【変数の基本的な使い方】

【ローカル変数および定数の宣言】

サンプルコード

```
||次に例を示します。
{do
    let int-var:int = 4
    let dbl-var:double
    let char-var = 'z'
    def message = "Hello World"
}
```

【let と def の違い】

サンプルコード

```
||-----------------------------------------------
||以下のコードは、let と def の違いを示すためのサンプルです。
||-----------------------------------------------
||プロシージャを定義する
{define-proc {hello}:#String
    {return "hello"}
}

||違いを実行結果にて確認してください
||詳細は、Curl開発者ガイドを確認してください
{value
    def def-s = {hello}
    let let-s = {hello}

    {format
        "def-s type is '%s', let-s type is '%s'",
        {compile-time-type-of def-s},
        {compile-time-type-of let-s}
    }
}
```

実行結果

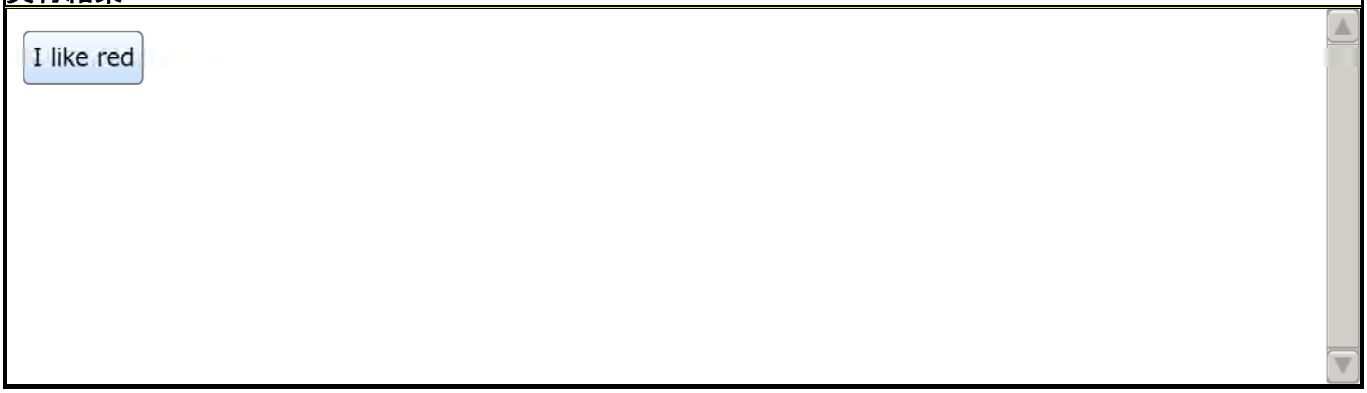
```
def-s type is '#String', let-s type is 'any'
```

【独自の定義における変数の使用】サンプル1

サンプルコード

```
||CommandButtonクラス型の変数を定義する
{let b:CommandButton =
    {CommandButton label="I like red",
        {on Action do
            set b.color = "red"
        }
    }
}

||"b"の値を表示する（CommandButtonが表示されます）
{value b}
```

実行結果

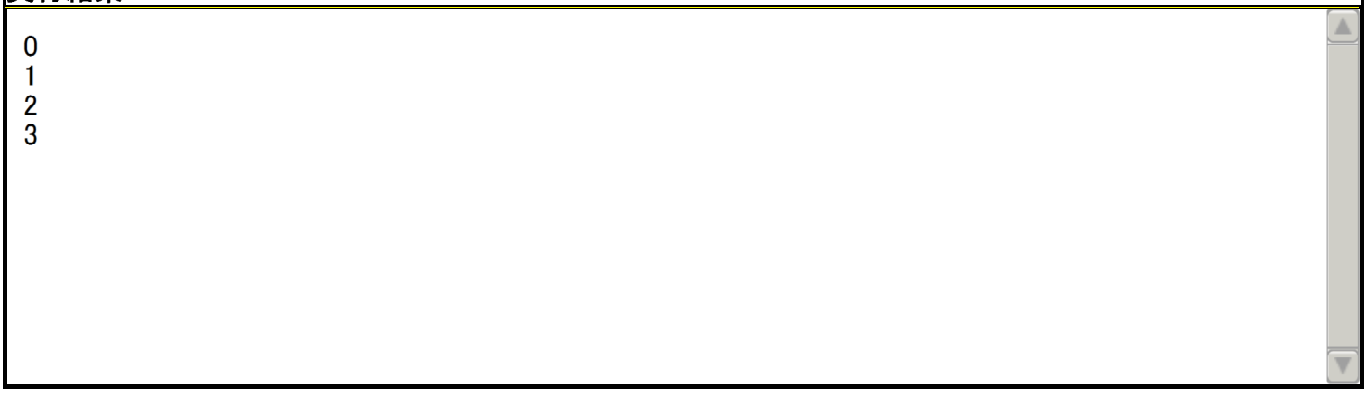


【独自の定義における変数の使用】サンプル2

サンプルコード

```
{value
    ||VBoxクラス型の変数を定義する
    let message:VBox = {VBox}

    ||４回繰り返す
    {for index:int = 0 to 3 do
        ||VBoxに値を追加する
        {message.add index}
    }

    ||"message"の値を表示する（VBoxを表示します）
    {value message}
}
```

