanceProfitParticipation()`
- `ProfitParticipation$getAdvanceProfitParticipationAfterUnitCosts()`
- `ProfitParticipation$setupRates()`
- `ProfitParticipation$adjustRates()`
- `ProfitParticipation$getProfitParticipation()`
- `ProfitParticipation$clone()`

**Method** `new()`**:** Create a new profit participation scheme

*Usage:*

```
ProfitParticipation$new(name = NULL, ...)
```

*Arguments:*

`name` The name of the profit scheme (typicall the name of the profit plan and its version)

`...` profit participation parameters to be stored in the Parameters field or calculation functions to be stored in the Functions‘ field

**Method** `setParameters()`**:** Store all passed parameters in the `Parameters` field

*Usage:*

```
ProfitParticipation$setParameters(...)
```

*Arguments:*

`...` any of the named fields defined in the `ProfitParticipation` sublist of the InsuranceContract.ParameterStructure. All other arguments will be ignored

**Method** `setFunctions()`**:** Store all passed functions in the `Functions` field

*Usage:*

```
ProfitParticipation$setFunctions(...)
```

*Arguments:*

`...` any of the functions defined in the `Functions` field. All other arguments will be ignored

**Method** `setFallbackParameters()`**:** Fill all missing parameters with the default fall-back values

*Usage:*

```
ProfitParticipation$setFallbackParameters()
```

**Method** `createModification()`**:** create a copy of a profit scheme with certain parameters changed

*Usage:*

```
ProfitParticipation$createModification(name = NULL, ...)
```

*Arguments:*

`name` The new name for the cloned ProfitParticipation object

`...` Parameters for the InsuranceContract.ParameterStructure, defining the characteristics of the tariff.

**Method** `getAdvanceProfitParticipation()`**:** Calculate and return the advance profit participation (to be applied on the actuarial gross premium)

*Usage:*

```
ProfitParticipation$getAdvanceProfitParticipation(params, values, ...)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.).

`...` optional parameters, to be passed to the advanceProfitParticipation field of the parameter structure (if that is a function)

*Returns:* Return either one numerical value (constant for the whole premium payment period) of a vector of numerical values for the whole contract period

**Method** `getAdvanceProfitParticipationAfterUnitCosts()`**:** Calculate and return the advance profit participation (to be applied after unit costs are added to the gross premium)

*Usage:*

```
ProfitParticipation$getAdvanceProfitParticipationAfterUnitCosts(
  params,
  values,
  ...
)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.).

`...` optional parameters, to be passed to the advanceProfitParticipationInclUnitCost field of the parameter structure (if that is a function)

*Returns:* Return either one numerical value (constant for the whole premium payment period) of a vector of numerical values for the whole contract period

**Method** `setupRates()`**:** Set up the data.frame containing the profit participation rates

*Usage:*

```
ProfitParticipation$setupRates(params, values, ...)
```

*Arguments:*

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.).

`...` additional parameters passed to the profit calculation functions stored in the `Functions` field.

**Method** `adjustRates()`**:** Adjust the data.frame of profit participation rates after their setup

*Usage:*

```
ProfitParticipation$adjustRates(rates, params, values)
```

*Arguments:*

`rates` data.frame of profit paticipation rates

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.).

**Method** `getProfitParticipation()`**:** Calculation the full time series of profit participation for the given contract values

*Usage:*

```
ProfitParticipation$getProfitParticipation(
  calculateFrom = 0,
  profitScenario = NULL,
  params,
  values,
  ...
)
```

*Arguments:*

`calculateFrom` The time from which to start calculating the profit participation. When a contract is changed at some time t (possibly even changing the profit scheme), all future profit participation needs to be re-calculated from that time on, without changing past profit participation. All values before `calculateFrom` will not be calculated.

`profitScenario` profit participation values from a previous calculation (NULL if profit calculation is to be calculated from the contract inception). Values before `calculateFrom` will be used from this data.frame.

`params` Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract)

`values` Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.).

`...` additional parameters to be passed to `ProfitParticipation$setupRates()`

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
ProfitParticipation$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

---

`ProfitParticipationFunctions`

*Helper functions for profit participation*

---

## Description

Various helper functions for the `ProfitParticipation` class that provide the building blocks for the individual components of profit participation, the rates and how the assigned profit is calculated.

