

# Irenka DOM Specification

---

Ver. 0.1.0

2007/11/11

The Ashikunep Kotan

# もくじ

---

第1章 イントロダクション.....	1
第2章 Irenka DOM .....	2
第1節 DOM要素.....	2
第2節 Irenka DOMの構造.....	5
第3節 Irenka DOMの操作.....	7
第4節 DOM要素の比較 .....	9
第3章 宣言と参照 .....	12
第1節 マスタ参照.....	13
第2節 スレーブ参照.....	14
第4章 型.....	15
第1節 基本型.....	16
第2節 宣言型.....	16
第3節 型変数.....	17
第4節 ワイルドカード .....	17
第5節 配列型.....	18
第6節 特殊型.....	18
第5章 型の宣言 .....	20
第1節 クラス .....	20
第2節 インターフェース .....	21
第3節 列挙 .....	21
第4節 注釈 .....	22
第6章 メンバ .....	23
第1節 フィールド.....	24
第2節 列挙定数 .....	24
第3節 メソッド .....	24

第4節	コンストラクタ	25
第5節	注釈要素	26
第6節	初期化子	26
第7節	メンバ型	27
第7章	ジェネリック参照	28
第1節	型変数の宣言	28
第2節	総称化コンテキスト	29
第8章	文	30
第1節	式文	30
第2節	ラベル付き文	30
第3節	ブロック	31
第4節	ローカルクラス宣言	31
第5節	ローカル変数宣言文	32
第6節	空文	32
第7節	if文	32
第8節	assert文	33
第9節	switch文	33
第10節	while文	34
第11節	do文	34
第12節	for文	34
第13節	拡張for文	35
第14節	break文	35
第15節	continue文	35
第16節	return文	36
第17節	throw文	36
第18節	synchronized文	36
第19節	try文	37

第9章 式	38
第1節 括弧付きの式	38
第2節 字句上のリテラル	39
第3節 クラス・リテラル	39
第4節 this	40
第5節 クラス・インスタンス生成式	40
第6節 配列生成式	40
第7節 変数アクセス式	41
第8節 配列長参照式	41
第9節 メソッド起動式	41
第10節 配列アクセス式	42
第11節 単項演算式	42
第12節 キャスト式	43
第13節 二項演算式	43
第14節 instanceof式	45
第15節 条件演算式	45
第16節 代入演算式	45
第10章 宣言の属性	47
第1節 注釈	47
第2節 修飾子	47
第3節 ドキュメンテーションコメント	48
第11章 Javadoc	49
第1節 ドキュメント本体	49
第2節 ブロック要素	50
第3節 タグ	50
第4節 テキスト	51
第5節 インラインタグブロック	51

第 12 章 ソースプログラム.....	52
第 1 節 コンパイル単位 .....	52
第 2 節 位置情報 .....	54
参考文献 .....	55
用語索引 .....	56

## 第1章 イン트로ダクション

---

Irenka Document Object Model (Irenka DOM)は Irenka がプロジェクトをコンパイルする際に使用するモデルで、プロジェクトに含まれるプログラムが持つそれぞれの意味と対応します。これは木構造をしており、それぞれのノードを DOM 要素と呼び、それぞれプロジェクトに含まれるプログラム片が持つ意味と対応します。DOM 要素には様々な種類があり、プロジェクト全体を構成するパッケージツリーや、プロジェクトで宣言された個々のクラスなど大きなものから、ドキュメンテーションコメントや整数リテラルなどの小さなものまで様々なプログラム片を表現することができます。

プログラムには字句、構文、意味の 3 種類の構造があります。最初の字句とは文字の並びであるプログラムを単語(キーワード、識別子、数値、記号など)の列とみなしたもので、次の構文とは字句の並びを構造化した文や式などの要素からなる構造です。最後の意味とは構文構造に含まれる要素を解釈しプログラムとしての意味を与えた構造です。Irenka DOM はこのうち、プログラムの意味構造に対応するモデルで、構文上の意味を持たない細かな差異を無視して意味のみを取り扱うことができます。たとえばプログラミング言語 Java において、"Object"と表記されたものと"java.lang.Object"と表記されたものは、ほとんどの場合で java.lang.Object クラス型を表現します。字句構造や構文構造においてはこれら 2 つの表記を異なるものとして取り扱いますが、Irenka ではどちらも java.lang.Object 型であるという意味に注目し、同じ値を持つ要素として取り扱うことができます。

Irenka の主要機能である Hack は、この Irenka DOM を対象に行われます。Irenka はプロジェクトから Irenka DOM を生成しコンパイルを終了させるまでの間に、ユーザが定義した Hack を実行します。このとき、Hack の対象を検出するための Search Query は Irenka DOM に対して行われ、プログラムの操作は Irenka DOM 上の対応する DOM 要素を変更することによって行われます。Irenka DOM はプログラムの意味を表現していますので、ユーザはプログラムの表記や複数のプログラム間の参照関係を意識することなく、それらが持つ意味だけを考慮してプログラムの監査や操作を行うことができます。

本書では、Irenka DOM の構造や、それぞれの DOM 要素の特徴について詳しく紹介します。Irenka DOM を検索する Irenka Search Query については Irenka Search Query Specification、開発環境である Irenka Studio の使い方や Irenka DOM の扱い方に関するサンプルなどについては Irenka Studio Users' Guide を参照してください。

## 第2章 Irenka DOM

---

Irenkaはプログラムをコンパイルする際に、それらをDocument Object Model (DOM)に変換し、その上で全ての処理を行います。このモデルをIrenka DOMと呼び、対応するソースプログラムと同等の意味構造を持ちます。Irenka DOM全体はコンパイル対象プロジェクトを表現する木構造をなして、ツリー上の各ノードをDOM要素(第 1 節)と呼びます。

Irenka DOMを構成するそれぞれのDOM要素は、プロジェクトに含まれるプログラム片が持つ意味と対応します。これには様々な粒度があり、クラスの宣言などの大きなものから、整数リテラルなどの小さなものまで、プログラム内で意味を持つものは全てDOM要素で表現することができます。そしてそれぞれのDOM要素はツリー状に構造化(第 2 節)され、Irenka DOM全体のモデルを構成します。

また、DOM要素を操作する(第 3 節)と、対応するプログラムも意味構造に従って変更されます。それぞれのDOM要素はその種類ごとにインターフェースが公開されており、そのインターフェースを介してJavaプログラムの監査や操作を行うことができます。これらのインターフェースはJavaのインターフェースとして作成されており、本文書ではそれぞれのインターフェースと、それに対応するDOM要素の仕様について規定します。なお、ほとんどのDOM要素を表すインターフェースはorg.ashikunep.irenka.domパッケージ上に宣言されています。本文書では、とくに断りのない限り単純名で表現された型はorg.ashikunep.irenka.domパッケージに宣言された型を表現します。

### 第1節 DOM 要素

Irenka DOM 上に出現する各要素を、DOM 要素と呼びます。すべての DOM 要素はCtElement インターフェースを実装しており、要素が表現する意味ごとにCtElement のサブインターフェースが定義されています。

CtElement インターフェースは大きく 4 種類の機能を提供します。詳しくは各項を参照してください。

1. DOM要素の属性に関する機能(第 1 項)
2. Irenka DOMの構造に関する機能(第 2 項)
3. DOM要素の操作に関する機能(第 3 項)
4. DOM要素に対応するソースプログラムに関する機能(第 4 項)

#### 第1項 DOM 要素の属性に関する機能

表 1は、CtElementが提供するDOM要素の属性に関する機能を表します。DOM要素は様々な属性を持っており、これらの機能はそれぞれの属性を参照したり変更したりすることができます。個々のメソッドについて、より詳しくはIrenka End User API Referenceを参照してください。

メソッド *getElementKind* は対象の DOM 要素の種類を返し、それぞれの DOM 要素の種類は列挙 *ElementKind* によって表現されます。これは DOM 要素の種類ごとに対応する列挙定数が用意されており、この値を検査することによって DOM 要素の種類を調べることができます。一部の実装では一つのクラスで複数の DOM 要素として表現可能なものがありますので、*instanceof* 式による検査では要素の種類を特定できない場合があります。

メソッド *exists* と *isSynthetic* は対象の DOM 要素の存在に関する情報を返します。Irenka DOM では各 DOM 要素の操作 (第 3 節) を単純にするため、存在しない DOM 要素を null で表現するのではなく、メソッド *exists* が偽を返す要素として表現します。このような DOM 要素を存在しない DOM 要素と呼びます。また、ソースプログラム上で省略されていた要素を DOM 要素として表現する<sup>1</sup> 場合、メソッド *isSynthetic* が真を返すような DOM を生成します。これは Irenka によって自動的に挿入された DOM 要素を表し、合成された DOM 要素と呼びます。

メソッド *isParentFrozen*、*isChildFrozen* は DOM 要素の親子に関する変更可能性に関する情報を返します。偽を返した場合、対応する要素を変更することはできません。また、メソッド *freezeParent*、*freezeChild* を呼び出すことによって、これらの状態を変更することができます。

また、メソッド *getClientStorage* および *putClientStorage* を利用して、新しい属性を定義することもできます。このメソッドは `org.ashikunep.irenka.util.ClientStorage` という追加情報を保持するためのインターフェースを持ったデータ構造を取り扱うことができます。新しい属性を作成するには *ClientStorage* インターフェースを実装したクラスを作成し、そこに必要な属性を定義します。このような情報を、外部属性と呼びます。なお、*ClientStorage* は Irenka が提供する一部のパーサやエミッタなども利用します。たとえば、型の単純名や完全限定名による表記は DOM 要素で表現できないため、パーサやエミッタは *ClientStorage* を利用してそれらの外部属性を保持させています。

表 1. *CtElement* の機能 (要素の属性に関する機能)

メソッド名	概要
<i>getElementKind</i>	この要素の種類を返す
<i>exists</i>	この要素が存在するか検査する
<i>isSynthetic</i>	この要素が Irenka によって合成された要素かどうか検査する
<i>isParentFrozen</i>	この要素の親要素を変更可能かどうか検査する
<i>isChildFrozen</i>	この要素の子要素を変更可能かどうか検査する
<i>freezeParent</i>	この要素の親要素を変更不可能にする
<i>freezeChild</i>	この要素の子要素を変更不可能にする
<i>getClientStorage</i>	この要素に付与された追加情報を返す
<i>putClientStorage</i>	この要素に追加情報を付加する

<sup>1</sup> 表記 "public class A {...}" の意味は "public class A extends java.lang.Object {...}" です。このような場合、クラス A は親クラスに `java.lang.Object` 型を表現する DOM 要素を持ち、これは合成された DOM 要素です。

## 第2項 Irenka DOM の構造に関する機能

表 2はIrenka DOMの構造に関する機能を表します。Irenka DOMは木構造をなし、それぞれのDOM要素は1つの親要素と複数の子要素を持つことができます。各要素が持てる親要素および子要素は、それぞれのDOM要素の種類によって決まります。また、いくつかのDOM要素は親や子を持ってない場合があります。

メソッド *getParent* は DOM 要素の親要素を取得することができます。ほとんどの DOM 要素は親要素をもっていますが、Irenka DOM のルートとなった要素や、操作の過程で出現する新しい DOM 要素などは親要素を持ちません。親要素が存在しない場合、この呼び出しは null を返します。

メソッド *getLocationInParent* は DOM 要素の親からの位置を取得することができます。親要素から見た位置とは、たとえば"1 + 2"という DOM 要素が存在したとき、これは"1"および"2"という子要素を持ちます。これらの子要素はどちらも"1 + 2"という DOM 要素を親要素に持ちますが、"1"は親要素から見て左項、"2"は親要素から見て右項に位置しています。このような親要素から見た位置は `org.ashikunep.irenka.dom.navigator.ChildLocation` 型のオブジェクトでカプセル化され、このメソッドを利用して取得することができます。

メソッド *getChild* は DOM 要素の子要素を取得することができます。子要素の特定には前述の `ChildLocation` を利用します。個々の DOM 要素に対応するインターフェースは、その種類に適した子要素を取得するためのメソッドが公開されています。通常は個々のインターフェースに公開されたメソッドを利用の方が安全です。

メソッド *getProperty* は DOM 要素のプロパティを取得することができます。プロパティについて詳しくは第 2 節第 1 項を参照してください。

表 2. `CtElement` の機能(Irenka DOM の構造に関する機能)

メソッド名	概要
<code>getParent</code>	親要素を返す
<code>getLocationInParent</code>	親要素から見たこの要素の位置を返す
<code>getChild</code>	子要素を返す
<code>getProperty</code>	プロパティを返す

## 第3項 DOM 要素の操作に関する機能

表 3はDOM要素の操作に関する機能を表します。

メソッド *substitute* は全てのDOM操作の基本となるもので、現在のDOM要素がある位置に他のDOM要素を配置することができます。置換および複製の動作について詳しくは第 3 節を参照してください。また、DOM要素の比較に関しては第 4 節を参照してください。

表 3. `CtElement` の機能(操作に関する機能)

メソッド名	概要
-------	----

substitute	この要素を別の要素で置換する
copy	この要素を複製する
equals	この要素と別の要素の同値性を比較する

## 第4項 DOM 要素に対応するソースプログラムに関する機能

表 4はDOM要素に対応するソースプログラムに関する機能を表します。これらの機能はDOM要素が表記されたソースプログラムに関する情報を取得するために存在し、Irenka DOMとは直接関係ありません。これらについては、第 12 章で詳しく解説します。

表 4. CtElement の機能(ソースプログラムに関する機能)

メソッド名	概要
getCompilationUnit	この要素が属するコンパイル単位を返す
getCorrespondedFile	この要素が記載されたソースプログラムを返す
getLocation	この要素が記載されたソースプログラム内の位置を返す

## 第2節 Irenka DOM の構造

Irenka DOMは無名パッケージ(JLS3-7.4.2 Unnamed Packages)を表現する CtPackage<sup>Master</sup> 型のオブジェクトを根とする、木構造で表現されます。まず、対象プロジェクトの無名パッケージ上に存在するパッケージメンバ(JLS3-7.1 Package Members)が無名パッケージの直接の子要素として配置され、さらにそのパッケージメンバが再帰的にパッケージメンバの子要素に持ちます。また、パッケージメンバには型の宣言も含まれ、型の宣言は第 5 章に定義される方法でツリーを構成します。たとえば、"org.ashikunep.irenka.dom.CtElement"は図 1のようにIrenka DOM内で表現されます。なお、CtElementインターフェースはさらに様々な子要素がありますが、ここでは省略しています。