実行結果

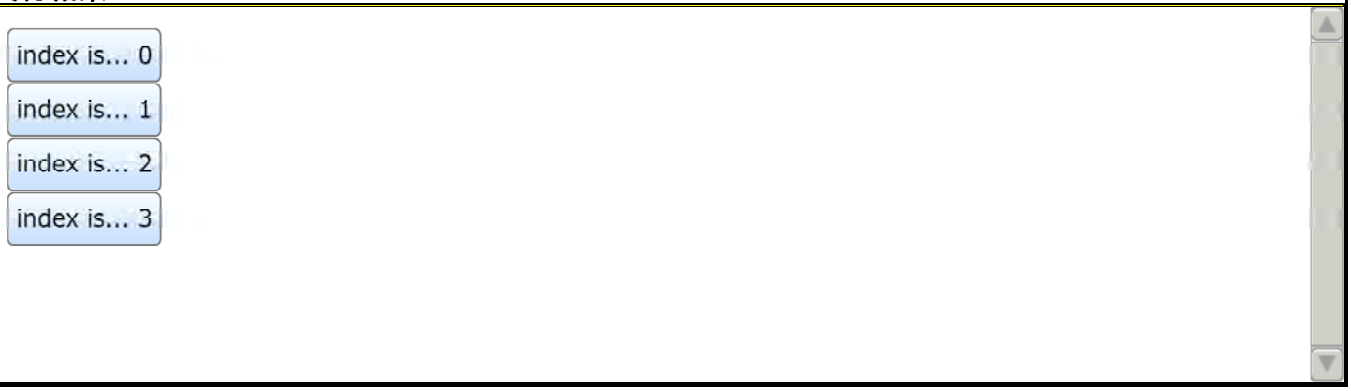


【独自の定義における変数の使用】サンプル3

サンプルコード

```
{value
    ||VBoxクラス型の変数を定義する
    let buttons:VBox = {VBox}
    let message:VBox = {VBox}

    ||４回繰り返す
    {for index:int = 0 to 3 do
        ||CommandButtonクラス型の変数を定義する
        let b:CommandButton = {CommandButton label="index is... " & index,
                                    {on Action do
                                        {message.add index}
                                    }
                                }
        ||CommandButtonを追加する
        {buttons.add b}
    }

    ||変数を画面に表示する
    {VBox
        {value buttons},
        {value message}
    }
}
```

実行結果

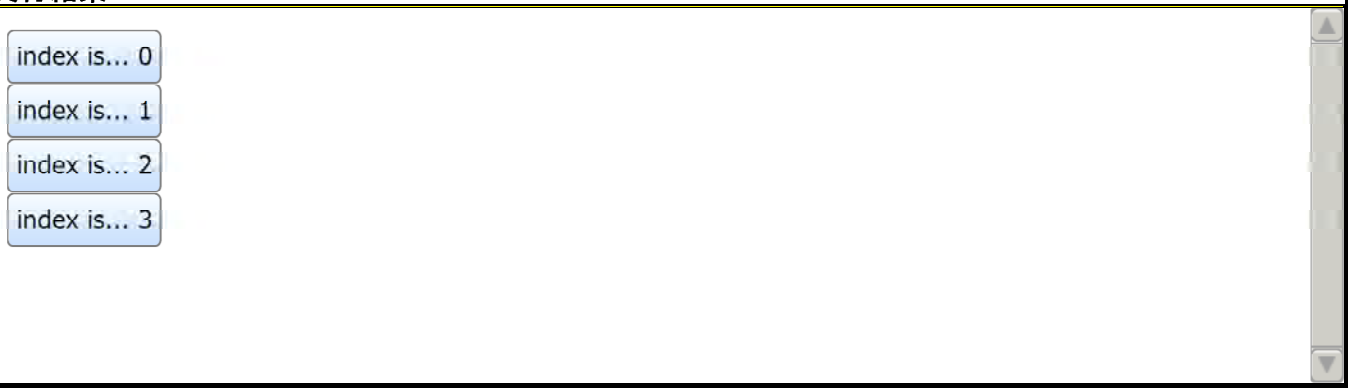


【独自の定義における変数の使用】サンプル4

サンプルコード

```
{value
    ||変数を定義する
    let buttons:VBox = {VBox}
    let message:VBox = {VBox}

    ||４回繰り返す
    {for index:int = 0 to 3 do
        || int型変数を定義する
        let temp:int = index
        ||CommandButtonクラス型変数を定義する
        let b:CommandButton = {CommandButton label="index is... " & temp,
                                    {on Action do
                                        {message.add temp}
                                    }
                                }
        ||VBoxに追加する
        {buttons.add b}
    }

    ||変数を画面に表示する
    {VBox
        {value buttons},
        {value message}
    }
}
```

実行結果



【独自の定義における変数の使用】サンプル５

サンプルコード

```
{value
    ||変数を定義する
    let buttons:VBox = {VBox}
    let message:VBox = {VBox}
    let temp:int

    ||４回繰り返す
    {for index:int = 0 to 3 do
        ||indexの値をtempに設定する
        set temp = index
        ||CommandButtonクラス型変数を定義する
        let b:CommandButton = {CommandButton label="index is... " & temp,
                                    {on Action do
                                        {message.add temp}
                                    }
                                }
        ||VBoxに追加する
        {buttons.add b}
    }

    ||変数を画面に表示する
    {VBox
        {value buttons},
        {value message}
    }
}
```