## Usage

```
PP.base.NULL(rates, params, values, ...)

PP.base.PreviousZillmerReserve(rates, params, values, ...)

PP.base.ZillmerReserveT2(rates, params, values, ...)

PP.base.contractualReserve(rates, params, values, ...)

PP.base.previousContractualReserve(rates, params, values, ...)

PP.base.meanContractualReserve(rates, params, values, ...)

PP.base.ZillmerRiskPremium(rates, params, values, ...)
```

```
PP.base.sumInsured(rates, params, values, ...)

PP.base.totalProfitAssignment(res, ...)

PP.rate.interestProfit(rates, ...)

PP.rate.riskProfit(rates, ...)

PP.rate.expenseProfit(rates, ...)

PP.rate.sumProfit(rates, ...)

PP.rate.terminalBonus(rates, ...)

PP.rate.terminalBonusFund(rates, ...)

PP.rate.interestProfitPlusGuarantee(rates, ...)

PP.rate.interestProfit2PlusGuarantee(rates, ...)

PP.rate.totalInterest(rates, ...)

PP.rate.totalInterest2(rates, ...)

PP.rate.interestProfit2(rates, ...)

getTerminalBonusReserve(
  profits,
  rates,
  terminalBonus,
  terminalBonusAccount,
  params,
  values,
  ...
)

PP.calculate.RateOnBase(base, rate, waiting, rates, params, values, ...)

PP.calculate.RateOnBaseMin0(base, rate, waiting, rates, params, values, ...)

PP.calculate.RatePlusGuaranteeOnBase(
  base,
  rate,
  waiting,
  rates,
  params,
  values,
  ...
```

```
)

PP.calculate.RateOnBaseSGFFactor(
  base,
  rate,
  waiting,
  rates,
  params,
  values,
  ...
)

sumProfits(profits, cols)

PP.benefit.ProfitPlusTerminalBonusReserve(profits, ...)

PP.benefit.Profit(profits, ...)

PP.benefit.ProfitPlusGuaranteedInterest(profits, rates, ...)

PP.benefit.ProfitPlusTotalInterest(profits, rates, params, values)

PP.benefit.ProfitPlusHalfTotalInterest(profits, ...)

PP.benefit.ProfitPlusHalfGuaranteedInterest(profits, rates, ...)

PP.benefit.ProfitPlusInterestMinGuaranteeTotal(profits, rates, ...)

PP.benefit.ProfitPlusHalfInterestMinGuaranteeTotal(profits, rates, ...)

PP.benefit.ProfitGuaranteeSupporting(profits, rates, params, values, ...)

PP.benefit.TerminalBonus5YearsProRata(profits, params, ...)

PP.benefit.TerminalBonus5Years(profits, params, ...)

PP.benefit.TerminalBonus(profits, params, ...)

PP.benefit.None(profits, ...)
```

## Arguments

| | |
|---|---|
| rates | data.frame of profit rates |
| params | Contract-specific, full set of parameters of the contract (merged parameters of the defaults, the tariff, the profit participation scheme and the contract) |
| values | Contract values calculated so far (guaranteed component of the insurance contract, including cash flows, premiums, reserves etc.). |
| ... | Other values that might be used for the calculation (currently unused) |

| | |
|---|---|
| res | the data.frame of reserves. |
| profits | The array of profit participation component values |
| terminalBonus | The terminal bonus calculated |
| terminalBonusAccount | The terminal bonus account (like a bank account, where terminal bonuses are accrued, potentiall discounted from the maturity) |
| base | The profit calculation base, on which the rate is to be applied |
| rate | The profit participation rate |
| waiting | A possible waiting period |
| cols | The columns of the profit values array to be summed (columns given that do not exist in the profits array are ignired) |

## Functions

- PP.base.NULL: Basis for profit: NONE (i.e. always returns 0)
- PP.base.PreviousZillmerReserve: Basis for profit: Previous Zillmer reserve (no administration cost reserve)
- PP.base.ZillmerReserveT2: Basis for profit: Zillmer reserve (no administration cost reserve) at time t-2
- PP.base.contractualReserve: Basis for profit: Contractual reserve (including administration costs) at time t
- PP.base.previousContractualReserve: Basis for profit: Contractual reserve (including administration costs) at time t-1
- PP.base.meanContractualReserve: Basis for profit: Contractual reserve (including administration costs) averaged over t and t-1
- PP.base.ZillmerRiskPremium: Basis for risk/mortality profit: Zillmer Risk Premium of the past year
- PP.base.sumInsured: Basis for expense/sum profit: sum insured
- PP.base.totalProfitAssignment: Basis for Terminal Bonus Fund Assignment: total profit assignment of the year
- PP.rate.interestProfit: Returns the array of interest profit rates (keyed by year)
- PP.rate.riskProfit: Returns the array of risk profit rates (keyed by year)
- PP.rate.expenseProfit: Returns the array of expense profit rates (keyed by year)
- PP.rate.sumProfit: Returns the array of sum profit rates (keyed by year)
- PP.rate.terminalBonus: Returns the array of terminal bonus rates (keyed by year)
- PP.rate.terminalBonusFund: Returns the array of terminal bonus rates (keyed by year) as the terminal bonus fund ratio
- PP.rate.interestProfitPlusGuarantee: Rate for interest on past profits: total credited rate, but at least the guarantee
- PP.rate.interestProfit2PlusGuarantee: Rate for interest on past profits: total creditedrate2, but at least the guarantee
- PP.rate.totalInterest: Rate for interest on past profits: total interest rate

