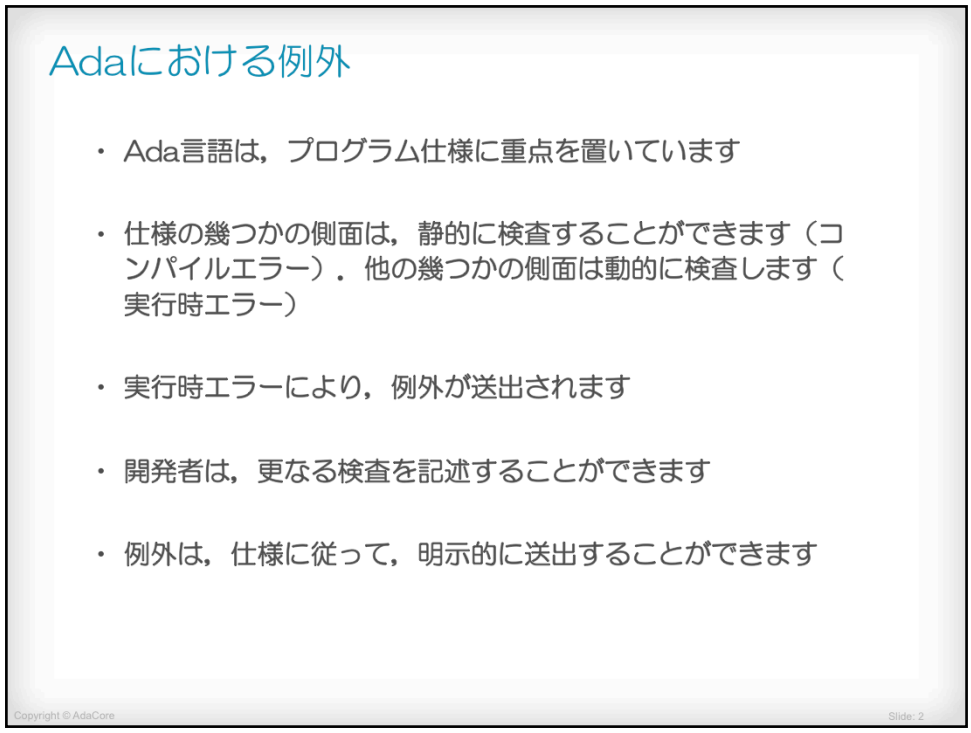
私の名前は，Martyn Pike です．大規模プログラムについての Ada University の講義に，ようこそ．

この講義のテーマは，Ada言語における例外の扱いです．一緒にスライドを終えた後，学習の程度をみるための幾つかのクイズを行います．



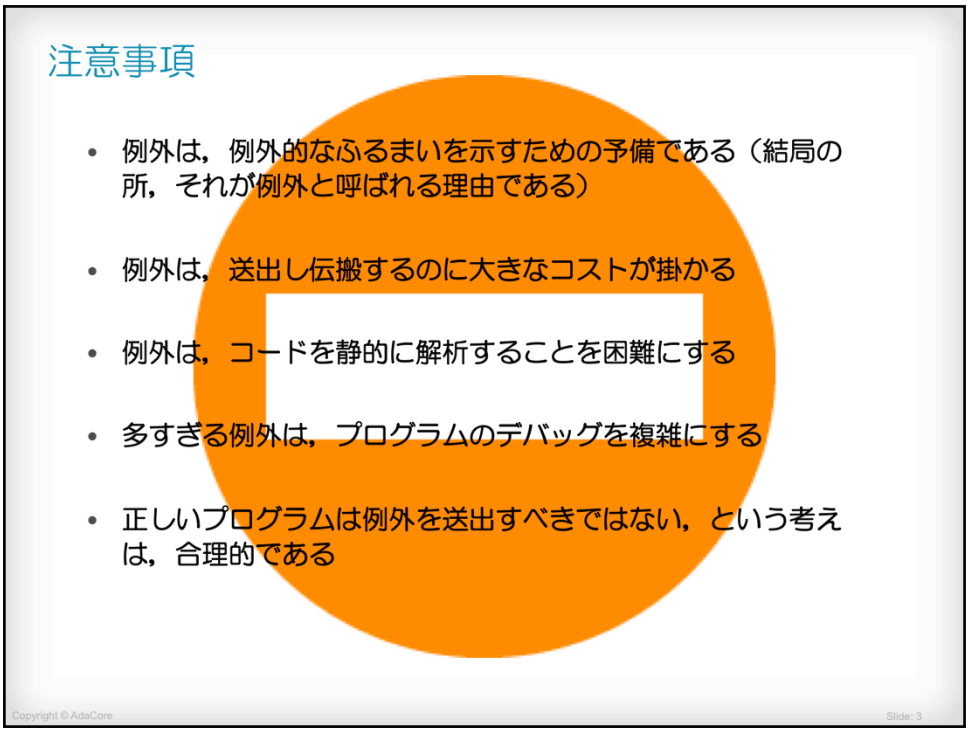
Ada 言語でプログラミングを行うときに鍵となるのは，Ada コンパイラのソースコードに対する向き合い方です．プログラムの実行に関する回路図がそうであるように，コンパイラはソースコードをどのように扱うかです．

この「回路図」は，Ada コンパイラによって完全にチェックすることができます．コンパイラは，要すれば，警告やエラーを出力します．

これは，静的検査と呼ばれます．しかし，Ada　コンパイラは，プログラムにコードを挿入し，結果的に例外を送出する実行時エラーが検出できるようになります．

開発者は，明示的に例外送出が生じる検査を，追加で記述することもできます．

しかしながら，理解すべき大事な点は，実行時エラーは，多くは開発者にとって思いがけないものであるということです．Ada 言語仕様は，これらエラーがプログラムの流れに如何に影響を与えるか関して確固とした規則を持っています．



例外を利用したプログラムをしているときに，注意すべきいくつかの落とし穴があります．

例えば，例外は，通常のプログラムフローで用いられるべきではありません．また，例外的なふるまいを示すことは予備的であるべきです．

落とし穴に落ちることは，通常，「例外を利用したプログラミング」と呼ばれます．

ソースコードへの影響は最小限ですが，実行時性能は，驚くほど大きく影響を受けます．例外を送出し伝搬するために，コンパイラが多くのコードを挿入するからです．

開発者が注意深さに欠けていると，「例外を利用したプログラミング」をしているとは思っていないときですら，スパゲッティコードを作ることは可能です．そして，それは，静的解析が困難で，デバッグすら難しいコードです．