実行結果

【プリミティブ型、4つのパターンと使い方】

【整数】

サンプルコード

```
||変数を定義する
{let a:int}
{let b:int=-5}
{let c:int=5}
{let d:int8=5}
{let e:int8=256 asa int8}
{let f:int16=5}
{let g:int32=5}
{let h:int64=5}
{let i:int=(2.7 asa int)}

||上記定義の違いにより実行結果が異なることを確認してください
a is ... {value a}{br}
b is ... {value b}{br}
c is ... {value c}{br}
d is ... {value d}{br}
e is ... {value e}{br}
f is ... {value f}{br}
g is ... {value g}{br}
h is ... {value h}{br}
i is ... {value i}
```

実行結果

```
a is ... 0
b is ... -5
c is ... 5
d is ... 5
e is ... 0
f is ... 5
g is ... 5
h is ... 5
i is ... 2
```

【浮動小数点数】

サンプルコード

```
||変数を定義する
{let a:float}
{let b:float=5}
{let c:float=5.0f}
{let d:float=5.5f}
{let e:float=-5.5f}
{let f:float=1234.1234f}
{let g:double}
{let h:double=123456.123456}
{let i:double=12345678.12345678}

||上記定義の違いにより実行結果が異なることを確認してください
a is ... {value a}{br}
b is ... {value b}{br}
c is ... {value c}{br}
d is ... {value d}{br}
e is ... {value e}{br}
f is ... {value f}{br}
g is ... {value g}{br}
h is ... {value h}{br}
i is ... {value i}
```

実行結果

```
a is ... 0
b is ... 5
c is ... 5
d is ... 5.5
e is ... -5.5
f is ... 1234.12
g is ... 0
h is ... 123456
i is ... 1.23457e+007
```

【ブール値】

サンプルコード

```
||変数を定義する
{let a:bool}
{let b:bool = true}

||上記定義の違いにより実行結果が異なることを確認してください
a is ... {value a}{br}
b is ... {value b}
```

実行結果

```
a is ... false
b is ... true
```

【文字】

サンプルコード

```
||変数を定義する
{let foo:char}
{let bar:char='a'}
{let baz:char='¥u0061'}

||上記定義の違いにより実行結果が異なることを確認してください
foo is ... {value foo}{br}
bar is ... {value bar}{br}
baz is ... {value baz}
```

実行結果

```
foo is ...
bar is ... a
baz is ... a
```

【その他の型(null/列挙型)】

【数量】サンプル1

サンプルコード

```
||変数を定義する
{let d1:Distance = 4in}
{let d2:Distance = 10m}

||単位の異なる数量を比較します
||実行結果を確認してください
{if d1 > d2 then
    d1
 else
    d2
}
```

実行結果

```
10m
```

【数量】サンプル2

サンプルコード

```
||変数を定義する
{let s1:Speed = 15m / 2s}
{let s2:Speed = 7.5(m/s)}
||type-of 演算子を使用して独自の型を作成する
{let speed-squared:{type-of 1(m^2/s^2)} = s1 * s2}

||結果を表示する
||実行結果を確認してください
{value speed-squared}
```

実行結果

```
56.25(m^2*s^-2)
```

【null 値】

サンプルコード

```
|| 変数を定義する
{let x:#VBox}

|| 変数xがnullかどうかチェックする
x is equal to null is ... {value x == null}

|| 変数xにVBoxクラスのインスタンスを割り当てる
{set x = {VBox}}

|| 変数xがnullかどうかチェックする
||  実行結果を確認してください
x is equal to null is ... {value x == null}
```

実行結果

```
x is equal to null is ... true

x is equal to null is ... false
```

【列挙型の定義】

サンプルコード

```
||-----------------------------------------------------
|| 以下のコードは、列挙型を説明するためのサンプルです。
||  詳細は、Curl開発者ガイドを確認してください
||-----------------------------------------------------

|| 列挙型を定義する
{define-enum Bear
    polar = "endangered",
    grizzly = "threatened",
    panda,
    pooh = "abundant"
}

|| 定義された要素（Bear）の数を返す
{enum-size Bear}
```

実行結果

```
4
```

【列挙型プロパティの使用】

サンプルコード

```
|| 列挙型を定義する
{define-enum Bear
    polar = "endangered",
    grizzly = "threatened",
    panda,
    pooh = "abundant"
}

|| 列挙型の要素へアクセスするためのサンプル
||  実行結果を確認してください
{value Bear.polar.name}
{value Bear.polar.index}
{value Bear.polar.value}

{value {Bear name="polar"}.name}
{value {Bear name="polar"}.index}
{value {Bear name="polar"}.value}

{value {Bear index=0}.name}
{value {Bear index=0}.index}
{value {Bear index=0}.value}

{value {Bear value="endangered"}.name}
{value {Bear value="endangered"}.index}
{value {Bear value="endangered"}.value}
```

実行結果

```
polar 0 endangered

polar 0 endangered

polar 0 endangered

polar 0 endangered
```

【列挙型の変数】1

サンプルコード

```
||列挙型を定義する
{define-enum Bear
    polar, grizzly, panda, pooh
}

||列挙型の変数に値を代入するためのサンプルです（パターン１）
||実行結果を確認してください
The value Bear.panda is of type {type-of Bear.panda}.
{let x:Bear = Bear.panda}
{br} x has value {value x}.

||列挙型の変数に値を代入するためのサンプルです（パターン２）
||実行結果を確認してください
{let y:any = x}
{br} Now y has value {value y} and runtime type {type-of y}.
```

実行結果

```
The value Bear.panda is of type Bear.
x has value Bear.panda.
Now y has value Bear.panda and runtime type Bear.
```