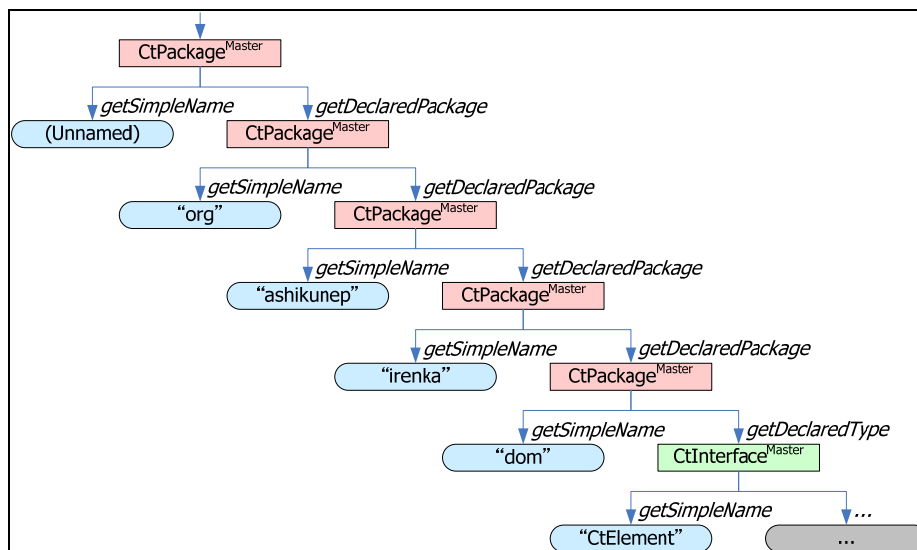


図 1. org.ashikunep.irenka.dom.CtElement が配置される様子

なお、パッケージ(CtPackage)や型の宣言(CtDeclaredType)は Irenka DOM の名前空間を構築するという意味から、名前空間を表す CtNamespace 型のインターフェースのサブインターフェースとして宣言されています。そのため、すべてのパッケージメンバは CtNamespace のサブタイプで表現されます。

## 第1項 DOM 要素のプロパティ

それぞれの DOM 要素は、その種類に適したプロパティを有します。プロパティの種類には、次のようなものがあります。

1. 子要素を表現する DOM 要素
2. 子要素を表現する DOM 要素のリスト
3. 列挙定数
4. その他の定数値(String や boolean など)

子要素とプロパティは異なるもので、前者の子要素は Irenka DOM の構造における親子関係について表し、後者のプロパティは DOM 要素が持つ情報のことを表しています。たとえば、メソッド宣言(第 6 章第 3 節)にはメソッド仮引数宣言の一覧を含めることができます。個々の"メソッド仮引数宣言"はメソッド宣言から見ると子要素で、"メソッド仮引数宣言の一覧"はメソッド宣言を表現する DOM 要素が持つプロパティです。このプロパティは個々のメソッド引数宣言を表現する DOM 要素のリストで表現されます。

それぞれの DOM 要素は複数のプロパティを持つことがあり、各プロパティの種類は列挙 `org.ashikunep.irenka.dom.navigator.Navigation` 内で定義されています。この列挙定数を CtElement が定義するメソッド `getProperty` の引数に指定することで適切なプロパティを取得することができます。そして、個々のプロパティの種類を調べる場合には、Navigation が定義するメソッド `getKind` を利用します。これは列挙 `NavigationKind` に定義されたプロパティの種類に対応する定数を返します。

## 第2項 DOM 要素の親子関係

各要素は 1 つの親要素と、その要素の種類に適した複数の子要素を持つことができます。親子は双方向に接続されていて、相互に参照が可能となっています。親要素を取得するには CtElement が提供するメソッド `getParent` を利用し、子要素を取得するにはメソッド `getChild`、または DOM 要素を表す個々のインターフェースに定義されたメソッドを利用します(第 1 節第 2 項)。

省略可能な子要素が実際に省略されている場合、それらはできる限り null 参照ではなく、存在しない DOM 要素として表現します。または、子要素が属するプロパティがリストとして表現されている場合は、要素数 0 のリストとして表現します。このような設計は、DOM 要素の操作に関する API を単純にし、DOM 要素に対する変更は置換、またはリストに対する変更のいずれかで実現することができます。

## 第3節 Irenka DOM の操作

Irenka DOMを操作する場合、変更対象のDOM要素に対して操作を行います。これには、次のような方法があります。

1. 現在の要素を他の要素で置換する
  1. `CtElement#substitute(CtElement)`
2. 子要素を保持するプロパティを変更する
  1. 子要素に対して `CtElement#substitute(CtElement)` を実行
  2. 子要素のリストを保持するプロパティの内容を変更する

プロパティが子要素のリストを保持している場合、そのリストは `java.util.List<? extends CtElement>` 型のサブタイプとして表現されます。これらは特殊なリストで、そのリストに対する変更は Irenka DOM に反映されます。

Irenka DOM に対してこれらの操作を行う場合、次のような処理が自動的に行われます。

- 文脈依存変換(第 2 項)
- 親子関係の調整(第 3 項)

なお、省略可能な DOM 要素に対して `CtElement#substitute(null)` を起動すると、その要素は削除されます。ただし、その要素が null に置き換わるのではなく、省略されたことを表現する存在しない DOM 要素に置き換わることに注意してください。

### 第1項 DOM 要素の複製

DOM要素の複製を作成するには、`CtElement`が提供するメソッド`copy`を利用します。このメソッドの結果として返されるDOM要素は元の要素のディープコピーであり、さらに定義 1にある特徴を持ちます。

ある DOM 要素 `e` に対してメソッド `copy` を呼び出した結果の要素 `c` は、

次の特徴をすべて有する。

1. `c` は `e` と同一でない
2. `c` は親要素を持たない

**定義 1. メソッド `copy` によって作成される DOM 要素の特徴**

メソッド *copy*には 2 種類あり、引数に `org.ashikunep.irenka.util.CopyMonitor` を取るものと何も引数を取らないものがあります。前者の `CopyMonitor` を利用すると、この操作によって生成された全ての DOM 要素を監視することができます。ディープコピーを作成すると参照が破壊される場合があるため、それらを監視して参照先を再設定する場合などに有効です

また、メソッド *copy*はDOM要素の外部属性(第 1 節第 1 項)についても同時に複製を行います。外部属性は `ClientStorage` インターフェイスで表現され、メソッド *copy* が実行される際に定義 2の方法で外部属性を複製します。

複製元の DOM 要素を `orig` とおき、複製した DOM 要素を `copy` とおく

`orig` が持つ任意の外部属性を `attr` とおき、その登録キーを `key` とおく

1. `attr.migrate(orig)`を実行した結果が `null` となる場合
  - `attr` は `copy` に複製されない
2. `attr.migrate(orig)`を実行した結果が `null` とならない場合
  - `copy` は `attr.migrate(orig)`の結果を `key` というキーによって保持する

### 定義 2. `ClientStorage` のコピー

## 第2項 文脈依存変換

ある要素を親のプロパティに追加したり、他の要素を置換したりする場合、それぞれの要素は設定先に適した要素である必要があります。しかし、Irenka DOMの理解は複雑で間違いやすいため、直観的にDOM要素を取り扱うための機構として文脈依存変換が用意されています。これは、設定先に適さない要素を指定しまった際に可能な限り適用され、要素の設定を行う直前に設定する要素を設定先に適した要素へと変換します。これは、すべての要素に対して文脈依存変換が行えるわけではなく、定義 3にある条件を満たし、かつ文脈依存変換に適した文脈と要素でなければなりません。

設定する要素が次のいずれかであった場合、文脈依存変換が試行される

1. 要素の種類が設定先に対して適切でない
2. 設定先は参照が必要な文脈であり、要素が宣言を表す
3. 設定先は宣言が必要な文脈であり、要素が参照を表す

### 定義 3. 文脈依存変換が施行される場面

ここで、文脈とは設定する要素に求められた種類や性質のことで、個々のプロパティごとに設定されています。プロパティが持つ文脈を取得するには、プロパティの種類を表現する列挙

Navigation が持つメソッド *getContext* を利用します。これは列挙 `Navigation.Context` で定義される、対応する文脈を表現した定数を返します。

### 第3項 親子関係の調整

すでに親要素を持つDOM要素を他の要素の子要素として設定しようとした場合、その要素は複数の親要素を持つことになってしまい、Irenka DOMの親子関係が一貫性を保てなくなります。そのため、そのような場合は、設定しようとする要素を自動的に複製(第 1 項)し、その複製を子要素として代わりに設定することで回避します。

あるDOM要素の子要素として設定可能なDOM要素は、親要素を持たないか、複製によって親が存在しなくなったもののどちらかでなければなりません。これらの要素を設定する際、親子関係の一貫性を維持するために定義 4のような処理が実行されます。このように、必ず親と子の相互関係は一つの処理で設定され、親子関係の一貫性を保っています。

設定する要素を `elem`、設定先の親要素を `parent` とする。

また、設定先に既に存在する要素を `orig` とする。

1. `parent` の子要素 `orig` を、子要素として設定していない状態に変更する
2. `orig` の親要素を `null` に設定する
3. `orig` の親要素からの位置を `null` に設定する
4. `elem` の親要素を `parent` に変更する
5. `elem` の親要素からの位置を、対応するプロパティに適したものに設定する
6. `parent` の子要素として、`elem` を設定する

**定義 4. 要素を設定する際に実行される親子関係の調整処理**

## 第4節 DOM 要素の比較

`CtElement` が定義するメソッド *equals* を利用すると、DOM 要素の同値性比較を行うことができます。

### 第1項 要素の同値性

DOM要素の同値関係は、対象のDOM要素が宣言または参照(第 3 章)を表現するか、それ以外の要素であるかによって計算方法が異なります。宣言を表す(マスタ参照である)DOM要素については定義 5、参照を表す(マスタ参照である)DOM要素については定義 6、それ以外のDOM要素については定義 7で同値性を定義します。また、DOM要素の同値比較にはプロパティの同値比較が必要になる場合があります。これは第 2 項を参照してください

次のすべてを満たす場合のみ、宣言を表す要素 a と、比較対象の要素 b は同値である。

1. a と b は同一

#### 定義 5. 宣言を表す DOM 要素の同値関係

次のすべてを満たす場合のみ、参照を表す要素 a と、比較対象の要素 b は同値である。

1. a と b が持つ要素の種類は同一
2. b は宣言でない
3. a と b の宣言はそれぞれ同値
4. a と b のプロパティはそれぞれ同値

#### 定義 6. 参照を表す DOM 要素の同値関係

次のすべてを満たす場合のみ、宣言および参照でない要素 a と、比較対象の要素 b は同値である。

1. a と b が持つ要素の種類は同一
2. a と b のプロパティはそれぞれ同値

#### 定義 7. 定義および参照でない DOM 要素の同値関係

## 第2項 プロパティの同値性

DOM要素のプロパティ(第2節第1項)の同値関係は、プロパティの種類によって異なります。DOM要素を表すプロパティについては定義 8、そのリストを表すプロパティについては定義 9、列挙型を表すプロパティについては定義 10、それ以外のプロパティについては定義 11で同値性を定義します。

次のすべてを満たす場合のみ、要素 a と、比較対象の値 b は同値である。

1. b は DOM 要素である
2. a と b は同値である

#### 定義 8. DOM 要素を持つプロパティの同値関係

次のすべてを満たす場合のみ、要素のリスト  $a$  と、比較対象の値  $b$  は同値である。

1.  $b$  はリストである
2.  $a$  の要素数と  $b$  の要素数は等しい
3.  $a$  の要素数を  $n$ ,  $i=0..n-1$  とおくと、任意の  $i$  に対して  $a[i]$  と  $b[i]$  は同値である

#### 定義 9. DOM 要素のリストを持つプロパティの同値関係

次の要素をすべて満たす場合のみ、列挙型定数  $a$  と、比較対象の値  $b$  は同値である。

1.  $b$  は列挙型定数を表す
2.  $a$  を定義した列挙型を  $ta$  とおくと、 $ta$  は定数  $b$  を定義する
3. 列挙型定数  $a$  と列挙型定数  $b$  の単純名は同値である

#### 定義 10. 列挙型定数を持つプロパティの同値関係

次の要素をすべて満たす場合のみ、DOM 要素、DOM 要素のリスト、列挙型定数のいずれでもない  $a$  と、比較対象の値  $b$  は同値である。

1.  $a$  が null を表す場合
  - $b$  は null である
2.  $a$  が非 null の参照型である場合
  - $a.equals(b)$  が true と評価される
3.  $a$  が参照型でない場合
  - $a==b$  が true と評価される

#### 定義 11. それ以外のプロパティの同値関係

## 第3章 宣言と参照

Irenka では、宣言とその参照を表すそれぞれの DOM 要素は同一のインターフェースで宣言され、かつ同一の要素の種類で表現されます。このうち、宣言をマスタ参照、それに対する参照をスレーブ参照と呼びます。また、これらをまとめて参照と呼び、全ての参照を表す DOM 要素は CtReference 型のインターフェースのサブインターフェースとして宣言されます。

これら参照を表す DOM 要素は、表 5 に示されるインターフェースのうちいずれかを実装します。個々のインターフェースに対する解説はそれぞれの章を参照してください。それぞれのマスタ参照は何らかの宣言を表し、Irenka DOM 上に同じものは 1 つしか存在しません。それに対するスレーブ参照は対応するマスタ参照を利用するための要素で、複数存在することができます。以降、識別のため Xxx 型のマスタ参照を  $Xxx^{\text{Master}}$  と表記し、スレーブ参照を  $Xxx^{\text{Slave}}$  と表記します。たとえば、 $\text{CtClass}^{\text{Master}}$  と  $\text{CtClass}^{\text{Slave}}$  はどちらも CtClass インターフェースで表現されますが、前者はクラス型の宣言を表し、後者はクラス型を表すものとします。

表 5. CtReference のサブインターフェース

インターフェース	マスタ参照の意味	スレーブ参照の意味
CtPackage	Irenka DOM の名前空間	パッケージ宣言文
CtClass	クラス型の宣言	クラス型
CtInterface	インターフェース型の宣言	インターフェース型
CtEnum	列挙型の宣言	列挙型
CtAnnotation	注釈型の宣言	注釈型
CtField	フィールドの宣言	フィールド
CtEnumConstant	列挙定数の宣言	列挙定数
CtMethod	メソッドの宣言	メソッド
CtConstructor	コンストラクタの宣言	コンストラクタ
CtAnnotationElement	注釈要素の宣言	注釈要素
CtInitializer	初期化子の宣言	(なし <sup>2</sup> )
CtTypeParameter	型変数の宣言	型変数
CtParameter	仮引数の宣言	仮引数
CtLocalVariable	ローカル変数宣言文	ローカル変数
CtLabel	ラベル付き文のラベル宣言	ターゲットラベル