開発者は，ソースコードをプログラムの正しい意図したふるまいとなるようにすることが大事です．
日常的に例外を送出するようなプログラムを書くべきではありません．

Ada プログラム言語は，幾つかの既定義の例外を持ち，コンパイル中および実行時に検査を行います．

最もよくある例外は，CONSTRAINT_ERROR です．これは，主として Ada が強い型づけを持った言語であるという特徴に由来しています．

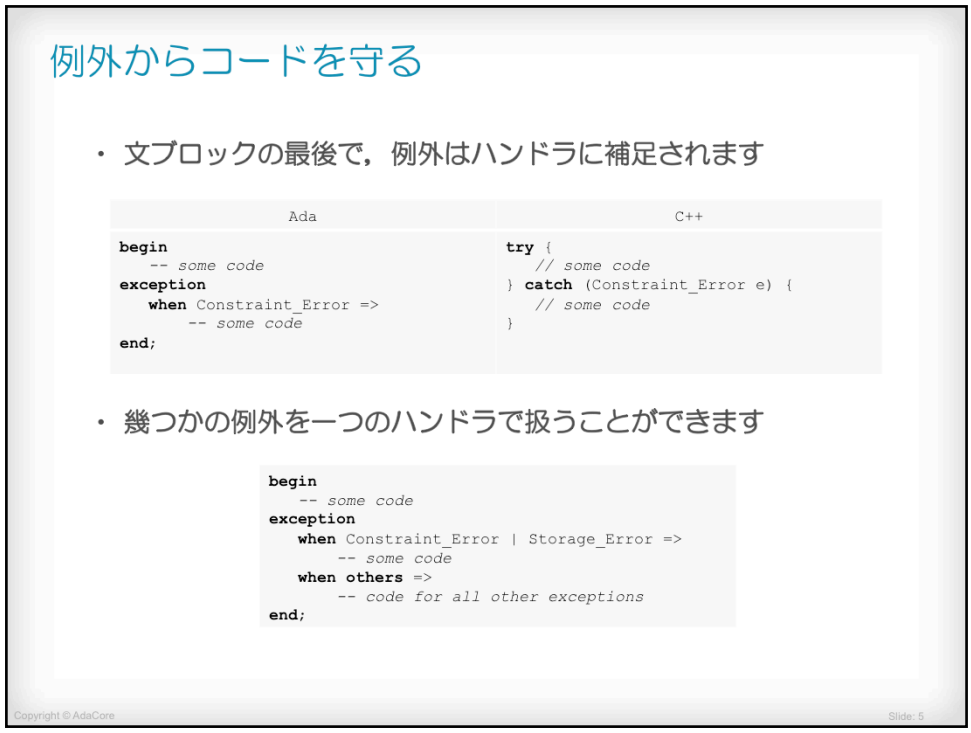
もし，スカラー値が指定した範囲を超えたり，配列添字が範囲を超えると，CONSTRAINT_ERROR が送出されます．

Ada が他の言語と違うのは，ヌルポインタが値参照されたときに例外を送出することができるという点です．これは，ハードウェアや他で開発されたコードとインターフェイスする必要がある組み込みソフトウェアには，計り知れない価値を持った特徴です．

ここに示す例にあるコードの断片では，実行時に CONSTRAINT_ERROR 例外が送出されます．問題は，値 11 が，My_Array 型が受け入れ可能な添字範囲外だということです．

これは，かなり明白なソースコード中のエラーであり，良い Ada コンパイラーであれば，例外が生じるに違いないことを，コンパイル中に開発者に警告します．

もし，開発者がこの警告を無視すると，このコード行が実行されるやいなや例外が送出されます．これは，思いも掛けないふるまいから，他のプログラムフローを守るためです．



開発者は，ソースコード中に例外ハンドラを書くことができます．また，ハンドラには，例外が送出されたときに実行するコードを含んでいます．

例外ハンドラは，命令文ブロックの最後に置かれます．「exception」 キーワードで開始し，case 文と同様の when 文が続きます．

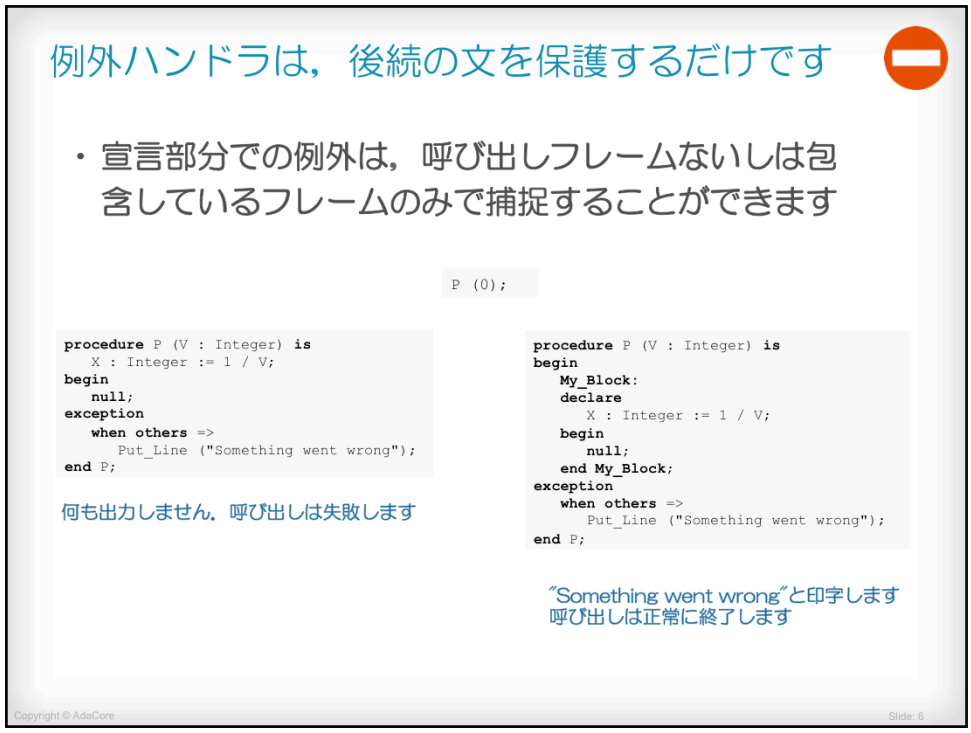
各 when 文は，一つ以上の名前付き例外を補足します．或いは 「when others」 によって，全ての（残りの）例外の補足を行います．

ここにある例では，Ada の例外ハンドラと， C++ の try ブロックを比較しています．この例のように，現在では，例外処理は，Ada 特有というわけではありません．しかし，Ada は，例外をサポートする最初の言語の一つでした．