【列挙型の変数】2

サンプルコード

```
|| 列挙型を定義する
{define-enum Bear
    polar, grizzly, panda, pooh
}

|| 列挙型の対応する要素に自動的にキャストされることを確認するためのサンプルです
||  コードと実行結果を確認してください
{let my-bear:Bear = "panda"}
{br} Initially my-bear has value {value my-bear}.

{set my-bear = "pooh"}
{br} Now my-bear has value {value my-bear}.
```

実行結果

```
Initially my-bear has value Bear.panda.

Now my-bear has value Bear.pooh.
```

【列挙型の変数】3

サンプルコード

```
|| 列挙型を定義する
{define-enum Bear
    UNKNOWN,
    polar,
    grizzly,
    panda,
    pooh
}
|| 変数を定義する
{let my-bear:Bear}

||列挙型の変数の既定値は、その型の最初の要素です
||これを確認するためのサンプルです。実行結果を確認してください
{value my-bear}
```

実行結果

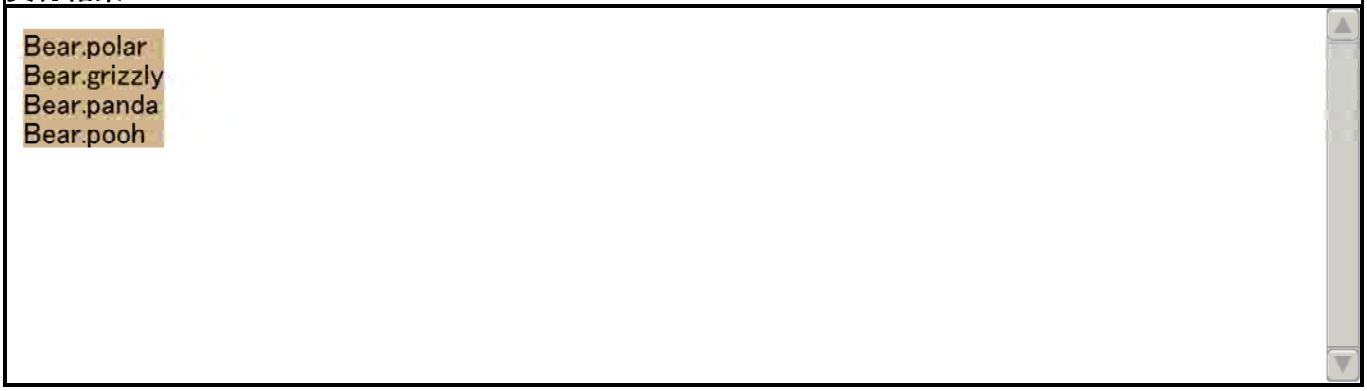
```
Bear.UNKNOWN
```

【列挙型の値の繰り返し処理】

サンプルコード

```
||-----------------------------------------
||以下のコードは、列挙型に対する繰り返し処理を示したサンプルです。
||-----------------------------------------
|| 列挙型を定義する
{define-enum Bear
    polar = "endangered",
    grizzly = "threatened",
    panda,
    pooh = "abundant"
}

|| 変数を定義する
{let bears:VBox =
    {VBox
        background = "tan"
    }
}

|| for ループで列挙型の値を簡単に繰り返し処理できます
{for bear-type in Bear do
    {bears.add bear-type}
}

|| 繰り返しの結果を表示する
||  実行結果を確認してください
{value bears}
```

実行結果



【データ型の操作】

【式が指定されたデータ型を持つかどうかの確認】1

サンプルコード

```
||------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||------------------------------------------

||isa演算子はリテラル値が type 型の変数に格納できる場合、
||表示を変更せずに true を返します。それ以外の場合は false を返します。
98 isa int... {value 98 isa int}

98 isa int8... {value 98 isa int8}

98 isa int32... {value 98 isa int32}

98 isa int64... {value 98 isa int64}

98 isa float... {value 98 isa float}

98 isa double... {value 98 isa double}

98 isa char... {value 98 isa char}
```

実行結果

```
98 isa int... true
98 isa int8... false
98 isa int32... true
98 isa int64... false
98 isa float... false
98 isa double... false
98 isa char... false
```

【式が指定されたデータ型を持つかどうかの確認】2

サンプルコード

```
||------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||------------------------------------------

||isa演算子は変数のデータ型が type またはその型の別名であれば true を返します。
||それ以外の場合は false を返します。
98 (an int) isa int...
{value
    let x:int = 98
    x isa int
}

98 (an int) isa int32...
{value
    let x:int = 98
    x isa int32
}

98 (an int8) isa int...
{value
    let x:int8 = 98
    x isa int
}
```

実行結果

```
98 (an int) isa int... true
98 (an int) isa int32... true
98 (an int8) isa int... false
```

【式が指定されたデータ型を持つかどうかの確認】3

サンプルコード

```
||------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||------------------------------------------

||isa演算子は変数のデータ型が type または type のサブクラスのいずれかであれば true を返します。
||それ以外の場合は false を返します。
{define-class Animal}
{define-class Cow {inherits Animal}}

{let harriet:Cow = {Cow}}

harriet isa Cow is ... {value harriet isa Cow}

harriet isa Animal is ... {value harriet isa Animal}

harriet isa double is ... {value harriet isa double}
```

実行結果

```
harriet isa Cow is ... true
harriet isa Animal is ... true
harriet isa double is ... false
```

【式のデータ型の取得】1

サンプルコード

```
||-----------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||-----------------------------------------------

||プリミティブ データ型の type-of 式のサンプルを示します
{let i:int = 3}
The data type of i is... {type-of i}

{let c:char = 'a'}
The data type of c is... {type-of c}
```