CtReference は表 6 にある機能を提供します。表の「提供」の列はその機能がマスタ参照で利用可能か、スレーブ参照で利用可能か、または共通して利用可能であることを表します。ただし、マスタ参照からスレーブ参照の機能を利用したり、スレーブ参照からマスタ参照の機能を利用したりすることも可能です。それらの解釈についてはそれぞれの説を参照してください。

<sup>2</sup> 初期化子はプログラム上から参照する方法がありません

表 6. CtReference が提供するメソッド

メソッド名	提供	概要
isMaster	共通	マスタ参照かどうか検査する
getMaster	スレーブ	対応するマスタ参照を返す
newReference	マスタ	新しいスレーブ参照を返す
getJavadoc	マスタ	宣言されたドキュメンテーションコメントを返す
getModifiersAndAnnotations	マスタ	宣言された修飾子と注釈の一覧を返す
getModifiers	マスタ	宣言された修飾子の一覧を返す
getAnnotations	マスタ	宣言された注釈の一覧を返す

マスタ参照(宣言)とスレーブ参照(宣言に対する参照)は 1 対多の関係にあります。つまり、1 つのマスタ参照に対して複数のスレーブ参照が作成されることがありますが、1 つのスレーブ参照が複数のマスタ参照を指すことはありません。マスタ参照からそれに対するスレーブ参照を作成するには、メソッド *newReference* を利用します。逆に、スレーブ参照から対応するマスタ参照を取得するには、メソッド *getMaster* を利用します。また、ある参照がマスタ参照であるかスレーブ参照であるか検査するには、メソッド *isMaster* を利用します。このメソッドが真を返す DOM 要素はマスタ参照を表し、そうでない DOM 要素はスレーブ参照を表します。

## 第1節 マスタ参照

マスタ参照はソースプログラム上に出現する「宣言」を表す要素で、1 つの Irenka DOM 上に同値であるものは 1 つしか存在しません。それぞれのマスタ参照は、対応する宣言に関するすべての情報をプロパティとして有します。

表 7はCtReference<sup>Master</sup>が提供する機能を表します。これは、表 6からスレーブ参照特有の機能を除去したものです。CtReference<sup>Master</sup>は一般的な宣言に付与させることができるJavaドキュメンテーションコメント(*getJavadoc*)、修飾子(*getModifiers*)、注釈(*getAnnotations*)をそれぞれ取得するためのメソッドが用意されています。これらについて詳しくは第 10 章を参照してください。

表 7. CtReference<sup>Master</sup>が提供するメソッド

メソッド名	概要
isMaster	常に真を返す
newReference	新しいスレーブ参照を返す
getJavadoc	宣言されたドキュメンテーションコメントを返す
getModifiersAndAnnotations	宣言された修飾子と注釈の一覧を返す
getModifiers	宣言された修飾子の一覧を返す
getAnnotations	宣言された注釈の一覧を返す

また、メソッド *newReference* はこのマスタ参照に対する新しいスレーブ参照を作成します。作成されたスレーブ参照に対するマスタ参照は、常にこのメソッドを呼び出した DOM 要素を指します。マスタ参照からスレーブ参照が提供するメソッドを呼び出そうとした場合、*newReference*

によってスレーブ参照を生成した後に、そのスレーブ参照を用いて対象のメソッドを呼び出します。つまり、*getMaster*を呼び出した場合は自分自身が返されます。

## 第2節 スレーブ参照

スレーブ参照はソースプログラム上に出現する「宣言」への参照を表す DOM 要素で、1つの宣言を表すマスタ参照に対して複数のスレーブ参照を生成することができます。また、それぞれのスレーブ参照は同値であることがあります。それぞれのスレーブ参照は、そのマスタへの参照と個々のスレーブ参照に付与された情報を保持します。

表 8はCtReference<sup>Slave</sup>が提供する機能を表します。これは、表 6からマスタ参照特有の機能を除去したものです。CtReference<sup>Slave</sup>はメソッド*getMaster*によって参照先のマスタ参照を取得することができます。スレーブ参照からマスタ参照特有の機能を呼び出した場合、*getMaster*によってマスタ参照を取得した後に、そのマスタ参照を用いて対象のメソッドを呼び出します。

表 8. CtReference<sup>Slave</sup>が提供するメソッド

メソッド名	概要
isMaster	常に偽を返す
getMaster	対応するマスタ参照を返す

スレーブ参照の多くは特有のプロパティを有さず、ほとんどの情報はマスタ参照が有するものを利用します。スレーブ参照に特有のプロパティには、適用された型変数(第 7 章)などがあります。このようなスレーブ参照特有のプロパティは同値性比較の際に考慮され、それぞれのプロパティも同値でなければなりません(第 2 章第 4 節)。

## 第4章 型

型(JLS3-4 Types, Values, and Variables)を表現するすべての DOM 要素は CtType インターフェイス、またはそのサブインターフェイスで表現されます。

型を表現するCtTypeはIrenka DOM上に出現するすべての型の親インターフェイスで、型に関する基本的な機能を提供します。表 9はCtTypeが提供する主な機能の一覧で、型が提供するメンバを取得するメソッド、検査を行うメソッド、変換を行うメソッドなどが提供されます。

表 9. CtType が提供するメソッド

メソッド名	概要
getTypeKind	型の種類(TypeKind)を返す
getName	完全限定名を返す
getSuperClass	親クラスを返す
getSuperInterfaces	親インターフェイスの一覧を返す
getField	公開するフィールドを返す
getMethod	公開するメソッドを返す
isAssignableFrom	代入可能かどうかの検査を行う
isCompatible	互換である型かどうかの検査を行う
isSame	同一の型かどうかの検査を行う
array	配列型を返す
box	ボックス変換を適用した型を返す
unbox	アンボックス変換を適用した型を返す
erasure	イレイジャ変換を適用した型を返す
apply	総称化コンテキスト(第 7 章第 2 節)を適用した型を返す

CtTypeが表現する型の種類を調べるには、メソッド*getTypeKind*を利用します。このメソッドは型の種類を列挙したTypeKind型の値を返し、表 10のように分類されます。実装によっては複数の分類に同時に該当する型が存在する場合がありますので、instanceof式による検査ではなく、TypeKindの比較による検査を行う必要があります。

表 10. TypeKind の種類

定数名	分類	対応する型の種類
INT	基本型(第 1 節)	int 型
LONG	基本型(第 1 節)	long 型
FLOAT	基本型(第 1 節)	float 型
DOUBLE	基本型(第 1 節)	double 型
BYTE	基本型(第 1 節)	byte 型
SHORT	基本型(第 1 節)	short 型
CHAR	基本型(第 1 節)	char 型
BOOLEAN	基本型(第 1 節)	boolean 型
VOID	基本型(第 1 節)	void 型

定数名	分類	対応する型の種類
CLASS	宣言型(第 2 節)	クラス型
INTERFACE	宣言型(第 2 節)	インターフェース型
ENUM	宣言型(第 2 節)	列挙型
ANNOTATION	宣言型(第 2 節)	注釈型
TYPE_PARAMETER	型変数(第 3 節)	型変数
WILDCARD	ワイルドカード(第 4 節)	ワイルドカード
ARRAY	配列型(第 5 節)	配列型
NULL	特殊型(第 6 節)	null 型
CAPTURED	特殊型(第 6 節)	捕捉変換によって生成された型
RANGE	特殊型(第 6 節)	型境界
INTERSECTION	特殊型(第 6 節)	共通型
EMPTY	特殊型(第 6 節)	存在しない型

なお、型の中には参照(第 3 章)の特性を有するインターフェースで表現されているものがあります。これらはいずれも型を表現する場合にスレーブ参照として Irenka DOM 上に出現します。

## 第1節 基本型

Irenka DOM では基本型(JLS3-4.2 Primitive Types and Values)に対して特別なインターフェースを用意しておらず、すべての基本型は `CtType` 型として表現されます。`CtType` 型で表現された基本型がどの型を表現しているか調べるには、`TypeKind` を利用する必要があります。

基本型の特徴として、親クラスを返すメソッド `getSuperClass` が `null` を返します。これは DOM 要素のプロパティとしては例外的で、基本型が `java.lang.Object` のサブクラスでないためにこのような扱いになっています。同様に基本型は実装インターフェースも存在しないため、メソッド `getSuperInterfaces` は空のリストを返します。

基本型のうちプリミティブ型に属するものは、ボックス変換を適用するメソッド `box` によって対応するラップ型へと変換することができます。また、メソッド `isCompatible` によって、プリミティブ型であるかそのラップ型であるかを意識せずに、同一の型であるかどうかを比較することができます。このように、プリミティブ型と対応するラップ型を同一の型とみなした上で、型が同一であることを互換の型と呼びます。

## 第2節 宣言型

宣言型(JLS3-4.3 Reference Types and Values)は `CtDeclaredTypeSlave` 型のインターフェースで表現され、これは `CtClassSlave` (クラス, 第 5 章第 1 節)、`CtInterfaceSlave` (インターフェース, 第 5 章第 2 節)、`CtEnumSlave` (列挙, 第 5 章第 3 節)、`CtAnnotationSlave` (注釈, 第 5 章第 4 節)それぞれの親インターフェースです。これらはいずれもスレーブ参照ですが、対応するマスタ参照はそれぞれの種類に対応する型の宣言を表現します。型の宣言については第 5 章を参照してください。

また、Irenka DOMではクラス型やインターフェース型のパラメータ化型(JLS3-4.5 Parameterized Types)もそのままクラス型、インターフェース型として取り扱います。すべての宣言型はジェネリック参照(第7章)として定義され、CtGenericReferenceインターフェースを実装しています。このインターフェースが公開するメソッド`getTypeArguments`を通じてパラメータ化型の型変数情報を取得することができます。また、メソッド`getParameterized`を利用して対応する宣言型に対するパラメータ化型を作成することもできます。

表 11はCtDeclaredType<sup>Slave</sup>が提供する主な機能です。CtDeclaredTypeはより多くの機能を提供していますが、そのほとんどはマスタ参照であるCtDeclaredType<sup>Master</sup>によって提供される機能です。CtDeclaredType<sup>Slave</sup>からCtDeclaredType<sup>Master</sup>のメソッドを呼び出した場合、それらはマスタ参照によって処理が行われます。CtDeclaredType<sup>Master</sup>のメソッドについては第5章で詳しく説明します。

表 11. CtDeclaredType<sup>Slave</sup>が提供するメソッド

メソッド名	概要
<code>getField</code>	フィールドを返す
<code>getMethod</code>	メソッドを返す
<code>getConstructor</code>	コンストラクタを返す
<code>getMemberType</code>	メンバ型を返す
<code>raw</code>	未加工型を返す
<code>getTypeArguments</code>	型変数に適用された型の一覧を返す

## 第3節 型変数

型変数(JLS3-4.4 Type Variables)はCtTypeParameter<sup>Slave</sup>型のインターフェースで表現されます。これは、型変数の宣言を表現するCtTypeParameter<sup>Master</sup>のスレーブ参照で、宣言された型変数に対応した型変数として利用することができます。

CtTypeParameter<sup>Slave</sup>は特別な機能を提供ないため、そのマスタ参照を経由して情報を取得します。なお、CtTypeParameter<sup>Slave</sup>からCtTypeParameter<sup>Master</sup>が提供するメソッドを呼び出すと、自動的にマスタ参照の情報が返されます。CtTypeParameter<sup>Master</sup>のメソッドについては第7章第1節で詳しく説明します。

## 第4節 ワイルドカード

ワイルドカード(JLS3-4.5.1 Type Arguments and Wildcards)はJava言語仕様では型として取り扱いませんが、Irenka DOMでは型として取り扱い<sup>3</sup>CtWildcard型のインターフェースで表現されます。

<sup>3</sup> 型そのものではなく、型の範囲を表現します。Irenkaではあえて表現として区別せずに、内部の処理でのみ区別しています。

表 12はCtWildcardが提供する主な機能です。境界の種類はWildcardBoundKindという列挙定数で表現され、その一覧が表 13にあります。CtWildcardに境界の指定がない(WildcardBoundKind.UNBOUNDEDである)場合、メソッド`getBound`は`java.lang.Object`型を表現するクラス型を返します。

表 12. CtWildcard が提供するメソッド

メソッド名	概要
<code>getBoundKind</code>	境界の種類(WildcardBoundKind)を返す
<code>getBound</code>	境界型を返す

表 13. WildcardBoundKind の種類

定数の名前	概要
UNBOUNDED	境界は存在しない(?)
UPPER_BOUNDED	上限境界が存在する(? extends X)
LOWER_BOUNDED	下限境界が存在する(? super X)

## 第5節 配列型

配列型(JLS3-10.1 Array Types)はCtArray型のインターフェースで表現されます。これは、表 14のようなメソッドを提供します。

表 14. CtArray の提供するメソッド

メソッド名	概要
<code>getComponentType</code>	要素型を返す
<code>getScalarType</code>	0次元の要素型を返す
<code>newArray</code>	この型に対応する配列生成式(第9章第6節)を生成する

## 第6節 特殊型

特殊型はソースプログラム上に表記できない型で、式の評価結果や型推論の途中などに出現する場合があります。これらは DOM 要素として表現されながら、Irenka DOM 上には出現しません。そのため、親子関係が適切でない要素が出現する場合があります。

### 第1項 null 型

null 型(JLS3-4 Types, Values, and Variables)は null リテラルの評価結果として利用される型で、TypeKind.NULL という種類を有します。この型は全ての参照型のサブタイプ(JLS3-4.10 Subtyping)であるという特徴を有し、比較には注意が必要です。

## 第2項 捕捉型

捕捉型<sup>4</sup>は捕捉変換(JLS3-5.1.10 Capture Conversion)の結果として出現する型で、`TypeKind.NULL`という種類を有します。これは、ワイルドカードを含むパラメータ化型を持つ式を評価した際などに取り扱われます。

## 第3項 型境界

型境界は上限境界と下限境界を同時に指定可能な型の範囲で、`TypeKind.RANGE`という種類を有します。この型は`CtBoundedType`というインターフェースを有し、メソッド `getUpperBound`、および `getLowerBound` を利用して上限境界および下限境界を取得することができます。なお、`CtBoundedType`は`CtWildcard`(第4節)や`CtTypeParameter`(第3節第7章第1節)の親インターフェースとしても出現します。この場合、上限境界または下限境界のいずれか1つは`java.lang.Object`型または`null`型に固定されています。