二番目の例は，少し複雑な例外ハンドラの例です．もし，Constraint_Error ないしは Storage_Error が送出された場合に，コードを実行します（-- some code とある場所）．他の型の例外が送出された場合には，異なるコードを実行します（-- code for all other exceptionsの場所）

例外ハンドラには，通常エラーメッセージのログを含みます．また，残りのプログラムフローが予測可能となるように定まった値を持つ変数を返します．

例外ハンドラ内部のコード自身が例外を送出しないように，注意が必要です．こういったことが生じるかどうかを静的に検出するのは困難です．従って，可能な限り例外ハンドラを単純にすることで，より安全になります．



例外は，ブロック中の一連の文を保護するのみである，ということを覚えておくことは重要です．

例における CONSTRAINT_ERROR　は，主として記述した型モデルに対する違反に関係しています．従って，これら例外は，ブロックの宣言部内で生じます．

全ての記述した例外ハンドラは，begin と end の間にある文のブロックの中で送出された例外のみを扱うことができます．

宣言部で送出された例外は，包含しているフレーム内にあるハンドラに渡されます．

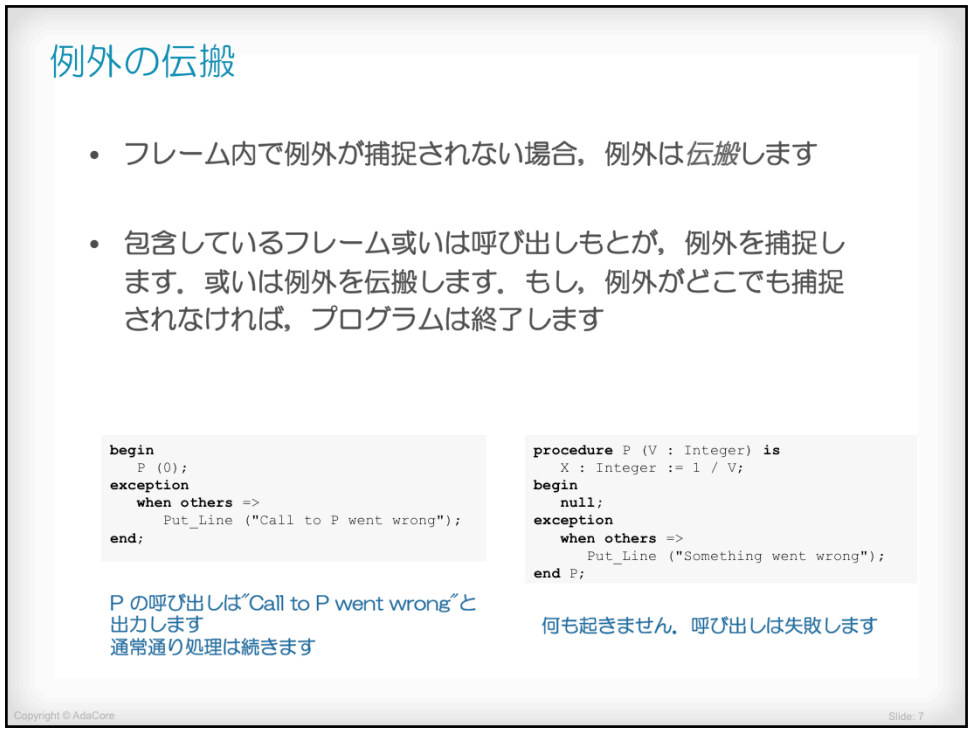
例えば，整数パラメータ V に対して引数ゼロを与えて，手続き P を呼び出してみます．

左側にある最初の P の実現では，実行時に宣言部で CONSTRAINT_ERROR が送出されます．X は，1/V であり，いま V は 0で初期化されるからです．

この例外を扱うために，ハンドラは用いられません．なぜならば，宣言部で例外が生じており，その例外は包含するフレームの制御下にあるからです．

右側の実現では，入れ子になったブロック中に誤りを含んだ宣言部があります．例外送出で，P 中の例外ハンドラがこの例外を扱います．即ち，” Something went wrong（何かがおかしい）”と出力します．

これは，例外をデバッグするときに覚えておくべき重要なことです．宣言部で例外が送出されたときに，例外の原因箇所に関して勘違いしやすいからです．



例外について話をするときに，共に考えるべき言葉があります．それは，例外伝搬によって伝搬するということです．

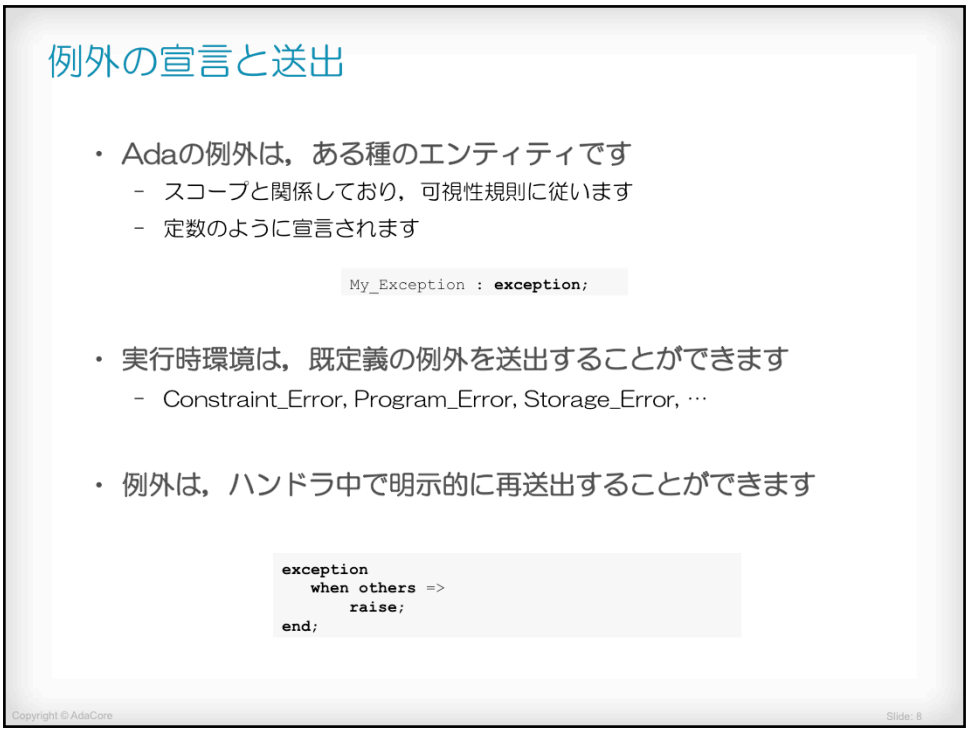
例外ハンドラの記述のないフレーム内で例外が送出された場合，例外は，包含しているフレームに伝搬します．