実行結果

```
The data type of i is... int

The data type of c is... char
```

【式のデータ型の取得】2

サンプルコード

```
||----------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||----------------------------------------------

|| any データ型の type-of 式のサンプルを示します
{let a:any = 3}
The data type of a is... {type-of a}

{let b:any = 'a'}
The data type of b is... {type-of b}
```

実行結果

```
The data type of a is... int

The data type of b is... char
```

【式のデータ型の取得】3

サンプルコード

```
||-----------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||-----------------------------------------

|| クラス型変数のある type-of を使用するサンプルです
{|let c:VBox = {VBox}|}
The data type of c is... {type-of c}

||type-of 式は式の実行時データ型を返します
{|let d:Box = {VBox}|}
The data type of d is... {type-of d}
```

実行結果

```
The data type of c is... VBox
The data type of d is... VBox
```

【式のデータ型の取得】4

サンプルコード

```
||------------------------------------------
||以下のコードは、データ型を確認するためのサンプルです。
||コードと実行結果を確認してください
||------------------------------------------

||type-of を使用して任意の式のデータ型をクエリすることもできます
||以下のコードと実行結果を確認してください
The data type of 3 is ... {type-of 3}

The data type of 3 * 7.0 is ... {type-of (3 * 7.0)}

The data type of 'a' is ... {type-of 'a'}

The data type of "xyz" is ... {type-of "xyz"}

The data type of {StringBuf "xyz"} is ... {type-of {StringBuf "xyz"}}
```

実行結果

```
The data type of 3 is ... int
The data type of 3 * 7.0 is ... double
The data type of 'a' is ... char
The data type of "xyz" is ... String
The data type of xyz is ... StringBuf
```

【サブタイプのリレーションシップの確認】

サンプルコード

```
||----------------------------------------------------
||以下のコードは、サブタイプを説明するためのサンプルです。
||サブタイプの詳細は、Curl開発者ガイドを確認してください。
||
||subtype-of? メソッドを使用して
||データ型が第 2 データ型のサブタイプであるかどうかを判断しています
||コードと実行結果を確認してください
||----------------------------------------------------

String subtype of String: {(String asa Type).subtype-of? String}

String subtype of #String: {(String asa Type).subtype-of? #String}

#String subtype of String: {(#String asa Type).subtype-of? String}

String subtype of Object:  {(String asa Type).subtype-of? Object}

String subtype of any:  {(String asa Type).subtype-of? any}

int subtype of int64: {int.subtype-of? int64}
```

実行結果

```
String subtype of String: true
String subtype of #String: true
#String subtype of String: false
String subtype of Object: true
String subtype of any: true
int subtype of int64: false
```

【データ型の変換】

サンプルコード

```
||以下のコードは、データ型変換のサンプルです。
{value
    || asa 演算子を使用して値をあるデータ型から他のデータ型へ明示的に 変換することができます
    let d:double = 37.7

    ||実行結果を確認してください
    d asa int
}
```

実行結果

```
37
```

【プリミティブ データ型の変換】1

サンプルコード

```
{value
    ||整数と浮動小数点データ型で変換した際の挙動を確認するサンプルです
    ||コードと実行結果を確認してください
    let i:int
    set i = 37.7 asa int

    i
}
```

実行結果

```
37
```

【プリミティブ データ型の変換】2

サンプルコード

```
{value
    || 大きな整数値を浮動小数点データ型の変数に代入する場合、
    || コンピュータによっては浮動小数点数を格納する方法により、情報が失われることがあります
    ||コードと実行結果を確認してください
    let f:float
    set f = 2147483525

    f asa int
}
```

実行結果

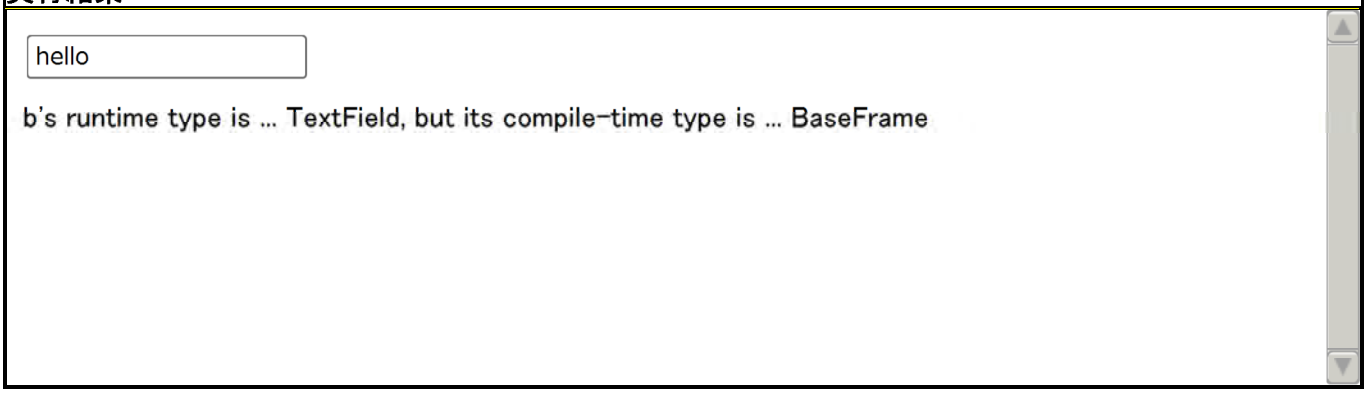
```
2147483520
```

【クラス型の変換】1

サンプルコード

```
||クラス型の変換を説明するためのサンプルです。
||詳細はCurl開発者ガイドを確認してください
|| 以下のコードでは、代入ステートメントは暗黙に TextField を BaseFrame に変換しています。
{let b:BaseFrame = {BaseFrame}}

{let t:TextField = {TextField value = "hello"}}

{set b = t}


{value b}

b's runtime type is ... {type-of b},
but its compile-time type is ... {compile-time-type-of b}
```