型境界は捕捉変換の処理やメソッド起動式(第9章第9節)などの型推論中に出現する可能性があります。

## 第4項 共通型

共通型(JLS3-4.9 Intersection Types)はスーパータイプのみが判明している無名の型で、`TypeKind.INTERSECTION`という種類を有します。この型は `CtIntersectionType` というインターフェースを有し、メソッド `getTypes` を利用してスーパータイプの一覧を取得することができます。

共通型は捕捉変換の処理やメソッド起動式(第9章第9節)などの型推論中に出現する可能性があります。また、`CtBoundedType`が提供するメソッドの中には共通型を返すものがあります。

## 第5項 存在しない型

存在しない型は名前の通り存在しない型を表し、Irenka に内蔵されたコンパイラがエラー処理などに利用します。この型は `TypeKind.EMPTY` という種類で表わされます。

---

<sup>4</sup> 未実装

## 第5章 型の宣言

型の宣言はCtDeclaredType<sup>Master</sup>型のインターフェースで表現されます。これは CtClass<sup>Master</sup> (クラス, 第1節)、CtInterface<sup>Master</sup> (インターフェース, 第2節)、CtEnum<sup>Master</sup> (列挙, 第3節)、CtAnnotation<sup>Master</sup> (注釈, 第4節)それぞれの親インターフェースです。

CtDeclaredType<sup>Master</sup>は、さまざまな型宣言が持つ基本的な機能を提供します(表 15)。提供されるメソッドのうち、*getDeclared~*から始まるメソッドは全ての宣言型が提供する機能ではありませんが、提供しない場合は単純に発見できなかった場合と同じ動作をします。

表 15. CtDeclaredType<sup>Master</sup>が提供するメソッド

メソッド名	概要
getDeclaringPackage	この型を宣言したパッケージを返す(トップレベル型のみ)
getSimpleName	単純名を返す
getTypeParameters	宣言された型変数の一覧を返す
getSuperClass	親クラスを返す
getSuperInterfaces	親インターフェースの一覧を返す
getMembers	宣言されたメンバの一覧を返す
getDeclaredField	宣言されたフィールドを返す
getDeclaredConstructor	宣言されたコンストラクタを返す
getDeclaredMethod	宣言されたメソッドを返す
getDeclaredMemberType	宣言されたメンバ型を返す
getDeclaredLocation	この型が宣言されたスコープ(DeclaredScopeKind)を返す
getParameterized	この型のパラメータ化型を返す

パッケージの宣言については第2章第2節、型変数の宣言については第7章第1節、メンバの宣言については第6章、宣言されたスコープについては表 16をそれぞれ参照してください。

表 16. DeclaredScopeKind の種類

定数の名前	概要
TOP_LEVEL	トップレベル型として宣言されたことを表す
MEMBER	メンバ型として宣言されたことを表す
LOCAL	ローカルクラスとして宣言されたことを表す
ANONYMOUS	匿名クラスとして宣言されたことを表す

### 第1節 クラス

クラスの宣言(JLS3-8.1 Class Declaration)はCtClass<sup>Master</sup>型のインターフェースで表現され、パッケージの宣言を表すCtPackage<sup>Master</sup>(第2章第2節)の子要素、メンバ型の宣言(第6章第7節)、ローカルクラスの宣言(第8章第4節)、匿名クラスの宣言(第9章第5節)に利用することができます。

CtClass は CtDeclaredType のサブインターフェイスとして宣言されており、CtDeclaredType が提供するすべての機能を利用することができます。

## 第2節 インターフェイス

インターフェイスの宣言(JLS3-9.1 Interface Declarations)はCtInterface<sup>Master</sup>型のインターフェイスで表現され、パッケージの宣言を表すCtPackage<sup>Master</sup>(第2章第2節)の子要素、メンバ型の宣言(第6章第7節)に利用することができます。

CtInterfaceはCtDeclaredTypeのサブインターフェイスとして宣言されていますが、いくつかの機能が制限されています。まず、インターフェイスは明示的な親クラスを指定できないため、メソッド *getSuperClass* は常に `java.lang.Object` 型を表す要素 (CtClass<sup>Slave</sup> <Object>) が返されます。また、コンストラクタを宣言することが不可能であるため、メソッド *getConstructor* や *getDeclaredConstructor* はそれぞれコンストラクタの検出に常に失敗します。

## 第3節 列挙

列挙の宣言(JLS3-8.9 Enums)はCtEnum<sup>Master</sup>型のインターフェイスで表現され、パッケージの宣言を表すCtPackage<sup>Master</sup>(第2章第2節)の子要素、メンバ型の宣言(第6章第7節)に利用することができます。

CtEnumは列挙特有の機能として、宣言された列挙定数を取得するメソッド *getDeclaredConstant* を提供します。そして、CtEnumはCtClassのサブインターフェイスとして定義されていますが、いくつかの機能は制限されています(表 17)。まず、CtEnumは総称化することができないため、総称化に関するメソッド(第7章)は全て利用できません。また、親クラスを返すメソッド *getSuperClass* は常に `java.lang.Enum<?>` 型を表す要素 (CtClass<sup>Slave</sup> <Enum<?>>) が返されます<sup>5</sup>。

表 17. CtEnum<sup>Master</sup> が提供するメソッド

メソッド名	概要
<i>getDeclaringPackage</i>	この型を宣言したパッケージを返す(トップレベル型のみ)
<i>getSimpleName</i>	単純名を返す
<i>getSuperClass</i>	親クラスを返す
<i>getSuperInterfaces</i>	親インターフェイスの一覧を返す
<i>getMembers</i>	宣言されたメンバの一覧を返す
<i>getDeclaredConstant</i>	宣言された列挙定数を返す
<i>getDeclaredField</i>	宣言されたフィールドを返す
<i>getDeclaredMethod</i>	宣言されたメソッドを返す
<i>getDeclaredMemberType</i>	宣言されたメンバ型を返す

<sup>5</sup> 本来は、列挙型Xに対してその親クラスは `java.lang.Enum<X>` 型であるべきですが、型の整合性の問題で `Enum<?>` としています。

メソッド名	概要
getDeclaredLocation	この型が宣言されたスコープ(DeclaredScopeKind)を返す

## 第4節 注釈

注釈の宣言(JLS3-9.6 Annotation Types)はCtAnnotation<sup>Master</sup>型のインターフェースで表現され、パッケージの宣言を表すCtPackage<sup>Master</sup>(第2章第2節)の子要素、メンバ型の宣言(第6章第7節)に利用することができます。

CtAnnotationは注釈特有の機能として、宣言された注釈要素を取得するメソッド *getDeclaredElement*を提供します。そして、CtAnnotationはCtInterfaceのサブインターフェースとして定義されていますが、いくつかの機能は制限されています(表 18)。まず、CtAnnotationは総称化することができないため、総称化に関するメソッド(第7章)は全て利用できません。また、親インターフェースを返すメソッド *getSuperInterfaces*は常に要素が `java.lang.annotation.Annotation`型を表す要素(CtInterface<sup>Slave</sup><Annotation>)のみからなるリストを返します。

表 18. CtAnnotation<sup>Master</sup>が提供するメソッド

メソッド名	概要
getDeclaringPackage	この型を宣言したパッケージを返す(トップレベル型のみ)
getSimpleName	単純名を返す
getSuperClass	親クラスを返す
getSuperInterfaces	親インターフェースの一覧を返す
getMembers	宣言されたメンバの一覧を返す
getDeclaredElement	宣言された要素を返す
getDeclaredField	宣言されたフィールドを返す
getDeclaredConstructor	宣言されたコンストラクタを返す
getDeclaredMethod	宣言されたメソッドを返す
getDeclaredMemberType	宣言されたメンバ型を返す
getDeclaredLocation	この型が宣言されたスコープ(DeclaredScopeKind)を返す

## 第6章 メンバ

メンバ(JLS3-6.4.3 The Members of a Class Type, JLS3-6.4.4 The Members of an Interface Type)を表現するすべてのDOM要素は、CtMemberのサブインターフェースで表現されます。なお、Irenkaでは、コンストラクタ(第4節)を表すCtConstructorや、初期化子(第6節)を表すCtInitializerなども利便性を考慮してCtMemberのサブタイプとしています。ここでは、これらも単純にメンバと呼ぶことにします。

CtMemberは参照の特徴を持つDOM要素で、それぞれマスタ参照とスレーブ参照があります。CtMember<sup>Master</sup>はメンバの宣言を表現し、CtMember<sup>Slave</sup>はメンバへの参照を表現します。各メンバの意味は表19の通りです。マスタ参照とスレーブ参照で意味が異なることに注意してください。

表 19. CtMember のサブインターフェース

インターフェース	マスタ参照	スレーブ参照
CtField	フィールドの宣言	フィールド
CtEnumConstant	列挙定数の宣言	列挙定数
CtMethod	メソッドの宣言	メソッド
CtConstructor	コンストラクタの宣言	コンストラクタ
CtAnnotationElement	注釈要素の宣言	注釈要素
CtDeclaredType	メンバ型の宣言	宣言型
CtInitializer	初期化子の宣言	(なし)

また、メンバのスレーブ参照を利用する場合、多くは他のDOM要素の一部として利用されます。表20はメンバのスレーブ参照を利用して構成されるDOM要素の一覧です。ただし、CtDeclaredType<sup>Slave</sup>は通常の宣言型(第4章第2節)として利用可能であるため様々な個所から利用されます。また、CtInitializerはスレーブ参照が存在しないため、CtInitializer<sup>Slave</sup>が利用されることはありません。

表 20. メンバの参照方法

インターフェース	スレーブ参照を利用する DOM 要素
CtField <sup>Slave</sup>	CtVariableAccess (第9章第7節)
CtEnumConstant <sup>Slave</sup>	CtVariableAccess (第9章第7節)
CtMethod <sup>Slave</sup>	CtMethodInvocation (第9章第9節)
CtConstructor <sup>Slave</sup>	CtNewInstance (第9章第5節)
CtAnnotationElement <sup>Slave</sup>	CtAnnotationInstanceElement (第10章第1節)
CtDeclaredType <sup>Slave</sup>	(任意の宣言型を利用可能な要素)
CtInitializer <sup>Slave</sup>	(なし)

## 第1節 フィールド

フィールドの宣言(JLS3-8.3 Field Declarations)はCtField<sup>Master</sup>型のインターフェースで表わされ、任意の宣言型のメンバとすることができます。

CtFieldは変数を表すCtVariableのサブインターフェースとして定義されており、いくつかの機能はCtVariableで宣言されています。表 21はCtField<sup>Master</sup>が提供する機能を表し、表 22はCtField<sup>Slave</sup>が提供する機能を表します。なお、配列型のフィールドを初期化する際にJava言語仕様では配列初期化子が利用可能ですが、これをIrenkaでは配列生成式(第9章第6節)を利用して表現します。

表 21. CtField<sup>Master</sup>が提供するメソッド

メソッド名	概要
getType	宣言されたフィールドの型を返す
getSimpleName	単純名を返す
getInitializer	初期化式を返す

表 22. CtField<sup>Slave</sup>が提供するメソッド

メソッド名	概要
access	このフィールドを参照する CtVariableAccess を返す
accessFrom	指定のオブジェクトを利用してフィールドを参照する CtVariableAccess を返す

## 第2節 列挙定数

列挙定数の宣言(JLS3-8.9 Enums)はCtEnumConstant<sup>Master</sup>型のインターフェースで表わされ、列挙型のメンバとすることができます。

CtEnumConstantはCtFieldのサブインターフェースとして宣言されており、すべての列挙定数はフィールドとして取り扱うこともできます。ただし、CtEnumConstant<sup>Master</sup>はCtField<sup>Master</sup>に比べ、初期化式にとれる式の種類が大きく制限されます。列挙定数にはそれを宣言する列挙型のインスタンスが格納されるため、その初期化式は必ず存在し、Irenkaではクラス・インスタンス生成式(第9章第5節)として表現されます。引数や匿名ブロックを伴う列挙定数の宣言は、それぞれ初期化式のクラス・インスタンス生成式の実引数と匿名ブロックの宣言として表現されます。

## 第3節 メソッド

メソッドの宣言(JLS3-8.4 Method Declarations)はCtMethod<sup>Master</sup>型のインターフェースで表わされ、注釈型を除く任意の型宣言のメンバとすることができます。

CtMethodは起動可能なメンバを表すCtInvocableのサブインターフェースとして宣言されており、主要な機能はCtInvocableで宣言されています。表 23はCtMethod<sup>Master</sup>が提供する機能を表し、表 24はCtMethod<sup>Slave</sup>が提供する機能を表します。

表 23. CtMethod<sup>Master</sup>が提供するメソッド

メソッド名	概要
getTypeParameters	宣言された型変数の一覧を返す
getReturnType	宣言された戻り値型を返す
getSimpleName	単純名を返す
getParameters	宣言されたメソッド引数の一覧を返す
getThrows	宣言された例外の一覧を返す
getBody	メソッド本体を返す
getParameterized	このメソッドに型引数を指定したパラメータ化メソッドを返す

表 24. CtMethod<sup>Slave</sup>が提供するメソッド

メソッド名	概要
getTypeArguments	型変数に適用された型の一覧を返す
newInvocation	このメソッドを起動する CtMethodInvocation を返す

CtMethod<sup>Master</sup>が持つメソッド`getParameters`は、メソッド引数の一覧をCtParameter<sup>Master</sup>型のリストで返します。CtParameter<sup>Master</sup>はメソッド引数やコンストラクタ引数の宣言を表現します。CtParameter<sup>Master</sup>はCtField<sup>Master</sup>と似た機能を有しますが(第 1 節)、CtMemberとして取り扱うことができません。

表 25. CtParameter<sup>Master</sup>が提供するメソッド

メソッド名	概要
getType	宣言された型を返す
getSimpleName	単純名を返す
access	この引数を参照する CtVariableAccess を返す

## 第4節 コンストラクタ

コンストラクタの宣言(JLS3-8.8 Constructor Declarations)はCtConstructor<sup>Master</sup>型のインターフェースで表わされ、クラス型または列挙型のメンバとすることができます。

CtConstructorはCtMethodと同様にCtInvocableのサブインターフェースとして宣言されており、主要な機能はCtInvocableで宣言されています。表 26はCtConstructor<sup>Master</sup>が提供する機能を表し、表 27はCtConstructor<sup>Slave</sup>が提供する機能を表します。なお、コンストラクタの型変数はコンストラクタ宣言に付与された型変数宣言に対する適用であって、クラス宣言に付与された型変数宣言に対する適用と異なることに注意してください。つまり、HogeクラスのコンストラクタにXという型変数を与えると、"new Hoge<X>()"ではなく、"new<X> Hoge()"となります。また、コンストラクタ引数はメソッド引数(第 3 節)と同様にCtParameter<sup>Master</sup>型のインターフェースで表現されます。