伝搬は，捕捉可能なハンドラがその例外を捕捉するまで，包含するフレームを伝わります．もし，もうフレームがなければ，最終的にプログラムは終了します．

この例では，引数 0 で P を呼び出した場合の完全な状況を示しています．

P は，例外ハンドラを持っています．しかし，P の宣言部における例外は，包含している外側のフレームに伝搬される，ということを既に学びました．従って， P における例外ハンドラは，スコープ中にはなく， CONSTRAINT_ERROR 例外を捕捉することができません．

しかし，包含しているフレームに例外ハンドラがあります．それは全ての例外を捕捉するように書かれており，Text_IOパッケージの手続き Put_line を引数 ”Call to P went wrong（Pの呼び出しに誤りがある）” とともに呼び出します．



既定義の例外名をコード中で再利用することは可能です．しかし，混乱を避けるために，Ada では，新しい例外を，ある種のエンティティとして宣言できます．

最初に示した例は，My_Exception という名前の例外の宣言です．定数のように宣言され，関連するスコープを持ち，通常の可視性規則に従います．

全てのユーザ定義の例外に関して，開発者には，明示的に送出文を発行する責任があります．コンパイラは，例外が送出される状況下でも，検査を行うためのコードの埋め込みを行わないからです．

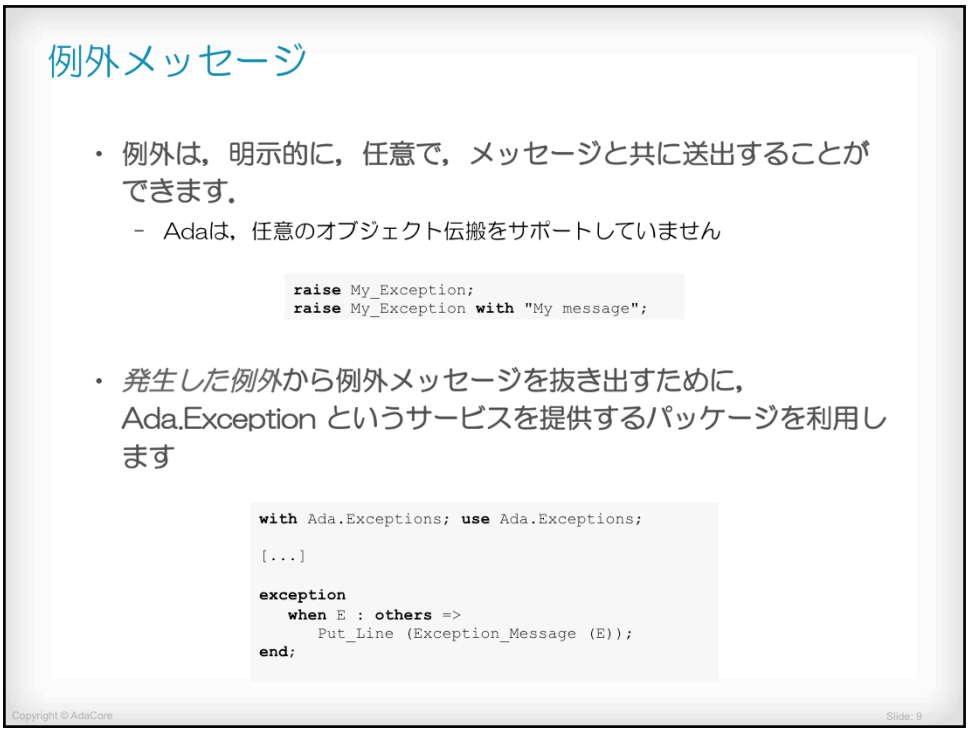
外部の利用者に対して，パッケージ仕様を提供するときに，ユーザ定義の例外を宣言し，どのような状況下で，それらが送出されるかを文書化することはよく行われます．

ユーザ定義例外に関してやり過ぎないように，注意が必要です．過剰に利用すると，初期に学んだように，ひどい落とし穴に陥ることになるからです．

例外を送出し，パッケージの利用者が包含するスコープ中に必要なハンドラを導入するするといったことに依存するより，ステータスフラグや返り値をもった API を設計することの方がはるかに良いやり方です．

送出（raise）キーワードは，例外ハンドラ中においても，発生した例外を，更に外側のハンドラに対して，更に伝搬するために用いられます．

もし，例外に関するエラーを直ぐに記録し，その後，例外を外側のスコープに再度送出す



前のスライドで，良く用いられるパターンについて説明しました．例外送出されたときに，最初にエラーメッセージを記録するというパターンです．送出キーワードには 2 つの異なる利用方法があります．

例外キーワードと例外エンティティの名前を送出することは可能で，既定義の例外においても同様です．別の方法として，例外エンティティ名の後にメッセージ文字列を出力することも可能です．

Ada 言語は，任意のオブジェクトを伴った例外の伝搬は支援していません．可能なのは，メッセージ文字列のみです．

Ada には，Ada.Exceptions という特別なパッケージがあります．このパッケージは，開発者が，例外発生に関する情報を得るための有益なサービスを提供しています．

最後のコードの断片中で，例外ハンドラが何らかの例外発生を捕捉し，E として識別したとします．E を一種のパラメータのように扱い，Ada.Exception の Exception_Message 関数を呼び出します．