実行結果

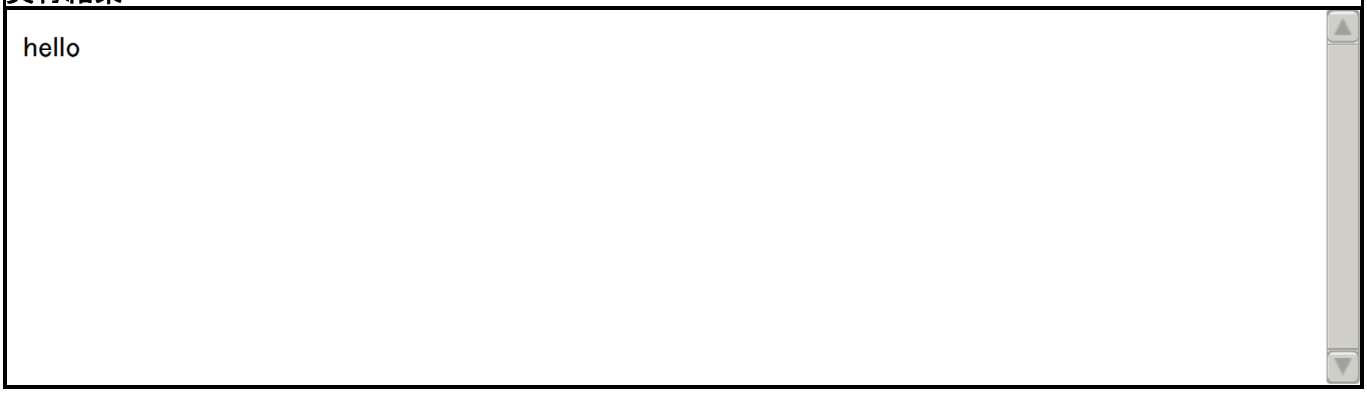


【クラス型の変換】2

サンプルコード

```
{value
    ||クラス型の変換を説明するためのサンプルです。
    ||詳細はCurl開発者ガイドを確認してください
    ||以下のコードでは、value ゲッターを使用するためのサンプルとして、
    ||オブジェクトを明示的に TextField に変換しています
    let b:BaseFrame = {BaseFrame}
    let t:TextField = {TextField value = "hello"}

    set b = t

    (b asa TextField).value
}
```

実行結果



【数量と単位の使い方】

【数量の表示】

サンプルコード

```
||以下のコードは、数量のサンプルです。
||実行結果を確認してください
{text 1inch displays as ... {value 1inch}}

{text 1foot displays as ... {value 1foot}}

{text 1meter displays as ... {value 1meter}}

{text 1mile displays as ... {value 1mile}}
```

実行結果

```
1inch displays as ... 0.0254m

1foot displays as ... 0.3048m

1meter displays as ... 1m

1mile displays as ... 1609.34m
```

【数量の表示】(format)

サンプルコード

```
||以下のコードは、formatを用いて数量を表示したサンプルです。
||距離を 1inch で除算する場合、元の数量のインチ数が単位なし数値として得られます。
||実行結果を確認してください
{format "%g is equal to %.2g inches", 10cm, 10cm/1inch}
{format "%g is equal to %g inches", 1yard, 1yard/1inch}
```

実行結果

```
0.1m is equal to 3.9 inches

0.9144m is equal to 36 inches
```

【組み込みの数量型】(isa)

サンプルコード

```
||以下のコードは、isa プリミティブ用いて数量の型を確認したサンプルです。
||実行結果を確認してください
{let alpha:Angle = 0.5rad}
{paragraph "alpha isa Angle" is...} {value alpha isa Angle}
{paragraph "alpha isa Distance" is...} {value alpha isa Distance}
```

実行結果

```
"alpha isa Angle" is...

true

"alpha isa Distance" is...

false
```

【組み込みの数量型】(type-of)

サンプルコード

```
||以下のコードは、数量単位とtype-ofを使ったサンプルです。
{let rate:Speed = 55mph,
    time:Time = 1hr + 15min}
||実行結果を確認してください
{paragraph rate * time is of type {type-of rate * time}}
{paragraph rate / time is of type {type-of rate / time}}
{paragraph time / rate is of type {type-of time / rate}}
{paragraph time * time is of type {type-of time * time}}
```

実行結果

```
rate * time is of type Distance

rate / time is of type Acceleration

time / rate is of type {type-of 1(m^-1*s^2)}

time * time is of type {type-of 1(s^2)}
```

【演算の数量】(SI 基本単位)

サンプルコード

```
||----------------------------------------------------------------
||数量は、値を指定するために使用した単位に関係なく、SI 基本単位を使用します
||以下のコードにて、数量の有効な操作の結果が基本単位で表示されることが確認できます
||----------------------------------------------------------------
||実行結果を確認してください
{value 1m + 1in}

{value 1in + 1m}

{value 1m - 1in}

{value 2ft * 3ft}

{value 50mi / 2hr}
```