表 26. CtConstructor<sup>Master</sup>が提供するメソッド

メソッド名	概要
getTypeParameters	宣言された型変数の一覧を返す
getParameters	宣言されたコンストラクタ引数の一覧を返す
getThrows	宣言された例外の一覧を返す
getBody	メソッド本体を返す
getParameterized	このコンストラクタに型引数を指定したパラメータ化コンストラクタを返す

表 27. CtConstructor<sup>Slave</sup>が提供するメソッド

メソッド名	概要
getTypeArguments	型変数に適用された型の一覧を返す
newInvocation	このコンストラクタを利用する CtNewInstance を返す

## 第5節 注釈要素

注釈要素の宣言(JLS3-9.6 Annotation Types)はCtAnnotationElement<sup>Master</sup>型のインターフェイスで表わされ、注釈型のメンバとすることができます。

すべての注釈要素は通常のインターフェイスメソッドとしても取り扱うことができるため、CtAnnotationElementはCtMethodのサブインターフェイスとして宣言されています。表 28はCtAnnotationElement<sup>Master</sup>が提供する機能を表し、表 29はCtAnnotationElement<sup>Slave</sup>が提供する機能を表します。なお、配列型の要素に対する規定値にJava言語仕様では配列初期化子が利用可能ですが、これをIrenkaでは配列生成式(第9章第6節)を利用して表現します。

表 28. CtAnnotationElement<sup>Master</sup>が提供するメソッド

メソッド名	概要
getReturnType	宣言された戻り値型を返す
getSimpleName	単純名を返す
getDefault	規定値を返す

表 29. CtAnnotationElement<sup>Slave</sup>が提供するメソッド

メソッド名	概要
newInvocation	このメソッドを起動する CtMethodInvocation を返す
newInstance	この注釈要素を参照する CtAnnotationInstanceElement を返す

## 第6節 初期化子

初期化子の宣言(JLS3- 8.6 Instance Initializers, JLS3-8.7 Static Initializers)はCtInitializer<sup>Master</sup>型のインターフェイスで表わされ、クラス型または注釈型のメンバとすることができます。

きます。CtInitializer<sup>Master</sup>の表現する要素がクラス初期化子またはインスタンス初期化子のいずれであるかは、その修飾子にstaticが指定されているかどうかで決まります。staticが指定されていればクラス初期化子を表現し、そうでなければインスタンス初期化子として取り扱われます。

初期化子は名前を持たないため、他のDOM要素から参照することはできません。そのため、CtInitializerのスレーブ参照は存在せず、宣言を表すマスタ参照のみがIrenka DOM上に出現します。表 30はCtInitializer<sup>Master</sup>が提供する機能を表します。

**表 30. CtInitializer<sup>Master</sup>が提供するメソッド**

メソッド名	概要
getBody	初期化子の本体を返す

## 第7節 メンバ型

メンバ型の宣言(JLS3-8.5 Member Type Declarations)は、通常の宣言型と同様にCtDeclaredType<sup>Master</sup>のサブインターフェースで表わされます。CtDeclaredTypeが提供する機能については、第5章(CtDeclaredType<sup>Master</sup>)および第4章第2節(CtDeclaredType<sup>Slave</sup>)を参照してください。

## 第7章 ジェネリック参照

ジェネリック参照とは型変数を取り扱うことのできる参照(第3章)のことで、CtGenericReference型のインターフェースのサブインターフェースとして宣言されます。ジェネリック参照は、それぞれCtClass(第5章第1節)、CtInterface(第5章第2節)、CtMethod(第6章第3節)、CtConstructor(第6章第4節)型で表現されたDOM要素が該当します。また、ジェネリック参照を表すCtGenericReferenceは、参照を表すCtReference型のサブインターフェースとして宣言されており、マスタ参照とスレーブ参照でそれぞれ異なる機能を有します。

ジェネリック参照のマスタ参照を表すCtGenericReference<sup>Master</sup>は、型変数の宣言が可能な要素を表現します。クラス、インターフェース、メソッド、コンストラクタはいずれも型変数の宣言が可能な要素で、それぞれジェネリック・クラス宣言、ジェネリック・インターフェース宣言、ジェネリック・メソッド宣言、ジェネリック・コンストラクタ宣言と呼ばれます。型変数の宣言については第1節を参照してください。表31はこのCtGenericReference<sup>Master</sup>が提供する機能を表します。

表 31. CtGenericReference<sup>Master</sup>が提供するメソッド

メソッド名	概要
getTypeParameters	宣言された型変数の一覧を返す
getParameterized	パラメータ化されたスレーブ参照を返す

ジェネリック参照のマスタ参照を表すCtGenericReference<sup>Slave</sup>は、パラメータ化可能な要素を表現します。クラス、インターフェース、メソッド、コンストラクタはいずれもパラメータ化が可能で、それぞれパラメータ化クラス、パラメータ化インターフェース、パラメータ化メソッド、パラメータ化コンストラクタと呼ばれます。表32はこのCtGenericReference<sup>Slave</sup>が提供する機能を提供します。

表 32. CtGenericReference<sup>Slave</sup>が提供するメソッド

メソッド名	概要
getTypeArguments	型変数に適用された型の一覧を返す

### 第1節 型変数の宣言

型変数(JLS3-4.4 Type Variables)とは型のプレースホルダのようなもので、ジェネリック宣言と同時に宣言されます。型変数にはジェネリック宣言をパラメータ化する際に実際の型が与えられ、ジェネリック宣言内で使用された全ての型変数はパラメータとして与えられた型と同様に扱うことができます。この型変数の宣言は、CtTypeParameter<sup>Master</sup>型のインターフェースで表わされます。

CtTypeParameter<sup>Master</sup>は表33にある機能を提供します。型変数は単純名を持ち、上限境界となる型を指定することができます。

表 33. CtTypeParameter<sup>Master</sup>が提供するメソッド

メソッド名	概要
-------	----

メソッド名	概要
getSimpleName	単純名を返す
getBounds	宣言された境界型の一覧を返す

CtTypeParameterMasterに対するスレーブ参照については、第 4 章第 3 節を参照してください。

## 第2節 総称化コンテキスト

総称化コンテキストは、パラメータ化された要素の各型変数に適用されたそれぞれの型に関する情報を表します。これは、GenericContext 型のインターフェースで表現され、パラメータ化された要素がもつ型パラメータに関する情報のみを分離して取り扱うことができます。

GenericContextは表 34にある機能を提供します。このうち、メソッド*erase*, *ignore*, *strict* はそれぞれGenericContextStrategyオブジェクトを返します。これは型を表現するCtType (第 4 章)が持つメソッド*apply*の引数にとることができ、対象の型に含まれる型変数を、現在の総称化コンテキストを利用して実際の型に置換することができます。

表 34. GenericContext が提供するメソッド

メソッド名	概要
getOwner	このコンテキストの所有者を返す
getTypeParameter	指定の名前を持つ型変数を返す
getTypeArgument	指定の型変数に適用された型を返す
erase	未指定時に型変数のイレイジャ変換を利用する戦略を返す
ignore	未指定時に型変数をそのまま返す戦略を返す
strict	未指定時にエラーを発生させる戦略を返す

この総称化コンテキストは、CtElement (第 2 章)が提供するメソッド*getGenericContext*を利用して取得します。対象のDOM要素がジェネリック参照であれば、このメソッドはそのままジェネリック参照に対する型引数の情報を返します。そうでない場合、親要素の中から適切なジェネリック参照を探し出し、その情報を返します。

## 第8章 文

文(JLS3-14 Blocks and Statements)を表現するすべての DOM 要素は、CtStatement のサブインターフェースで表現されます。文とはプログラムの実行順序を制御するための言語要素で、CtStatement のサブインターフェースは主に制御構造を表す DOM 要素を提供します。Java 言語仕様では文とブロックに関する言語要素を分けて取り扱っていますが、Irenka ではブロックに関する言語要素も CtStatement インターフェースを有し、単純に文として取り扱います。

CtStatementそのものはほとんど機能を持たず、表 35にある機能のみを提供します。ほとんどの機能はそれぞれのDOM要素に対応するCtStatementのサブインターフェースが機能を提供しています。また、CtStatementに出現するラベルについては第 2 節を、ブロックについては第 3 節を参照してください。それぞれのインターフェースについては、第 3 節以降を参照してください。

表 35. CtStatement が提供するメソッド

メソッド名	概要
getLabels	付与されたラベルの一覧を返す
substituteAsBlock	この文のみからなるブロックに置き換える

### 第1節 式文

式文(JLS3-14.8 Expression Statements)とは式をそのまま文として利用する文法で、CtExpressionStatementというインターフェースで表わされます。このインターフェースは文を表す CtStatementと式(第 9 章)を表すCtExpressionの両方を親インターフェースに有し、文としても式としても取り扱うことができます。

以下はCtExpressionStatementを親インターフェースに持つDOM要素の一覧です。それぞれの詳細については、第 9 章の各節を参照してください。

- CtNewInstance (第 9 章第 5 節)
- CtMethodInvocation (第 9 章第 9 節)
- CtUnary (第 9 章第 11 節)
- CtAssignment (第 9 章第 16 節)

### 第2節 ラベル付き文

ラベル付き文(JLS3-14.7 Labeled Statements)とは、名前の通りラベルが付与された文のことで、ラベルはCtLabel<sup>Master</sup>型のインターフェースで表現されます。ただし、ローカルクラス宣言(第 4 節)とローカル変数宣言(第 5 節)にはどちらもラベルを付与することができません。付与されたラベルはCtLabel<sup>Master</sup>型のインターフェースでラベルの宣言を表し、CtBreak (第 14 節)や

CtContinue (第 15 節)はそのラベルの参照を表すCtLabel<sup>Slave</sup>を利用することができます。文に付与されたラベルを取得するには、CtStatementが提供するメソッド*getLabels*を利用します。

CtLabel<sup>Master</sup>は表 36にある機能を提供します。

表 36. CtLabel<sup>Master</sup>が提供するメソッド

メソッド名	概要
getSimpleName	名前を返す

## 第3節 ブロック

ブロック(JLS3-14.2 Blocks)は CtBlock 型のインターフェースで表現され、通常の文やメソッドなどの本体として利用されます。

CtBlockは表 37にある機能を提供します。ブロックは複数の文を子要素に有し、それらを取得するためのメソッド*getStatements*に加え、それらを参照操作するためのいくつかのメソッドが提供されます。

表 37. CtBlock が提供するメソッド

メソッド名	概要
getStatements	含まれる文の一覧を返す
contains	指定した文が含まれているかどうか検査する
getFirst	ブロックの先頭の文を返す
getLast	ブロックの末尾の文を返す
insertBefore	指定した文の直前に文を挿入する
insertAfter	指定した文の直後に文を挿入する
remove	指定した文を削除する

## 第4節 ローカルクラス宣言

ローカルクラスの宣言<sup>6</sup>(JLS3-14.3 Local Class Declarations)はCtClass<sup>Master</sup>型のインターフェースとして宣言され、ブロックに含まれる文として利用することが可能です。

CtClass<sup>Master</sup>の機能については第 5 章第 1 節にその説明があります。ローカルクラスとして宣言されたCtClass<sup>Master</sup>は、スコープとしてDeclaredScopeKind.LOCALを有します。

<sup>6</sup> 未実装

## 第5節 ローカル変数宣言文

ローカル変数宣言文(JLS3-14.4 Local Variable Declaration Statements)は `CtLocalVariableMaster` 型のインターフェースとして宣言され、ブロックに含まれる文として利用することが可能です。

`CtLocalVariableMaster` は変数を表す `CtVariable` のサブインターフェースとして定義されており、表 38にあるような機能を提供します。なお、配列型のローカル変数を初期化する際にJava言語仕様では配列初期化子が利用可能ですが、これをIrenkaでは配列生成式(第9章第6節)を利用して表現します。

表 38. `CtLocalVariableMaster` が提供するメソッド

メソッド名	概要
<code>getType</code>	宣言された型を返す
<code>getSimpleName</code>	単純名を返す
<code>getInitializer</code>	初期化式を返す
<code>access</code>	この変数を参照する <code>CtVariableAccess</code> を返す

なお、宣言されたローカル変数を使用するには、変数アクセス式(第9章第7節)を使用します。このとき、`CtVariableAccess`の参照先には宣言されたローカル変数への参照を表す `CtLocalVariableSlave` が使用され、そのマスタ参照は変数の宣言を表す `CtLocalVariableMaster` です。

## 第6節 空文

空文(JLS3-14.6 The Empty Statement)は";"のみからなる何も行わない文のことで、主にループ本体を持たないループ構造などに利用されます。これは、`CtEmptyStatement` 型のインターフェースとして宣言され、通常の文として利用することが可能です。

空文は特に機能を有しません。

## 第7節 if 文

if 文(JLS3- 14.9 The if Statement)は `CtIf` 型のインターフェースとして宣言され、通常の文として利用することが可能です。

`CtIf`は表 39にある機能を提供します。この要素がif-then文を表す場合、メソッド`getElse`は存在しないDOM要素を返します。

表 39. `CtIf` が提供するメソッド

メソッド名	概要
<code>getCondition</code>	条件式を返す

メソッド名	概要
getThen	条件成立時に実行される文を返す
getElse	条件不成立時に実行される文を返す

## 第8節 assert 文

assert 文(JLS3-14.10 The assert Statement)は CtAssert 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtAssertは表 40にある機能を提供します。表明失敗時のメッセージは省略可能で、省略された場合はメソッド*getMessage*が存在しないDOM要素を返します。

表 40. CtAssert が提供するメソッド

メソッド名	概要
getAssertion	表明する条件式を返す
getMessage	表明失敗時のメッセージを表す式を返す

## 第9節 switch 文

switch 文(JLS3-14.11 The switch Statement)は CtSwitch 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtSwitchは表 41にある機能を提供します。switch文は分岐に利用する値を表すセレクタ(*getSelector*)と、switchブロック(*getBody*)から構成されます。switch文の本体はCtCase型のリストで表現され、それぞれのCtCaseは一つ分のcaseまたはdefaultラベルと、それに続く文のリストから構成されます。