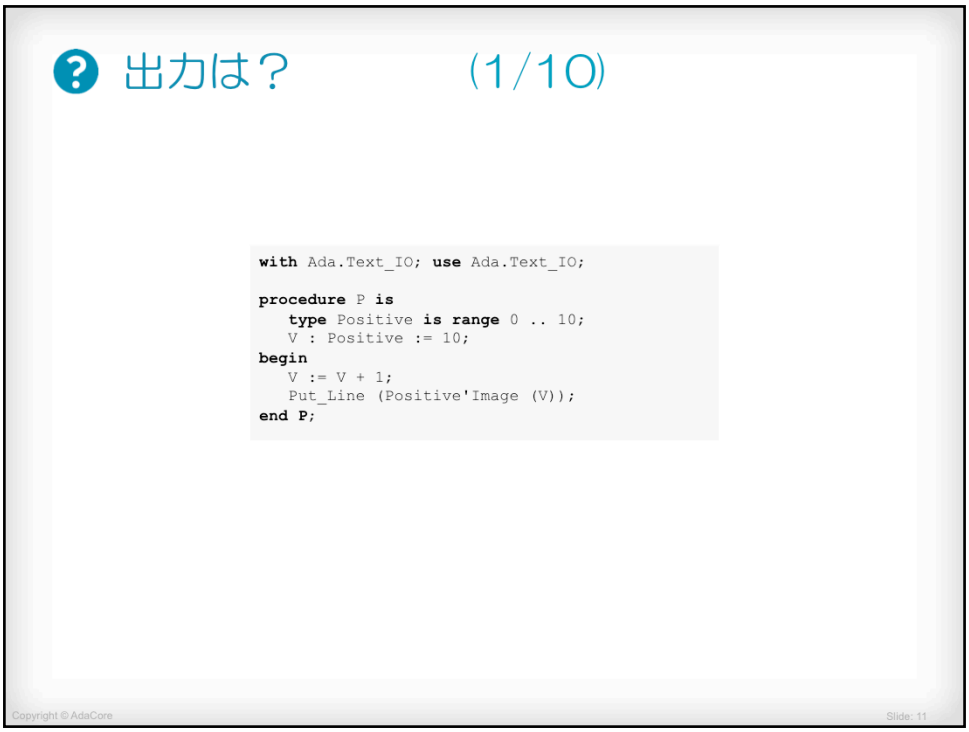
この関数は，例外メッセージが呼ばれたときに，もし例外エンティティ名が，ともに提供されていれば，そのメッセージ文字列を返します．

この講義の最後になりました.

既に Ada の例外について十分な知識を持っていると思います．あなたの理解を確認するように設計された質問からなるちょっとしたクイズをやってみましょう.

各問題に（正解・不正解の）印をつけると，講義の最後で自分のスコアを確認することができます.

幸運を！



最初の質問は，P と呼ぶ手続きから始めます．Ada.Text_IO パッケージが利用可能になっています．

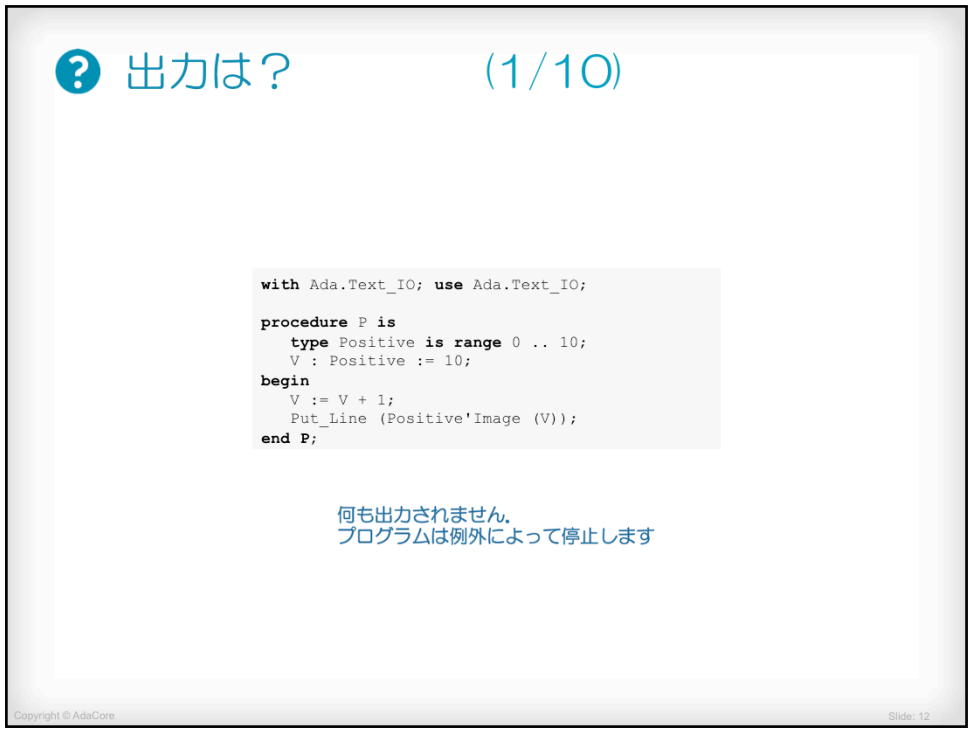
手続き P は，宣言部を持ち，Positive という名前の新しい数値型を含んでいます．Positive の範囲は，0 から 10 です．

新しい Positive 型のスタック領域オブジェクトが作られ，初期化として，値 10 を設定します．

手続きのボディ部では，直ぐに V の値を 1 増やします．次に，Ada.Text_IO の Putl_Line 手続きを呼び出します．

Positive 型の’ Image 属性が，スタック領域オブジェクト V に与えられており，数値の文字列表現を返します．

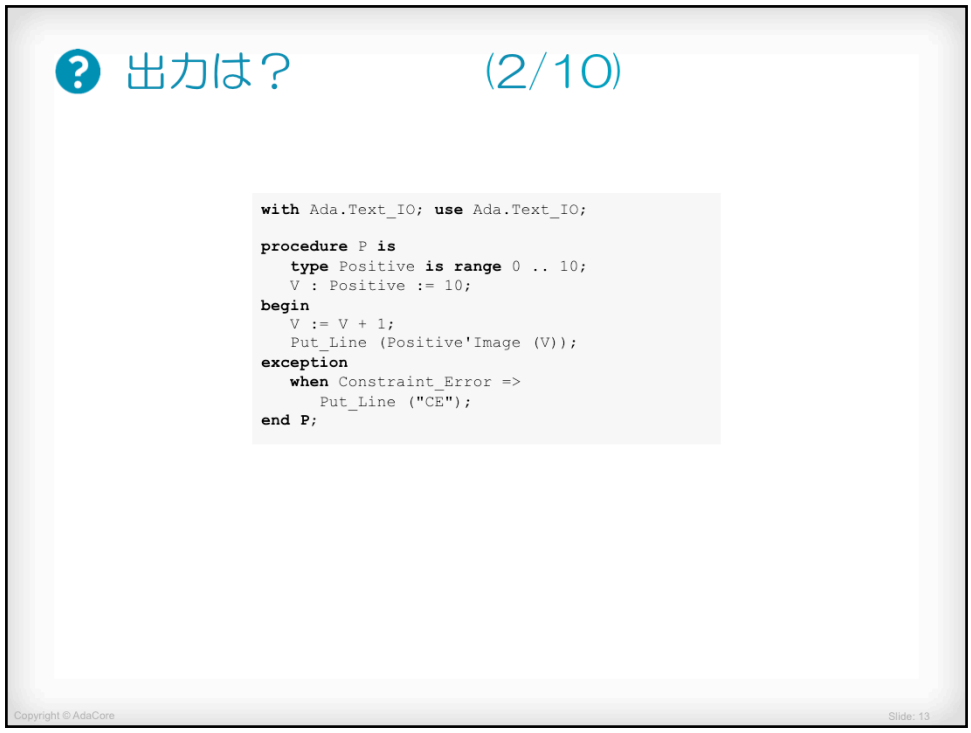
どういう文字列を Put_Line が出力するかが，あなたへの質問です．



答えは，出力なしです．

CONSTRAINT_ERROR 例外の発生によって，プログラムフローは中断します．これは，V の値を，許された範囲を超えて，増加させようとしたことに起因しています．