実行結果

```
1.0254m
1.0254m
0.9746m
0.557418(m^2)
11.176(m*s^-1)
```

【演算の数量】(ゼロ除算)

サンプルコード

```
||ゼロでは除算できません。ゼロで除算した場合、
||<infinity> や nan などの非数値の結果が表示されます。
{value 7m / 0s}
```

実行結果

```
<infinity>(m*s^-1)
```

【演算の数量】1

サンプルコード

```
||以下のコードでは、異なる数量単位での演算を行っています
||実行結果を確認してください
{value 1meter + 5centimeters}

{value 1minute - 15seconds}

Is 55mph greater than 70km/hr? ... {value 55mph > 70(km/hr)}
```

実行結果

```
1.05m

45s

Is 55mph greater than 70km/hr? ... true
```

【演算の数量】2

サンプルコード

```
||----------------------------------------------------
|| 乗算と除算では、オペランドを同じ型にする必要はありません。
|| （加算や減算ではできませんが）以下のコードのように
|| 同じ数量型または異なる数量型の数量の乗算または除算ができます
||----------------------------------------------------
||実行結果を確認してください
{value 10meters * 2meters}
{value 10meters / 2seconds}
```

実行結果

```
20(m^2)

5(m*s^-1)
```

【演算の数量】3

サンプルコード

```
||以下のコードのように単位付数量と単位なし数量を演算できます
||実行結果を確認してください
{value 2m * 3}

{value 12seconds / 4}

{value 25 + 10%}
```

実行結果

```
6m

3s

25.1
```

【数量と数値との比較】1

サンプルコード

```
||根本的に異なる数量型の数量（つまり、同じ基本単位で表現できない数量）は比較できません
||そのため、両方の値を距離にして、以下のように x を 0ft と比較します。
{let x:Distance = 6ft}
Is x greater than 0 feet? ... {value x > 0ft}
```

実行結果

```
Is x greater than 0 feet? ... true
```

【数量と数値との比較】2

サンプルコード

```
||数量から数値を抽出するには、以下のように
||適合する単位で除算すれば、単位なしの指数が得られます
{let x:Distance = 6ft}

The value of x is {value x}

The number of meters in x is {value x/1m}.

||SI-value プロシージャにて任意の数量から数値を抽出することができます
The SI value of x is {SI-value x}.

Is x's numerical value greater than 0? ... {value {SI-value x} > 0}
```

実行結果

```
The value of x is 1.8288m
The number of meters in x is 1.8288.
The SI value of x is 1.8288.
Is x's numerical value greater than 0? ... true
```

【数値計算ライブラリ プロシージャでの数量】

サンプルコード

```
||数値計算ライブラリの一部のプロシージャでは数量をサポートします
{cos 60deg}

{sin 0.5rad}

{tan 45deg}

{value {atan 1.0} == 45deg}
```

実行結果

```
0.5
0.479426
1
true
```

【単位なし数量】

サンプルコード

```
||数量の演算結果、単位にゼロ (0) の累乗が伴う場合は、単位なし数量が生成されます
||コードと実行結果を確認してください
{value 1m / 20cm}
```

実行結果

```
5
```

【パーセント】

サンプルコード

```
||% サフィックスは、その前の数値を 100 で除算する換算係数として作用します
||コードと実行結果を確認してください
Is 50% equal to .5? ... {value 50% == .5}

{value 30% + 50%}

{value 25% * 200m}
```

実行結果

```
Is 50% equal to .5? ... true
0.8
50m
```

【数量の変換】

サンプルコード

```
|| 数値から数量を作成するためのサンプルコードです
{let f:float = 5}

{value f * 1m}

{value f * 1yd}

{value f * 1s}
```

実行結果

```
5m
4.572m
5s
```

【値と数量の変換】

サンプルコード

```
|| 数量から数値を抽出するためのサンプルコードです
{let q:Distance = 5m}

{value q / 1m} meters

{value q / 1cm} centimeters

{value q / 1mm} millimeters

{value q / 1yards} yards

{value q / 1foot} feet
```

実行結果

```
5 meters
500 centimeters
5000 millimeters
5.46807 yards
16.4042 feet
```

【SI-value の使用】

サンプルコード

```
||以下のコードはSI-valueを使ったサンプルです。
||詳細は、Curl開発者ガイドを確認してください
{value
    let my-quantity:Distance = 3feet
    {SI-value my-quantity}
}
{value
    let my-quantity:Time = 2minutes
    {SI-value my-quantity}
}
```

実行結果

```
0.9144

120
```

【クラスの使い方】

【Object クラス】

サンプルコード

```
{value
    ||Curl 言語はシングル ルートです。
    ||Object クラスは Curl 言語のすべてのクラスのルートです
    ||詳細は、Curl開発者ガイドを確認してください
    let x:Object = {HBox "Hello"}

    {(x asa HBox).add " World"}
    x
}
```

実行結果

【変数を持つオブジェクトと持たないオブジェクトの使用】1

サンプルコード

```
||Curl 言語は、以下のように
||変数にオブジェクトを代入せずにオブジェクトを作成して使用することができます
{HBox
    {CommandButton},
    {CommandButton}
}
```

実行結果

【変数を持つオブジェクトと持たないオブジェクトの使用】2

サンプルコード

```
{value
    ||以下のように変数 h を使用して add メソッドを呼び出して同じ結果を生成できます
    let h:HBox = {HBox}
    let b:CommandButton = {CommandButton}

    {h.add b}
    {h.add {CommandButton}}
    h
}
```