表 41. CtSwitch が提供するメソッド

メソッド名	概要
getSelector	セレクタ式を返す
getBody	switch 文の本体を返す

switch文の本体を構成するCtCaseは、表 42にある機能を提供します。CtCaseはcaseラベルかdefaultラベルのどちらかを有し、メソッド*isDefault*によって判定することができます。CtCaseがcaseラベルを有する場合、メソッド*getExpression*はcaseラベル中の定数式を返します。そうでなく、defaultラベルを有する場合、メソッド*getExpression*は存在しないDOM要素を返します。また、メソッド*getStatements*は現在のラベルから次のラベルまで、またはswitchブロックの終端までに存在するすべての文を返します。

表 42. CtCase が提供するメソッド

メソッド名	概要
getExpression	case ラベル中の定数式を返す

メソッド名	概要
getStatements	この case に後続する式の一覧を返す
isDefault	default を表現するラベルかどうか検査する

## 第10節 while 文

while 文(JLS3-14.12 The while Statement)は CtWhile 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtWhileは表 43にある機能を提供します。

表 43. CtWhile が提供するメソッド

メソッド名	概要
getCondition	条件式を返す
getBody	ループ本体を返す

## 第11節 do 文

do 文(JLS3-14.13 The do Statement)は CtDo 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtDoは表 44にある機能を提供します。

表 44. CtDo が提供するメソッド

メソッド名	概要
getBody	ループ本体を返す
getCondition	条件式を返す

## 第12節 for 文

for文(JLS3-14.14.1 The basic for Statement)はCtFor型のインターフェースとして宣言され、通常の文として利用することが可能です。CtForはJava言語仕様で基本for文と呼ばれ、Java言語仕様第3版で新しく追加されたfor文を拡張for文(第13節)と呼びます。

CtForは表 45にある機能を提供します。このうち、初期化コード(*getInitializers*)は1つ以上のローカル変数宣言文(第5節)のみからなるリスト、1つ以上の式文(第1節)からなるリスト、または空のリストで表現されます。ただし、2つ以上のローカル変数宣言文を利用する場合は、それぞれの型が同一である必要があります。また、条件式(*getCondition*)は省略が可能で、省略された場合はtrueを表現する存在しないDOM要素が返されます。更新コード(*getUpdaters*)は1つ以上の式文(第1節)からなるリスト、または空のリストで表現されます。

表 45. CtFor が提供するメソッド

メソッド名	概要
getInitializers	初期化コードを返す
getCondition	条件式を返す
getUpdaters	更新コードを返す
getBody	ループ本体を返す

## 第13節 拡張 for 文

拡張 for 文(JLS3-14.14.2 The enhanced for statement)は CtForEach 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtForEachは表 46にある機能を提供します。形式的引数はメソッド引数を表す CtParameter<sup>Master</sup>ではなく、CtLocalVariable<sup>Master</sup>として表現されます。これには初期化式を設定できないことに注意が必要です。

表 46. CtForEach が提供するメソッド

メソッド名	概要
getParameter	形式的引数の宣言を返す
getIterable	反復対象の式を返す
getBody	ループ本体を返す

## 第14節 break 文

break 文(JLS3-14.15 The break Statement)は CtBreak 型のインターフェースとして宣言され、ループ構造や switch 文に含まれる通常の文として利用することが可能です。

CtBreakはメソッド内でジャンプを行う文を表現するCtJumpインターフェースのサブインターフェースとして宣言されており、ほとんどの機能はCtJumpで提供されます(表 47)。メソッド *getTargetLabel*はCtLabel<sup>Slave</sup>を返し、ラベル付き文(第 2 節)で宣言されたCtLabel<sup>Master</sup>に対する参照を表現します。ラベルが省略された場合は無効な参照を表現する存在しないDOM要素を返します。

表 47. CtBreak が提供するメソッド

メソッド名	概要
getTargetLabel	分岐先のラベルを返す

## 第15節 continue 文

continue 文(JLS3-14.16 The continue Statement)は CtContinue 型のインターフェースとして宣言され、ループ構造に含まれる通常の文として利用することが可能です。

CtContinueはメソッド内でジャンプを行う文を表現するCtJumpインターフェースのサブインターフェースとして宣言されており、ほとんどの機能はCtJumpで提供されます(表 48)。これは、CtBreak(第 14 節)が提供する機能と同様です。

**表 48. CtContinue が提供するメソッド**

メソッド名	概要
getTargetLabel	分岐先のラベルを返す

## 第16節 return 文

return 文(JLS3-14.17 The return Statement)は CtReturn 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtReturnは表 49にある機能を提供します。void型のメソッド等で返戻する値を明示的に指定しない場合、メソッド*getResult*は無効な式を表現する存在しないDOM要素を返します。

**表 49. CtReturn が提供するメソッド**

メソッド名	概要
getResult	返戻する式を返す

## 第17節 throw 文

throw 文(JLS3-14.18 The throw Statement)は CtThrow 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtThrowは表 50にある機能を提供します。この式の評価結果はjava.lang.Throwableまたはそのサブタイプでなければなりません。

**表 50. CtThrow が提供するメソッド**

メソッド名	概要
getThrowable	スローする式を返す

## 第18節 synchronized 文

synchronized 文(JLS3-14.19 The synchronized Statement)は CtSynchronized 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtSynchronizedは表 51にある機能を提供します。

**表 51. CtSynchronized が提供するメソッド**

メソッド名	概要
-------	----

メソッド名	概要
getLock	ロックに利用する式を返す
getBody	本体を返す

## 第19節 try 文

try 文(JLS3-14.20 The try statement)は CtTry 型のインターフェースとして宣言され、通常の文として利用することが可能です。

CtTryは表 52にある機能を提供します。try文に含まれるcatch節は複雑な構造をしているため、それぞれがCtCatchインターフェースで表現され、メソッド*getCatches*によって付与されたcatch節の一覧を取得することができます。また、finallyを持たないtry文に対してメソッド*getFinally*が呼び出された場合、空のブロックを表す存在しないDOM要素が返されます。

表 52. CtTry が提供するメソッド

メソッド名	概要
getTry	try 節の本体を返す
getCatches	catch 節の一覧を返す
getFinally	finally 節の本体を返す

try文で使用されるCtCatchは、表 53にある機能を提供します。例外パラメータはメソッドパラメータを表すCtParameter<sup>Master</sup>ではなく、CtLocalVariable<sup>Master</sup>として表現されます。これには初期化式を設定できないことに注意が必要です。

表 53. CtCatch が提供するメソッド

メソッド名	概要
getParameter	例外パラメータを返す
getBody	catch 節の本体を返す

## 第9章 式

式(JLS3-15 Expressions)を表現するすべてのDOM要素は、CtExpressionのサブインターフェイスで表現されます。式とは評価(JLS3-15.1 Evaluation, Denotation, and Result)可能な言語要素で、プログラムは式の評価とその評価結果を変数に代入することによって実行されます。また、いくつかの式はそのまま文(第8章)としても利用することができます。このような式を式文と呼び、第8章第1節で説明があります。

CtExpressionそのものは多くの機能を持たず、表54にある機能のみを提供します。各式の評価結果はJava言語仕様に準拠し、その型はDOM要素の型(第4章)として表現されます。

表 54. CtExpression が提供するメソッド

メソッド名	概要
evalType	評価結果が持つ型を返す

### 第1節 括弧付きの式

括弧付きの式(JLS3-15.8.5 Parenthesized Expressions)は Irenka DOM 上に出現しません。Java 言語仕様における括弧付きの式は式全体の評価順序に影響を与えますが、Irenka DOM はプログラムの意味を元にして構造を生成しており、そこには式の評価順序に関する情報も含まれます。そのため、明示的な括弧を表現するような DOM 要素を必要としません。

図2は" $1 + 2 * 3$ "という式と" $(1 + 2) * 3$ "という式をそれぞれ Irenka DOM として表現した場合の図です。後者は表記に括弧を含みますが、そのツリー上には括弧を表す DOM 要素は出現しません。代わりに、どちらも評価順序が先であるものがツリーの深い位置に配置されています。二項演算式(第13節)はその項を先に評価しますので、図では深い位置に配置された要素が先に評価されることになります。

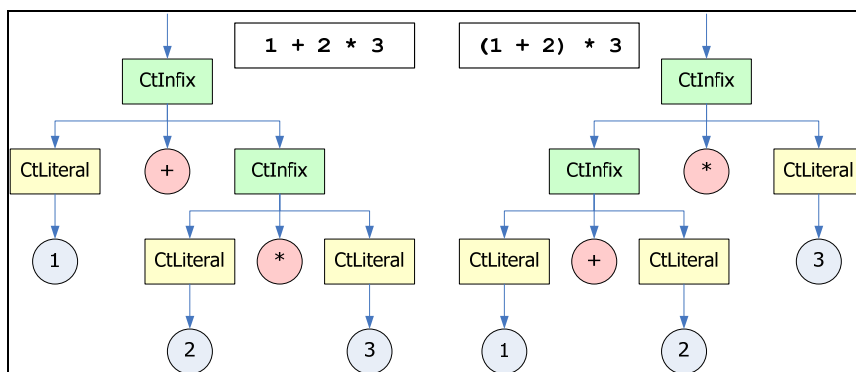


図 2.  $1+2*3$  と  $(1+2)*3$  に対応する Irenka DOM

## 第2節 字句上のリテラル

字句上のリテラル (JLS3-15.8.1 Lexical Literals)は定数を表現するリテラルで、CtLiteral型のインターフェースで表現されます。プログラミング言語Javaでは様々な種類のリテラルを表現することができますが、Irenka DOM上ではそれらを全てCtLiteralとして取り扱います。対象リテラルの種類を検査するには、メソッド*getLiteralKind*を利用します。このメソッドはリテラルの種類を列挙したLiteralKind型の値を返し、それぞれの列挙定数は対応する字句上のリテラルの種類を表現します(表 55)。

表 55. LiteralKind の種類

定数の名前	対応するリテラル
INT	整数リテラル(32bit)
LONG	整数リテラル(64bit)
FLOAT	浮動小数点数リテラル(単精度)
DOUBLE	浮動小数点数リテラル(倍精度)
CHAR	文字リテラル
BOOLEAN	真偽リテラル
STRING	文字列リテラル
NULL	null リテラル

CtLiteralは表 56にある機能を提供します。このうち、メソッド*getValue*はリテラルをJava言語仕様に従って解釈した際の実行時の値を返しますが、その値がプリミティブ型で表現される場合はボックス変換を適用した値を返します。

表 56. CtLiteral が提供するメソッド

メソッド名	概要
<i>getLiteralKind</i>	リテラルの種類を返す
<i>getLiteralString</i>	リテラルとして表記された文字列を返す
<i>getValue</i>	リテラルを解釈した実行時の値を返す

## 第3節 クラス・リテラル

クラス・リテラル (JLS3-15.8.2 Class Literals)は対象の型を表現するjava.lang.Class型のリテラルで、CtClassLiteral型のインターフェースで表現されます。これは字句上のリテラル(第2節)が持つ値と異なり、Irenka DOM上の型(第4章)を利用して表現されています。

CtClassLiteralは表 57にある機能を提供します。

表 57. CtClassLiteral が提供するメソッド

メソッド名	概要
<i>getValue</i>	対象の型を返す

## 第4節 this

this キーワード (JLS3-15.8.3 this, JLS3-15.8.4 Qualified this)はメソッドが起動されたオブジェクトや、コンストラクタで構築中のオブジェクトに対する参照を表す式です。これは CtThis 型のインターフェイスで表現されます。

CtThisは表 58にある機能を提供します。限定されたthisなどでスコープ外の型を参照している場合でも、メソッド*getType*は適切な型を返すことができます。

表 58. CtThis が提供するメソッド

メソッド名	概要
getType	対象の型を返す

## 第5節 クラス・インスタンス生成式

クラス・インスタンス生成式(JLS3-15.9 Class Instance Creation Expressions)は、クラスのインスタンスである新しいオブジェクトを生成するための式で、CtNewInstance 型のインターフェイスで表現されます。

CtNewInstanceは起動可能なメンバの起動式を表現するCtInvocationインターフェイスのサブインターフェイスとして宣言されており、いくつかの機能はCtInvocation内で提供されます。表 59はCtNewInstanceが提供する機能です。

表 59. CtNewInstance が提供するメソッド

メソッド名	概要
getQualifier	限定式を返す
getTarget	対象のCtConstructor <sup>Slave</sup> を返す
getArguments	実引数式の一覧を返す
getBody	匿名ブロックを返す <sup>7</sup>

## 第6節 配列生成式

配列生成式(JLS3-15.10 Array Creation Expressions)は、新たな配列を生成する式で、CtNewArray 型のインターフェイスで表現されます。

CtNewArrayは表 60にある機能を提供します。メソッド*getDimensions*は配列の各次元の要素数を表す式からなるリスト(次元式のリスト)を返し、*getInitialElements*は初期要素を表す式からなるリスト(配列初期化子)を返します。次元式のリストと配列初期化子を表すリストは同時に要素を設定することができず、かつ同時に空であってはけません。

<sup>7</sup> 未実装

表 60. CtNewArray が提供するメソッド

メソッド名	概要
getType	生成対象の配列型を返す
getDimensions	各次元の要素数を返す
getInitialElements	初期要素の一覧を返す

## 第7節 変数アクセス式

変数アクセス式(JLS3-6.5.6 Meaning of Expression Names, JLS3-15.11 Field Access Expressions)は宣言された変数を参照するための式で、CtVariableAccess型のインターフェースで表現されます。参照可能なすべての変数はCtVariable<sup>Slave</sup>型のインターフェースを有し、そのサブインターフェースとしてCtLocalVariable<sup>Slave</sup>、CtParameter<sup>Slave</sup>、CtField<sup>Slave</sup>、CtEnumConstant<sup>Slave</sup>があります。

CtVariableAccessは表 61にある機能を提供します。このうち、限定式はフィールド(第 6 章 第 1 節)または列挙定数(第 6 章 第 2 節)を参照対象にとる場合のみ利用可能で、それ以外では無効な存在しない式を表すDOM要素を返します。

表 61. CtVariableAccess が提供するメソッド

メソッド名	概要
getQualifier	限定式を返す
getVariable	参照対象の変数を返す

## 第8節 配列長参照式

配列長参照式(JLS3-6.5.6.2 Qualified Expression Names)は、対象配列の要素数を取得するための式で、CtArrayLength型のインターフェースで表現されます。

CtArrayLengthは表 62にある機能を提供します。

表 62. CtArrayLength が提供するメソッド

メソッド名	概要
getArray	対象の配列を表す式を返す

## 第9節 メソッド起動式

メソッド起動式(JLS3-15.12 Method Invocation Expressions)は、特定のメソッド(第 6 章 第 3 節)を起動するための式で、CtMethodInvocation型のインターフェースで表現されます。