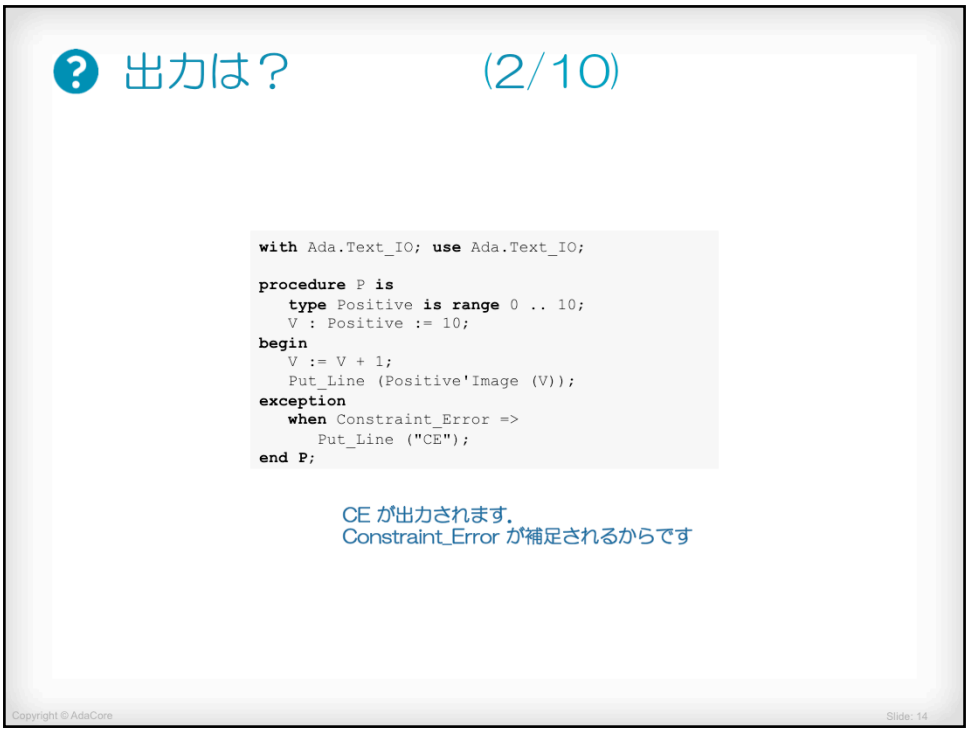
初期化において，既に許される範囲の最大値に値がセットされています．手続きのボディ部で，それを増加させるとエラーとなります．



二番目の問題は，ほぼ前回の問題と同じコードを使用しています．しかし，重要な違いがあります．

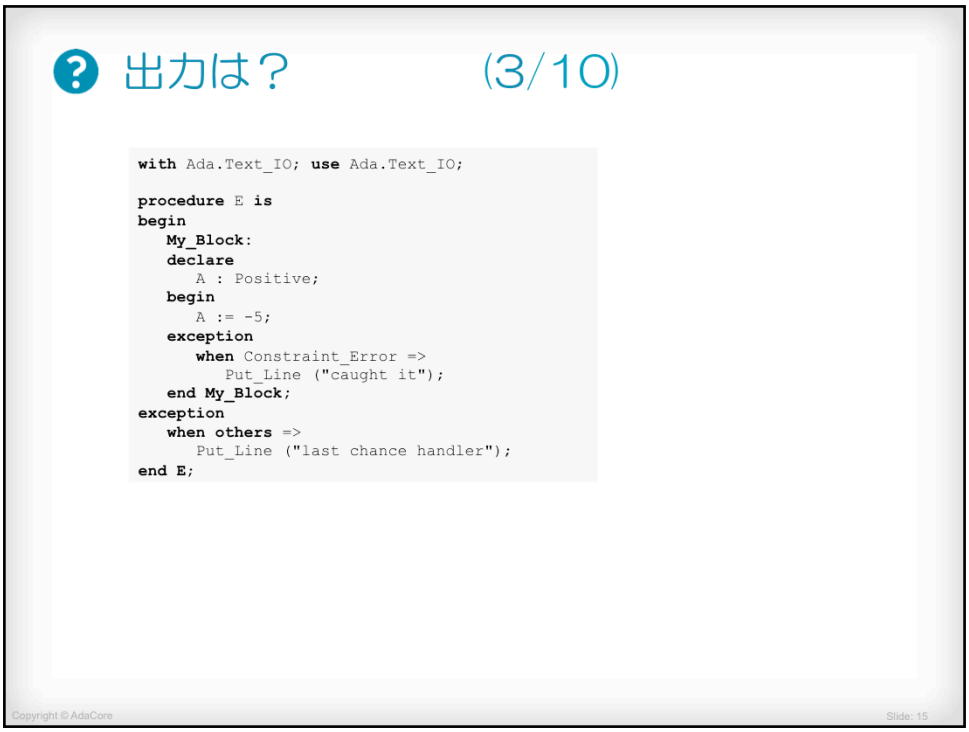
手続き P のボディ部内で，文のブロックに対して，例外ハンドラが提供されているところが，異なっています．

このコードは，何を出力すると考えますか



文字列”CE"が出力されます

どうしてそうなるかは理解できることを期待しています．念のために書いておきます．例外ハンドラが，CONSTRAINT_ERROR を捕捉して，文字列 ”CE" とともに Put_Line を呼び出しています．