- PP.rate.totalInterest2: Rate for interest on past profits: second total interest rate
- PP.rate.interestProfit2: Rate for interest on past profits: second interest profit rate (not including guaranteed interest), keyed by year
- getTerminalBonusReserve: Calculate the terminal bonus reserve.
- PP.calculate.RateOnBase: Calculate profit by a simple rate applied on the basis (with an optional waiting vector of values 0 or 1)
- PP.calculate.RateOnBaseMin0: Calculate profit by a simple rate applied on the basis (with an optional waiting vector of values 0 or 1), bound below by 0
- PP.calculate.RatePlusGuaranteeOnBase: Calculate profit by a rate + guaranteed interest applied on the basis (with an optional waiting vector of values 0 or 1)
- PP.calculate.RateOnBaseSGFFactor: Calculate profit by a simple rate applied on the basis (with only (1-SGFFactor) put into profit participation, and an optional waiting vector of values 0 or 1)
- sumProfits: Extract the given columns of the profit participation array of values and sum them up. Columns that do not exist, because the profit scheme does not provide the corresponding profit component will be silently ignored. This allows generic benefit calculation functions to be written that do not need to distinguish e.g. whether an old-style terminal bonus or a terminal bonus fund is provided.

  This function is not meant to be called directly, but within a profit benefit calculation function.
- PP.benefit.ProfitPlusTerminalBonusReserve: Calculate survival benefit as total profit amount plus the terminal bonus reserve
- PP.benefit.Profit: Calculate benefit as total profit accrued so far
- PP.benefit.ProfitPlusGuaranteedInterest: Calculate accrued death benefit as total profit with (guaranteed) interest for one year
- PP.benefit.ProfitPlusTotalInterest: Calculate accrued death benefit as total profit with total interest (interest on profit rate) for one year
- PP.benefit.ProfitPlusHalfTotalInterest: Calculate accrued benefit as total profit with total interest (interest on profit rate) for half a year
- PP.benefit.ProfitPlusHalfGuaranteedInterest: Calculate death benefit as total profit with (guaranteed) interest for one year
- PP.benefit.ProfitPlusInterestMinGuaranteeTotal: Calculate accrued benefit as total profit with interest for one year (max of guarantee and total interest)
- PP.benefit.ProfitPlusHalfInterestMinGuaranteeTotal: Calculate accrued benefit as total profit with interest for half a year (max of guarantee and total interest)
- PP.benefit.ProfitGuaranteeSupporting: Calculate accrued benefit as regular profit, but used to cover initial Zillmerization
- PP.benefit.TerminalBonus5YearsProRata: Calculate benefit from terminal bonus as 1/n parts of the terminal bonus reserve during the last 5 years
- PP.benefit.TerminalBonus5Years: Terminal bonus is only paid out during the last 5 years of the contract (but never during the first 10 years)
- PP.benefit.TerminalBonus: Calculate benefit from terminal bonus (full bonus), either old-style terminal bonus reserve or Terminal Bonus Fund (TBF)
- PP.benefit.None: No benefit paid out

## rollingmean *Calculate the rolling mean of length 2*

### Description

Calculate the rolling mean of length 2

### Usage

```
rollingmean(x)
```

### Arguments

| | |
|---|---|
| x | vector of values, for which the rolling mean is calculated |

### Examples

```
rollingmean(1:10)
```

## setCost *Update one component of an InsuranceTarif's cost structure*

### Description

Insurance tariff costs are defined by a cost matrix with dimensions: CostType, Basis, Period, where:

**CostType:** alpha, Zillmer, beta, gamma, gamma_nopremiums, unitcosts

**Basis:** SumInsured, SumPremiums, GrossPremium, NetPremium, Constant

**Period:** once, PremiumPeriod, PremiumFree, PolicyPeriod, AfterDeath, FullContract

After creation (using the function `initializeCosts()`), the funciton `setCost` can be used to modify a single entry. While `initializeCosts()` provides arguments to set the most common types of cost parameters in one call, the `setCost` function allows to modify any cost parameter.

### Usage

```
setCost(costs, type, basis = "SumInsured", frequency = "PolicyPeriod", value)
```

### Arguments

| | |
|---|---|
| costs | The cost definition matrix (usually created by `initializeCosts()`) |
| type | The cost type (alpha, Zillmer, beta, gamma, gamma_nopremiums, unitcosts) |
| basis | The basis fo which the cost rate is applied (default is SumInsured) |
| frequency | How often / during which period the cost is charged (once, PremiumPeriod, PremiumFree, PolicyPeriod, FullContract) |
| value | The new cost value to set for the given type, basis and frequency |

### Details

This function modifies a copy of the cost structure and returns it, so this function can be chained as often as desired and final return value will contain all cost modifications.