CtMethodInvocationは起動可能なメンバの起動式を表現するCtInvocationインターフェースのサブインターフェースとして宣言されており、いくつかの機能はCtInvocation内で提供されま

す。表 63はCtMethodInvocationが提供する機能です。単純メソッド呼び出しやクラスメソッドの起動を表現する場合、限定式を指定する必要はありません。

表 63. CtMethodInvocation が提供するメソッド

メソッド名	概要
getQualifier	限定式を返す
getTarget	起動対象のCtMethod <sup>Slave</sup> を返す
getArguments	実引数式の一覧を返す
inline	この起動をインライン展開した疑似文を返す

メソッド`inline`はメソッド起動式特有の機能で、対象メソッドのインライン化をすることができます。インライン化は、起動対象のメソッドを指定した実引数を利用して展開し、その内容を単一の式文(第 8 章第 1 節)へと変換します。現在はインライン化できるメソッド起動式に制限があり、定義 12にある条件を満たしたものに対してのみ有効です。

インライン化可能なメソッド起動式は次のいずれかを満たす。

1. 起動対象の戻り値型が `void` であるメソッドとして宣言され、かつメソッドに含まれるすべての参照がインライン化先からも参照可能である
2. そうでなく、起動対象のメソッドブロックの末尾に唯一の `return` 文を有し、かつメソッドに含まれるすべての参照がインライン化先からも参照可能である

#### 定義 12. インライン化可能なメソッド

## 第10節 配列アクセス式

配列アクセス式(JLS3-15.13 Array Access Expressions)は、配列の各要素を取得するための式で、CtArrayAccess 型のインターフェースで表現されます。

CtArrayAccessは表 64にある機能を提供します。

表 64. CtArrayAccess が提供するメソッド

メソッド名	概要
getArray	対象の配列を表す式を返す
getIndex	添え字を表す式を返す

## 第11節 単項演算式

単項演算式は後置式(JLS3-15.14 Postfix Expressions)と単項演算式を(JLS3-15.15 Unary Operators)を組み合わせたもので、Irenkaではこれらを合わせて単純に単項演算式と呼びます。単項演算式はCtUnary型のインターフェースで表現され、演算の内容に合った単項

演算子を有します。それぞれの単項演算子は、対応する列挙UnaryOperatorでのいずれか表現されます(表 65)。

**表 65. UnaryOperator の種類**

定数の名前	演算子	種類	概要
POSTFIX_DECREMENT	--	後置	後置デクリメント
POSTFIX_INCREMENT	++	後置	後置インクリメント
COMPLEMENT	~	前置	ビット反転
MINUS	-	前置	符号反転
NOT	!	前置	論理反転
PLUS	+	前置	単項プラス
PREFIX_DECREMENT	--	前置	前置デクリメント
PREFIX_INCREMENT	++	前置	前置インクリメント

CtUnaryは表 66にある機能を提供します。

**表 66. CtUnary が提供するメソッド**

メソッド名	概要
getOperator	演算対象の式を返す
getOperand	演算子(UnaryOperator)を返す

## 第12節 キャスト式

キャスト式(JLS3-15.16 Cast Expressions)は数値型を持つ値を別の数値型の近似値へと変換したり、クラス・オブジェクトを互換性のある別の方へ変換したりする際に利用される式です。これは、CtCast 型のインターフェースで表現されます。

CtCastは表 67にある機能を提供します。

**表 67. CtCast が提供するメソッド**

メソッド名	概要
getType	キャスト先の型を返す
getExpression	対象の式を返す

## 第13節 二項演算式

二項演算式は1つの中置演算子と2つの式からなる演算式で、CtInfix 型のインターフェースで表現されます。これは、Java 言語仕様で定義される下記の演算に対する式を表現することが可能です。

- 乗除(JLS3-15.17 Multiplicative Operators)

- 加減(JLS3-15.18 Additive Operators)
- シフト(JLS3-15.19 Shift Operators)
- 数値比較(JLS3-15.20.1 Numerical Comparison Operators)
- 等値比較(JLS3-15.21 Equality Operators)
- ビット演算、論理演算(JLS3-15.22 Bitwise and Logical Operators)
- 条件 And 演算(JLS3-15.23 Conditional-And Operator)
- 条件 Or 演算(JLS3-15.24 Conditional-Or Operator)

CtInfixの演算内容はその演算子で決まり、二項演算式が利用する演算子是对应する列挙InfixOperatorのいずれかで表現されます。列挙InfixOperatorは代入演算式(第 16 節)で利用する演算子を含み、一部は二項演算式で利用することができません(表 68)。

**表 68. InfixOperator の種類(CtInfix 向けのみ)**

定数の名前	演算子	分類	概要
TIMES	*	乗除	乗算
DIVIDE	/	乗除	除算
REMINDER	%	乗除	剰余
PLUS	+	加減	加算、文字列連結
MINUS	-	加減	減算
LEFT_SHIFT	<<	シフト	左ビットシフト
RIGHT_SHIFT_SIGNED	>>	シフト	算術右ビットシフト
RIGHT_SHIFT_UNSIGNED	>>>	シフト	論理右ビットシフト
LESS	<	関係	未満
LESS_EQUALS	<=	関係	以下
GREATER	>	関係	超過
GREATER_EQUALS	>=	関係	以上
EQUALS	==	等値	等値
NOT_EQUALS	!=	等値	偽等値
AND	&	ビット	論理積
OR		ビット	論理和
XOR	^	ビット	排他的論理和
CONDITIONAL_AND	&&	短絡	短絡論理積
CONDITIONAL_OR		短絡	短絡論理和

CtInfixは表 69にある機能を提供します。

**表 69. CtInfix が提供するメソッド**

メソッド名	概要
-------	----

メソッド名	概要
getLeftOperand	演算対象の左項を返す
getOperator	演算子(InfixOperator)を返す
getRightOperand	演算対象の右項を返す

## 第14節 instanceof 式

instanceof 式(JLS3-15.20.2 Type Comparison Operator)はクラス・オブジェクトの型比較を行うための式で、CtInstanceof 型のインターフェースで表現されます。

CtInstanceofは表 70にある機能を提供します。

表 70. CtInstanceof が提供するメソッド

メソッド名	概要
getExpression	対象の式を返す
getType	対象の型を返す

## 第15節 条件演算式

条件演算子式(JLS3-15.25 Conditional Operator)は条件式の真偽値を利用して2つの式のどちらかを選択して評価する式で、その演算子を三項演算子とも呼びます。これは、CtConditional 型のインターフェースで表現されます。

CtConditionalは表 71にある機能を提供します。

表 71. CtConditional が提供するメソッド

メソッド名	概要
getCondition	条件式を返す
getThen	条件成立時に実行される式を返す
getElse	条件不成立時に実行される式を返す

## 第16節 代入演算式

代入演算式(JLS3-15.26 Assignment Operators)は変数や配列へ何らかの値を代入するための式で、CtAssignment型のインターフェースで表現されます。代入演算子の左辺にとる式は、必ず変数アクセス式(第7節)または配列アクセス式(第10節)のいずれかでなくてはなりません。これらはいずれも代入可能な式を表現するCtAssignable型のサブインターフェースとして定義されています。

代入演算が使用する代入演算子は単純代入演算子と複合代入演算子に分類可能で、表 72のように列挙InfixOperatorのいずれかで表現することが可能です。列挙InfixOperator

は二項演算式(第 13 節)でも利用されており、一部は代入演算式で利用することができません。

表 72. InfixOperator の種類(CtInfix 向けのみ)

定数の名前	演算子	分類	概要
ASSIGN	=	単純代入	代入
TIMES	*=	複合代入	乗算
DIVIDE	/=	複合代入	除算
REMINDER	%=	複合代入	剰余
PLUS	+=	複合代入	加算、文字列連結
MINUS	-=	複合代入	減算
LEFT_SHIFT	<<=	複合代入	左ビットシフト
RIGHT_SHIFT_SIGNED	>>=	複合代入	算術右ビットシフト
RIGHT_SHIFT_UNSIGNED	>>>=	複合代入	論理右ビットシフト
AND	&=	複合代入	論理積
OR	=	複合代入	論理和
XOR	^=	複合代入	排他的論理和

CtAssignmentは表 73にある機能を提供します。左辺式を返すメソッド*getLeftHandSide*はCtAssignable型の式を返します。

表 73. CtAssignment が提供するメソッド

メソッド名	概要
getLeftHandSide	代入先の変数式を返す
getOperator	演算子(InfixOperator)を返す
getRightHandSide	代入する式を返す

## 第10章 宣言の属性

宣言を表す `CtReferenceMaster` (第3章)の多くには、注釈、修飾子、Javaドキュメンテーションコメントを付与することができます。

この章では、型として表現された注釈(`CtAnnotation`)を"注釈型"とよび、宣言に付与された注釈(`CtAnnotationInstance`)を"注釈"とよび、それぞれを区別します。

### 第1節 注釈

注釈(JLS3-9.7 Annotations)は構造化されたメタデータを宣言に提供するための構文要素で、`CtAnnotationInstace` 型のインターフェースで表現されます。さらに、`CtAnnotationInstance` が持つ注釈要素と値のペア 1 つ分は `CtAnnotationInstanceElement` 型のインターフェースで表現します。

`CtAnnotationInstance`は表 74にある機能を提供します。メソッド `getElements`は注釈に記述された注釈要素の一覧を `CtAnnotationInstanceElement`のリストとして返しますが、これは注釈として実際に表記されたもののみが含まれ、規定値を利用する要素については含まれません。規定値を利用した要素の値を調べる場合は、メソッド `getValueMap`を利用するか、メソッド `getType`を利用して注釈型の宣言を調べる必要があります。

表 74. `CtAnnotationInstance` が提供するメソッド

メソッド名	概要
<code>getType</code>	注釈型( <code>CtAnnotation<sup>Slave</sup></code> )を返す
<code>getElements</code>	表記された注釈要素の一覧を返す
<code>getElement</code>	表記された注釈要素を返す
<code>getValueMap</code>	注釈要素名と対応する値からなる map を返す

`CtAnnotationInstance`は 0 個以上の `CtAnnotationInstanceElement`を有し、それぞれは表 75にあるような機能を持ちます。

表 75. `CtAnnotationInstanceElement` が提供するメソッド

メソッド名	概要
<code>getTarget</code>	注釈要素( <code>CtAnnotationElement<sup>Slave</sup></code> )を返す
<code>getValue</code>	宣言された値を返す

### 第2節 修飾子

全ての修飾子は `CtModifier`型のインターフェースで表現され、それぞれの種類は対応する列挙 `ModifierKind`で表現します。表 76は利用可能な修飾子の種類を表しますが、BRIDGE、

SUPER, SYNTHETIC, VARARGSというJava言語仕様に出現しないキーワードを含みます。これらはそれぞれ、Java仮想マシン仕様のクラスファイル形式(JSR-202)に出現するACC\_BRIDGE, ACC\_SUPER, ACC\_SYNTHETIC, ACC\_VARARGSに対応します。

表 76. ModifierKind の種類

定数の名前	対応する修飾子	適用例
ABSTRACT	abstract	抽象クラス、抽象メソッド
BRIDGE	(なし)	ブリッジメソッド
FINAL	final	継承禁止クラス、変更禁止フィールド
NATIVE	native	ネイティブメソッド
PRIVATE	private	非公開メンバ
PROTECTED	protected	継承プライベートメンバ
PUBLIC	public	公開メンバ
STATIC	static	クラスメンバ
STRICTFP	strictfp	厳密な浮動小数点演算を行うメソッド
SUPER	(なし)	ACC_SUPER 指定されたクラス
SYNCHRONIZED	synchronized	オブジェクトモニタを利用するメソッド
SYNTHETIC	(なし)	コンパイラによって合成されたメンバ
TRANSIENT	transient	シリアライズされないフィールド
VARARGS	(なし)	可変長引数を持つメソッド
VOLATILE	volatile	TLS を利用しないフィールド

CtModifierはその種類を取得するメソッドのみを提供します(表 77)。

表 77. CtModifier が提供するメソッド

メソッド名	概要
getKind	修飾子の種類(ModifierKind)を返す

### 第3節 ドキュメンテーションコメント

ドキュメンテーションコメント(JLS1-18 Documentation Comments)はそれぞれの宣言に1つだけ付与させることができ、その宣言に対するドキュメントをコメントとして記述することができます。この要素は CtJavadoc 型のインターフェースで表現されます。

CtJavadocは表 78にある機能を提供します。CtJavadocは表記されたコメント全体を表し、メソッド *getBody*によってコメントに含まれる構造化されたドキュメントの内容を取得することができます。ドキュメントの内容はDocBody型のインターフェースで表現され、詳しくは第 11 章にその説明があります。

表 78. CtJavadoc が提供するメソッド

メソッド名	概要
getText	コメント全体の文字列を返す
getBody	ドキュメント本体を返す

## 第11章 Javadoc

Javadoc というのはソースプログラム上のドキュメンテーションコメントを解釈して API ドキュメントを生成する Sun が提供するツールの名称です。この章では、同ツールが処理可能なドキュメンテーションコメントを単純に Javadoc と呼びます。

Javadocは現在のJava言語仕様(JLS3)で規定されておらず、Java言語の構文要素ではありません。IrenkaではJava言語仕様第1版(JLS1)に従ってJavadocを解釈し、CtJavadoc型のインターフェースで表現されるDOM要素として出現します。定義 13はIrenkaが構築するJavadocのDOM要素としての表現を定義します。ゴール記号は*CtJavadoc*で、これはCtJavadoc型のインターフェースで表現されます(第10章第3節)。

```

CtJavadoc:
    DocBody
DocBody:
    DocBlock*
DocBlock:
    DocTag? DocElement*
DocTag:
    STRING
DocElement:
    DocText
    DocInline
DocText:
    STRING
DocInline:
    DocTag DocElement*

```

### 定義 13. Javadoc の構造

これらの要素を表現するインターフェースは、org.ashikunep.irenka.dom.javadoc パッケージに配置されています。

## 第1節 ドキュメント本体

Javadocのドキュメント本体はDocBody型のインターフェースで表現されます。これはJavadocから実際にドキュメントとして利用されない部分<sup>8</sup>(JLS1-18.1 The Text of a Documentation Comment)はドキュメント本体には含まれません。

<sup>8</sup> 先頭の"/\*\*", 末尾的"/", および行頭の" \* "などはドキュメンテーションコメントの解釈時にスキップされます

DocBlockは表 79にある機能を提供します。メソッド*getBlocks*はドキュメント本体に含まれるブロック要素(第 2 節)