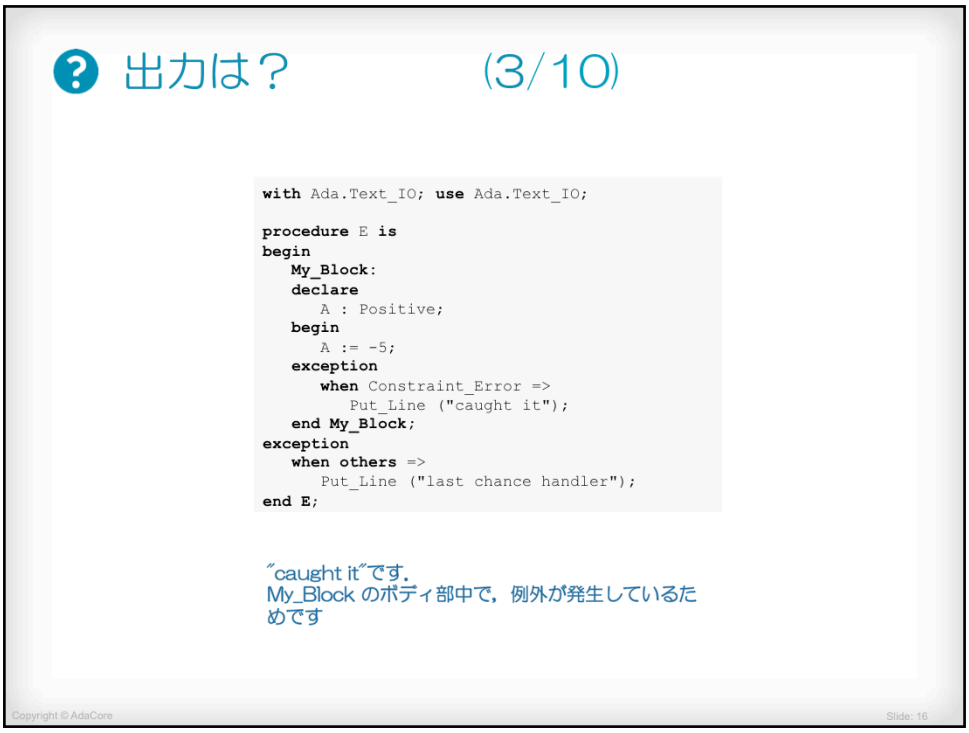
問題 3 は，もう少し複雑な E という手続きに関してです．

E 自身，文のブロックに対してイベントハンドラを持っています．このハンドラは，全ての例外を捕捉し，”last chance handler（最後の機会ハンドラ）" を出力します．

手続き E は，入れ子になったブロック My_Block を持ち，宣言部では，Positive 型のスタック領域オブジェクト A を宣言しています．My_Block のボディ部には，たった一つの文があり，もう一つの例外ハンドラを持ちます．