実行結果

【アクセッサ】

サンプルコード

```
||プライベートフィールドへアクセスするためのサンプルコード
||アクセッサに関する詳細は、Curl開発者ガイドを確認してください
{define-class public Person

  field private _age:int = 0

  {getter public {age}:int
    {return self._age}
  }

  {setter protected {age years:int}
    set self._age = years
  }
}
```

【アクセッサ(public-get)】

サンプルコード

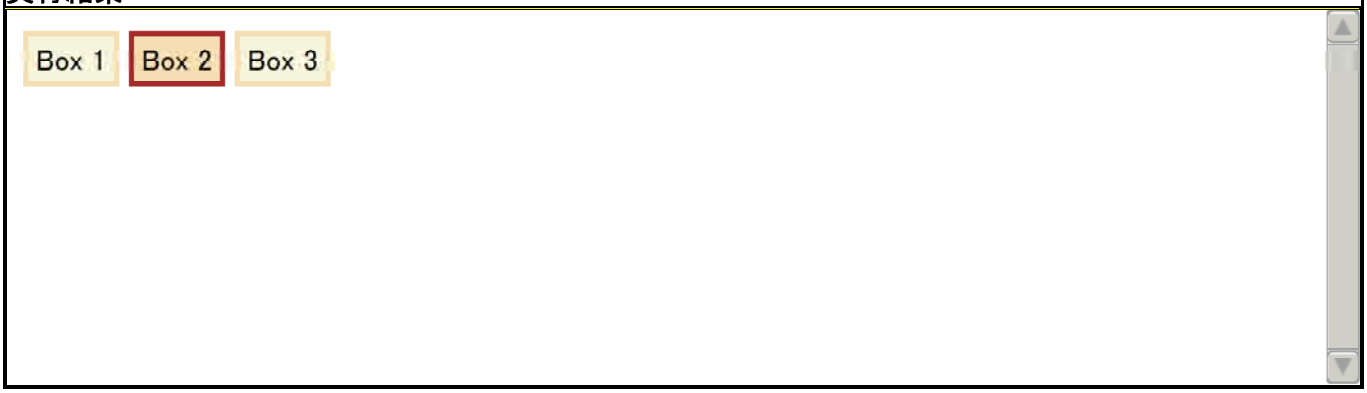
```
||以下のコードはpublic-getを使ったサンプルです
||詳細はCurl開発者ガイドを確認してください
{define-class public Person
  field public-get protected-set age:int = 0
}
```

【ローカル オプションの宣言】

サンプルコード

```
|| ローカル オプション宣言のためのサンプルコードです
|| 詳細は、Curl開発者ガイドを確認してください
{define-class MyBox {inherits HVBox}

  {local-option public my-highlight?:bool = false
    {if self.my-highlight? == true then
        set self.background="wheat"
        set self.border-color="brown"
    }
  }

  {constructor {default ...}
    {construct-super
        background="beige",
        border-color="wheat",
        border-width=2pt,
        margin=3pt,
        ...
    }
  }
}
{value
    let box1:MyBox = {MyBox "Box 1"}
    let box2:MyBox = {MyBox my-highlight?=true, "Box 2"}
    let box3:MyBox = {MyBox "Box 3"}

    {spaced-hbox box1, box2, box3}
}
```

実行結果

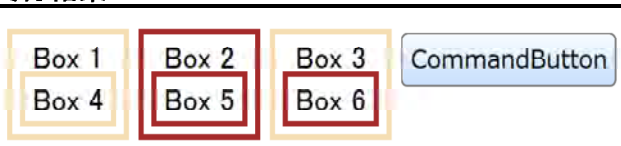


【非ローカル オプションの定義と宣言】

サンプルコード

```
|| 非ローカル オプション宣言のためのサンプルコードです
|| 詳細は、Curl開発者ガイドを確認してください
{define-nonlocal-option public my-highlight?:bool = false}

{define-class MyBox {inherits HVBox}
  {nonlocal-option public my-highlight?:bool
    {if self.my-highlight? == true then
        set self.border-color="brown"
     else
        set self.border-color = "wheat"
    }
  }
  {constructor {default ...}
    {construct-super
        border-color="wheat",
        border-width=2pt,
        margin=3pt,
        ...
    }
  }
}
{value
    let box4:MyBox = {MyBox "Box 4"}
    let box5:MyBox = {MyBox "Box 5"}
    let box6:MyBox = {MyBox my-highlight?=true, "Box 6"}

    let box1:MyBox = {MyBox "Box 1", box4}
    let box2:MyBox = {MyBox my-highlight?=true, "Box 2", box5}
    let box3:MyBox = {MyBox "Box 3", box6}

    {spaced-hbox box1, box2, box3,
        {CommandButton
            {on Action do
                {unset box2.my-highlight?}
            }
        }
    }
}
```

実行結果



【オプションの継承】

サンプルコード

```
{value
    ||オプションはサブクラス化によって継承されます。
    ||ただし、すべてのオプションがそれを継承するクラスで意味を持つとは限りません
    ||RadioButton は GraphicOptions のサブクラスです。
    let rb:RadioButton = {RadioButton label="Hello World!"}


    ||backgroundオプションを継承します。こちらは有効です。
    set rb.background = "beige"

    ||cell-marginオプションを継承します。こちらは意味を持ちません。
    set rb.cell-margin = 1cm

    ||実行結果を確認してください
    {VBox rb}
}
```

実行結果