表 79. DocBody が提供するメソッド

メソッド名	概要
getText	ドキュメント本体の文字列を返す
getBlocks	ブロックの一覧を返す

## 第2節 ブロック要素

ブロック要素とはJavadocの先頭に配置された概要ブロック (JLS1-18.3 Summary Sentence and General Description)と、それ以降に配置されたタグ付きブロック (JLS1-18.4 Tagged Paragraphs)の総称です。概要ブロックは先頭のタグ(第 3 節)を持たず通常のテキストで表記され、タグ付きブロックはブロックの先頭にタグが付与され、そのタグが表現する要素についての解説を記述することができます。これらはいずれも、DocBlock型のインターフェースで表現されます。

DocBlockは表 80にある機能を持ちます。メソッド*getTag*はブロックに付与されたタグ(第 3 節)を返しますが、概要ブロックはタグを持たないため、存在しないタグを表現するDOM要素が返されます。また、メソッド*getElements*はブロックの内容を返します。ブロックの内容は*DocElement*インターフェースを持つ要素のリストで表現され、これはDocText(第 4 節)型およびDocInline(第 5 節)型に共通する親インターフェースです。

表 80. DocBlock が提供するメソッド

メソッド名	概要
getText	ブロックの文字列を返す
getTag	タグを返す
getElements	要素の一覧を返す

## 第3節 タグ

タグはタグ付きブロック(第 3 節)の先頭に出現する要素で、DocTag型のインターフェースで表現されます。

DocTagは表 81にある機能を提供します。DocTagを有するブロック要素が概要ブロックであった場合、メソッド*isEmptyTag*は真を返します。このような場合、このタグ要素は無効であると解釈されます。

表 81. DocTag が提供するメソッド

メソッド名	概要
getText	タグ全体の文字列を返す
getName	タグの名前を返す

メソッド名	概要
isEmptyTag	存在しないタグであるかどうかを検査する

## 第4節 テキスト

テキストは Javadoc に出現する通常のテキストのことで、DocText 型のインターフェースで表現されます。

DocTextは表 82にある機能を提供します。また、DocTextはブロック要素内に出現する要素を表すDocElementインターフェースのサブインターフェースとして宣言されています。

**表 82. DocText が提供するメソッド**

メソッド名	概要
getText	文字列を返す

## 第5節 インラインタグブロック

インラインタグブロックはブロック要素(第 2 節)の中に出現し、"{"と"}"に囲まれてタグ付きブロックと同じ構造を持ちます。これは、DocInline型のインターフェースで表現されます。

DocInlineは表 83にある機能を提供します。また、DocInlineはブロック要素内に出現する要素を表すDocElementインターフェースのサブインターフェースとして宣言されています。

**表 83. DocInline が提供するメソッド**

メソッド名	概要
getText	インラインタグブロック全体の文字列を返す
getTag	タグを返す
getElements	要素の一覧を返す

## 第12章 ソースプログラム

プログラミング言語 Java の文法 (JLS3-2 Grammars) に従って記述されたテキストをソースプログラムと呼び、そこから Irenka DOM の一部を作成することができます。しかし、Irenka DOM はプログラムの意味を元に構成されたモデルで、構文などの表記を元にしたモデルではありません。そのため、ソースコード上での表記などの情報はすべて付加情報として取り扱われます。

この章では、表記に関する付加情報を取り扱うための方法について説明します。パーサは独自の方法で表記についての情報を付与することもあり、ここにはない情報についてはそれぞれのパーサに対するドキュメントも併せて参照してください。

なお、この章で紹介する要素を表すインターフェースは `org.ashikunep.irenka.dom.source` パッケージに配置されています。

### 第1節 コンパイル単位

コンパイル単位 (JLS3-7.3 Compilation Units) は 1 つ分のソースプログラムを表現し、`CtCompilationUnit` 型のインターフェースで表現されます。コンパイル単位は次の 3 つの構造を有します。なお、これらはいずれも省略することができます。

1. パッケージ宣言 (`CtPackageSlave`)<sup>9</sup>
2. インポート宣言 (`CtImport`) の一覧
3. トップレベルの型宣言 (`CtDeclaredTypeSlave`)<sup>10</sup> の一覧

`CtCompilationUnit` は表 84 のような機能を提供し、これらを利用して上記の構造を参照することができます。

表 84. `CtCompilationUnit` が提供するメソッド

メソッド名	概要
<code>getName</code>	コンパイル単位の名前を返す
<code>getPackage</code>	パッケージ宣言 ( <code>CtPackage<sup>Slave</sup></code> ) を返す
<code>getImports</code>	インポート宣言 ( <code>CtImport</code> ) の一覧を返す
<code>getTypes</code>	宣言された型 ( <code>CtDeclaredType<sup>Slave</sup></code> ) の一覧を返す
<code>findType</code>	宣言された型を返す

<sup>9</sup> コンパイル単位におけるパッケージ宣言は、コンパイル単位に記載された宣言型が属するパッケージを宣言するための構文で、パッケージそのものを宣言しません。`CtPackage` はパッケージそのものを表現しているため、`CtCompilationUnit` が返すパッケージ宣言は `CtPackageMaster` ではなく `CtPackageSlave` を利用します。なお、`CtPackageMaster` は Irenka DOM の名前空間を構築する DOM 要素です。

<sup>10</sup> `CtPackageSlave` と同様に、`CtCompilationUnit` 上での型の宣言も `CtDeclaredTypeSlave` として表現されます。

メソッド名	概要
getCorrespondedFile	コンパイル単位を構成するファイルを返す

現在の仕様では CtCompilationUnit は CtElement のサブインターフェースではなく、DOM 要素として取り扱うことができません。ただし、CtElement のメソッド *getCompilationUnit* を利用することによって、対象の DOM 要素が記載されたコンパイル単位を取得することができます。

## 第1項 パッケージ宣言

パッケージ宣言 (JLS3-7.4 Package Declarations) はコンパイル単位に記載された宣言型が属するパッケージを宣言することができます。このパッケージ宣言は CtPackage<sup>Slave</sup> 型のインターフェースで表現されます。なお、パッケージ宣言が省略された場合、CtCompilationUnit のメソッド *getPackage* は無名パッケージ (JLS3-7.4.2 Unnamed Packages) への参照を返します。

CtPackage<sup>Slave</sup> は特に機能を持たず、そのマスタ参照である CtPackage<sup>Master</sup> を経由して様々な情報を取得します。CtPackage<sup>Master</sup> の機能は表 85 のとおりです。

表 85. CtPackage<sup>Master</sup> が提供するメソッド

メソッド名	概要
getSimpleName	単純名を返す
getMembers	パッケージメンバの一覧を返す
getDeclaredPackages	サブパッケージの一覧を返す
getDeclaredTypes	トップレベルタイプの一覧を返す
getDeclaredPackage	サブパッケージを返す
getDeclaredType	トップレベルタイプを返す
getParentPackage	親パッケージを返す
isDefault	無名パッケージかどうか検査する
getName	完全限定名を返す

## 第2項 インポート宣言

インポート宣言 (JLS3-7.5 Import Declarations) は、宣言されたコンパイル単位内で異なるパッケージ上に宣言された型や、異なる宣言型上に宣言されたメンバを単純名で参照するために利用されます。これは、CtImport 型のインターフェースで表現されます。

インポート宣言は 4 種類存在し、インポートの種類は ImportKind という列挙によって表現されます (表 86)。そして、CtImport はメソッド *getKind* によってそのインポートの種類を表す ImportKind 型の列挙定数を返します。

表 86. ImportKind の種類

定数名	概要
TYPE	単一の型インポート宣言
TYPE_ON_DEMAND	オンデマンドの型インポート宣言
STATIC	単一の static インポート宣言

定数名	概要
STATIC_ON_DEMAND	オンデマンドの static インポート宣言

インポート宣言は名前空間(*getNameSpace*)とメンバ名(*getMemberName*)からなり、前者はCtNamespace<sup>Slave</sup>型(第2章第2節)で表現され、後者はjava.lang.String型で表現されます。それぞれの値は表 87に従って、インポート宣言の種類によって制約がかけられます。

表 87. インポート宣言ごとのプロパティの値

インポート宣言の種類	名前空間の型	メンバ名の有無
単一の型インポート宣言	CtDeclaredType <sup>Slave</sup>	無し
オンデマンドの型インポート宣言	CtPackage <sup>Slave</sup>	無し
単一の static インポート宣言	CtDeclaredType <sup>Slave</sup>	有り
オンデマンドの static インポート宣言	CtDeclaredType <sup>Slave</sup>	無し

CtImportは表 88のような機能を有します。インポート宣言の種類、名前空間、メンバ名を参照するメソッドをそれぞれ提供しています。

表 88. CtImport が提供するメソッド

メソッド名	概要
getKind	インポート宣言の種類(ImportKind)を返す
getNameSpace	インポート先の名前空間を返す
getMemberName	メンバ名を返す

### 第3項 トップレベルの型宣言

トップレベルの型宣言(JLS3-7.6 Top Level Type Declarations)は、トップレベル(DeclaredScopeKind.TOP\_LEVEL, 第5章)に宣言されたCtDeclaredType<sup>Master</sup>に対する参照、CtDeclaredType<sup>Slave</sup>として表現されます。これは通常の宣言型であるので、第4章第2節を参照してください。ただし、これらはいずれもパラメータ化されていない宣言型として表現されます。

### 第2節 位置情報

CtElementは、それが表すDOM要素が表記された位置を取得するための機能が用意されています(表 89)。これらの情報は、位置情報を記録するコンパイラが対象のDOM要素を生成した場合のみ取得することができます。

表 89. CtElement が提供する表記に関するメソッド

メソッド名	概要
getCorrespondedFile	この要素が表記されたファイル(CtFile)を返す
getLocation	この要素が表記されたファイル内の位置(CtLocation)を返す

## 参考文献

---

- Gosling, J., Joy, B., & Steele, G. L. (1996). *The Java(TM) Language Specification*. Addison Wesley Publishing Company.
- Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *The Java(TM) Language Specification* (Third ed.). Prentice Hall PTR.
- Irenka End User API
  - <http://irenka.ashikunep.org/documents/api>

## 用語索引

---

### D

DOM 要素	1, 2, 3, 12, 30, 53
親要素	4
合成された	3
子要素	4
種類	3
操作	3
存在しない	3, 6, 7
同値	9
複製	7
プロパティ	6, 7, 8, 10, 14
文脈	8
文脈依存変換	8

### I

Irenka DOM	1, 2
------------	------

### J

Javadoc	49
インラインタグブロック	49, 51
概要ブロック	50
タグ	49, 50
タグ付きブロック	50
テキスト	49, 51
ドキュメント本体	49
ブロック要素	49, 50

### N

null 型	18
--------	----

### T

this キーワード	40
------------	----

### い

インターフェース	16, 20, 21
宣言	21
インポート	52, 53
宣言	52, 53

### か

外部属性	3
型	15, 38, 52, 54
互換	16
宣言	5, 20, 52, 54
トップレベル	54
型境界	19
型変数	17, 20, 28
宣言	20, 28

### き

基本型	16
共通型	19

### く

クラス	16, 20
宣言	20
クラス・リテラル	39

### こ

後置式	42
コンストラクタ	25
宣言	25
コンパイル単位	52

---

**さ**

参照.....	12, 16, 28
ジェネリック参照.....	28
スレーブ.....	16
マスタ.....	16

---

**し**

ジェネリック参照.....	17
式 30, 38	
instanceof 式.....	45
this キーワード.....	40
括弧付きの式.....	38
キャスト式.....	43
クラス・インスタンス生成式.....	24
クラス・リテラル.....	39
三項演算子.....	45
条件演算式.....	45
代入演算式.....	45
単項演算式.....	42
二項演算式.....	43
配列アクセス式.....	42
配列生成式.....	40
配列序参照式.....	41
評価.....	38
変数アクセス式.....	41
メソッド起動式.....	41
リテラル.....	39
式文.....	30, 38
修飾子.....	27, 47
初期化子.....	26
インスタンス.....	27
クラス.....	27
宣言.....	26

---

**す**

スレーブ参照.....	12, 14, 16, 23
-------------	----------------

---

**せ**

宣言.....	12
宣言型.....	16, 53, 54

---

**そ**

総称化コンテキスト.....	29
存在しない型.....	19

---

**た**

代入演算子.....	45
単純代入演算子.....	45
複合代入演算子.....	45

---

**ち**

注釈.....	16, 20, 22, 47
型 16	
宣言.....	22
注釈要素.....	26, 47
宣言.....	26

---

**と**

ドキュメンテーションコメント.....	48
匿名クラス.....	20
宣言.....	20

---

**な**

名前空間.....	54
-----------	----

---

**は**

配列型.....	18
パッケージ.....	20, 21, 22, 52, 53
宣言.....	20, 21, 22, 52, 53
パッケージメンバ.....	5
パラメータ化.....	28
パラメータ化型.....	17, 19, 54

<hr/>	
<b>ひ</b>	
引数.....	25
宣言.....	25
<hr/>	
<b>ふ</b>	
フィールド.....	24
宣言.....	24
ブロック.....	31
文 30, 38	
assert 文.....	33
break 文.....	35
continue 文.....	35
do 文.....	34
for 文.....	34
if 文.....	32
return 文.....	36
switch 文.....	33
synchronized 文.....	36
throw 文.....	36
try 文.....	37
while 文.....	34
拡張 for 文.....	35
空文.....	32
ラベル.....	30
<hr/>	
<b>ほ</b>	
ボクシング変換.....	16
捕捉型.....	19
捕捉変換.....	19
<hr/>	
<b>ま</b>	
マスタ参照.....	12, 13, 23
<hr/>	
<b>め</b>	
メソッド.....	24
<hr/>	
宣言.....	24
メンバ.....	20, 23
宣言.....	20
メンバ型.....	20, 21, 22, 27
宣言.....	20, 21, 22, 27
<hr/>	
<b>ら</b>	
ラッパ型.....	16
<hr/>	
<b>り</b>	
リテラル.....	39
<hr/>	
<b>れ</b>	
列挙.....	16, 20, 21
宣言.....	21
列挙定数.....	24
宣言.....	24
<hr/>	
<b>ろ</b>	
ローカルクラス.....	20, 31
宣言.....	20, 31
ローカル変数.....	32
宣言.....	32
宣言文.....	32
<hr/>	
<b>わ</b>	
ワイルドカード.....	17, 19
境界.....	18
<hr/>	
<b>漢字</b>	
式	
クラス・インスタンス生成式.....	40