### Value

The modified cost structure

### Examples

```
costs = initializeCosts()
setCost(costs, "alpha", "SumPremiums", "once", 0.05)
```

---

## SexSingleEnum-class *Enum to describe possible sexes in an insurance contract or tariff.*

---

### Description

Enum to describe possible sexes in an insurance contract or tariff.

### Details

Currently, the only possible values are:

- "unisex"
- "male"
- "female"

---

## showVmGlgExamples *Display insurance contract calculation example*

---

### Description

Display the values of the example calculation of the given insurance contract as required by the Austrian regulation (LV-VMGLV, "LV Versicherungsmathematische Grundlagen Verordnung").

### Usage

```
showVmGlgExamples(contract, prf = 10, t = 10, t_prf = 12, file = "", ...)
```

## Arguments

| | |
|---|---|
| contract | The insurance contract to calculate and show |
| prf | Time of premium waiver (premium-free) |
| t | Time for which to show all values (except premium-free values) |
| t_prf | Time for which to show all values after the premium waiver |
| file | If given, outputs all information to the file rather than the console |
| ... | Further parameters (currently unused) |

## Examples

```
library(MortalityTables)
mortalityTables.load("Austria_Annuities_AVOe2005R")
# A trivial deferred annuity tariff with no costs:
tariff = InsuranceTarif$new(name="Test Annuity", type="annuity",
    mortalityTable = AVOe2005R.unisex, i=0.01)
contract = InsuranceContract$new(
    tariff,
    age = 35, YOB = 1981,
    policyPeriod = 30, premiumPeriod = 15, deferralPeriod = 15,
    sumInsured = 1000,
    contractClosing = as.Date("2016-10-01")
);
showVmGlgExamples(contract)

# Optionally output to a file rather than the console:
## Not run:
showVmGlgExamples(contract, file = "annuity-example.txt")

## End(Not run)
```

`TariffTypeSingleEnum-class`

*An enum specifying the main characteristics of the tarif. Possible values are:*

**annuity** *Whole life or term annuity (periodic survival benefits) with flexible payouts (constand, increasing, decreasing, arbitrary, etc.)*

**wholelife** *A whole or term life insurance with only death benefits. The benefit can be constant, increasing, decreasing, described by a function, etc.*

**endowment** *An endowment with death and survival benefits, potentially with different benefits.*

**pureendowment** *A pure endowment with only a survival benefit at the end of the contract. Optionally, in case of death, all or part of the premiums paid may be refunded.*

**terme-fix** *A terme-fix insurance with a fixed payout at the end of the contract, even if the insured dies before that time. Premiums are paid until death of the insured.*

**dread-disease** *A dread-disease insurance, which pays in case of a severe illness (typically heart attacks, cancer, strokes, etc.), but not in case of death.*

**endowment + dread-disease** *A combination of an endowment and a temporary dread-disease insurance. Benefits occur either on death, severe illness or survival, whichever comes first.*

---

## Description

An enum specifying the main characteristics of the tarif. Possible values are:

**annuity** Whole life or term annuity (periodic survival benefits) with flexible payouts (constand, increasing, decreasing, arbitrary, etc.)

**wholelife** A whole or term life insurance with only death benefits. The benefit can be constant, increasing, decreasing, described by a function, etc.

**endowment** An endowment with death and survival benefits, potentially with different benefits.

**pureendowment** A pure endowment with only a survival benefit at the end of the contract. Optionally, in case of death, all or part of the premiums paid may be refunded.

**terme-fix** A terme-fix insurance with a fixed payout at the end of the contract, even if the insured dies before that time. Premiums are paid until death of the insured.

**dread-disease** A dread-disease insurance, which pays in case of a severe illness (typically heart attacks, cancer, strokes, etc.), but not in case of death.

**endowment + dread-disease** A combination of an endowment and a temporary dread-disease insurance. Benefits occur either on death, severe illness or survival, whichever comes first.

`valueOrFunction` *If* `val` *is a function, evaluate it, otherwise return* `val`

## Description

If `val` is a function, evaluate it, otherwise return `val`

## Usage

```
valueOrFunction(val, ...)
```

## Arguments

| | |
|---|---|
| `val` | Function or value |
| `...` | Argument passed to `val` if it is a function |

## Examples

```
valueOrFunction(3) # returns 3
valueOrFunction(`+`, 1, 2) # also returns 3
A = `+`
valueOrFunction(A, 1, 2)
```

# Index