My_Block 中の例外ハンドラは，CONSTRAINT_ERROR を捕捉し，“caught it（つかまえた）” を出力します．

このコードは，何を出力するでしょうか

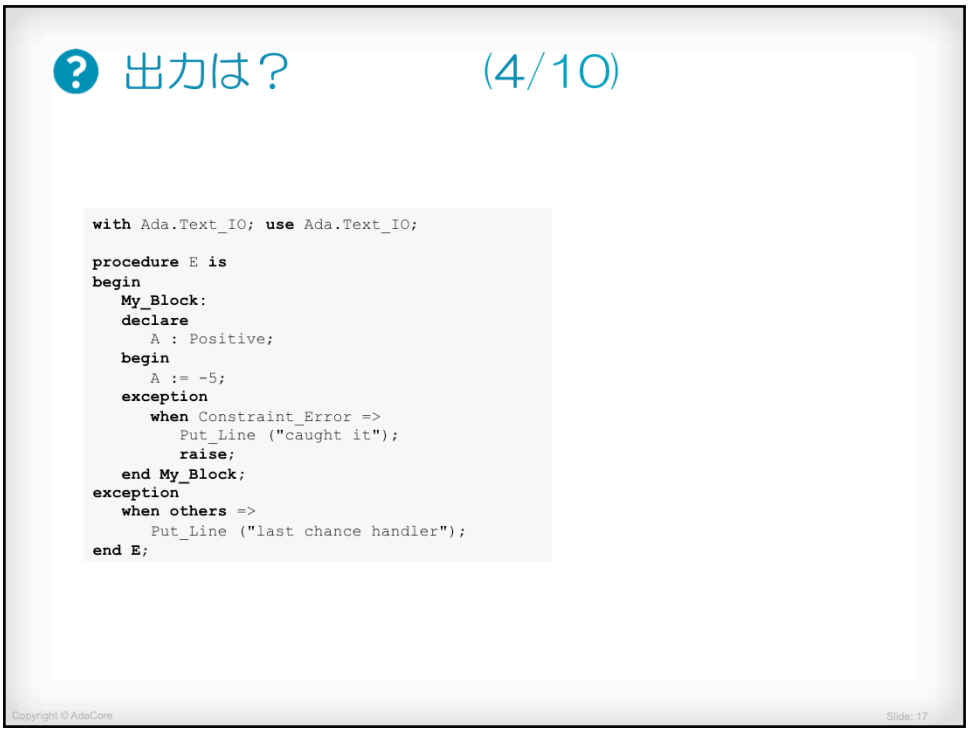


今回，コードは，“caught it” を出力します．My_Block 中の例外ハンドラが生成します．

例外の根本原因は，My_Block 中の文です．Positive 型である A に対して負数を代入したためです．

送出された例外は，外側のスコープのうち最も近い My_Block ハンドラに捕捉されます．結果として，“caught it” を表示します．

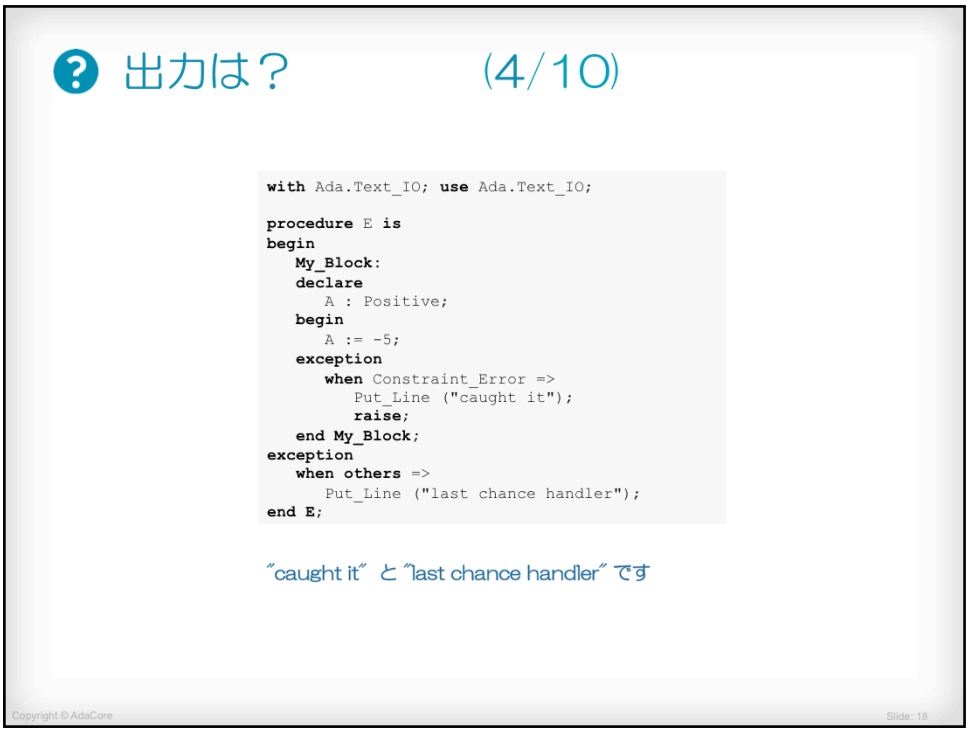
更なるアクションはありません．手続き E の残りの文が実行されます．



前回の例を再利用します．My_Block 中で宣言されている例外ハンドラにちょっとした変更を加えます．

文字列 “caught it” の出力のあとに，raise 文があります．

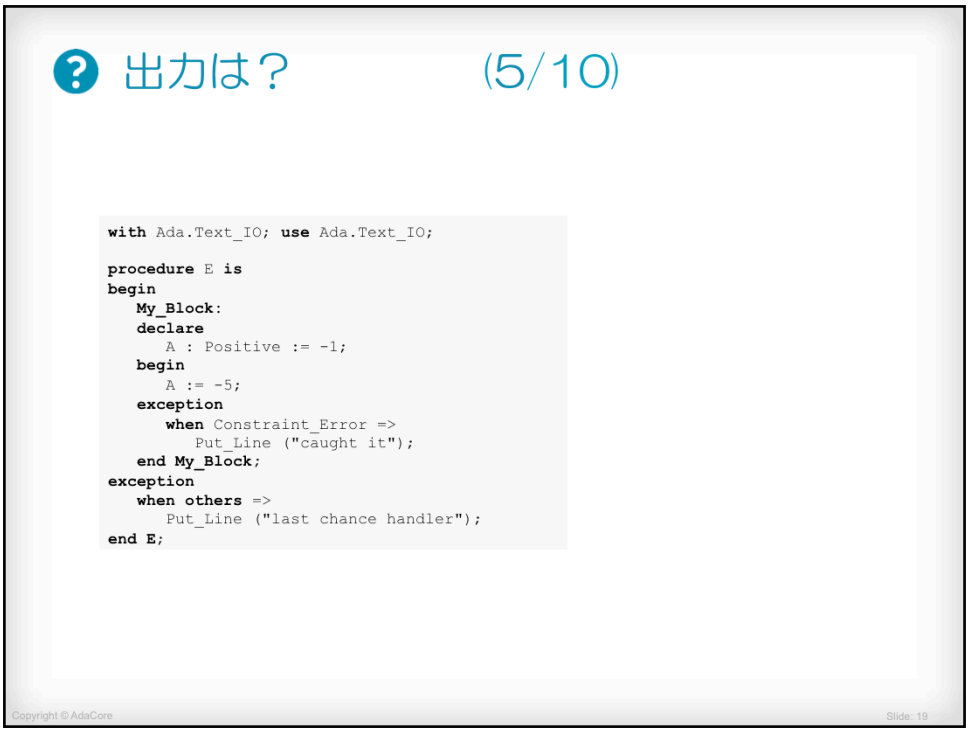
この修正したコードでは，どういう出力があると考えますか．



出力される値は，最初に “caught it” で，次に “last chance handler” になります．

My_Block 中で定義されている例外ハンドラの最後に， "raise" 文があるため，この結果となります．

この文は，外側のブロックにあるハンドラに，例外を更に伝搬するという効果があります．そして，外側のブロックには，手続き E 中で定義されたイベントハンドラがあります．

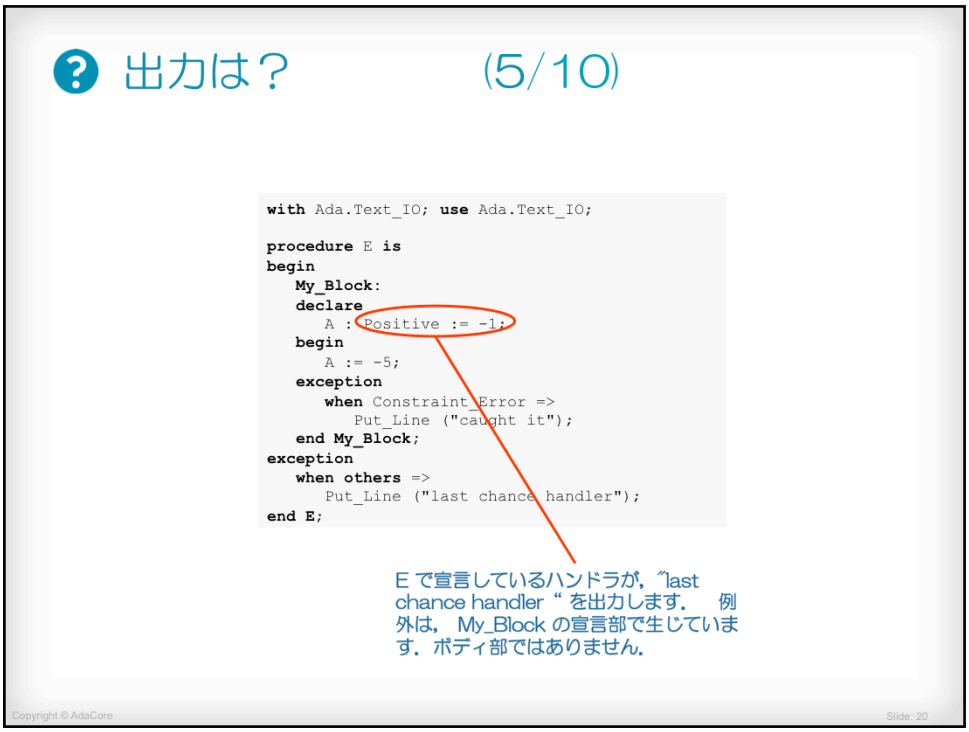


さらに同じコードの断片を用います．ただし，今回，My_Block の宣言部分を変更しています．

A を宣言し，初期値を -1 とします．

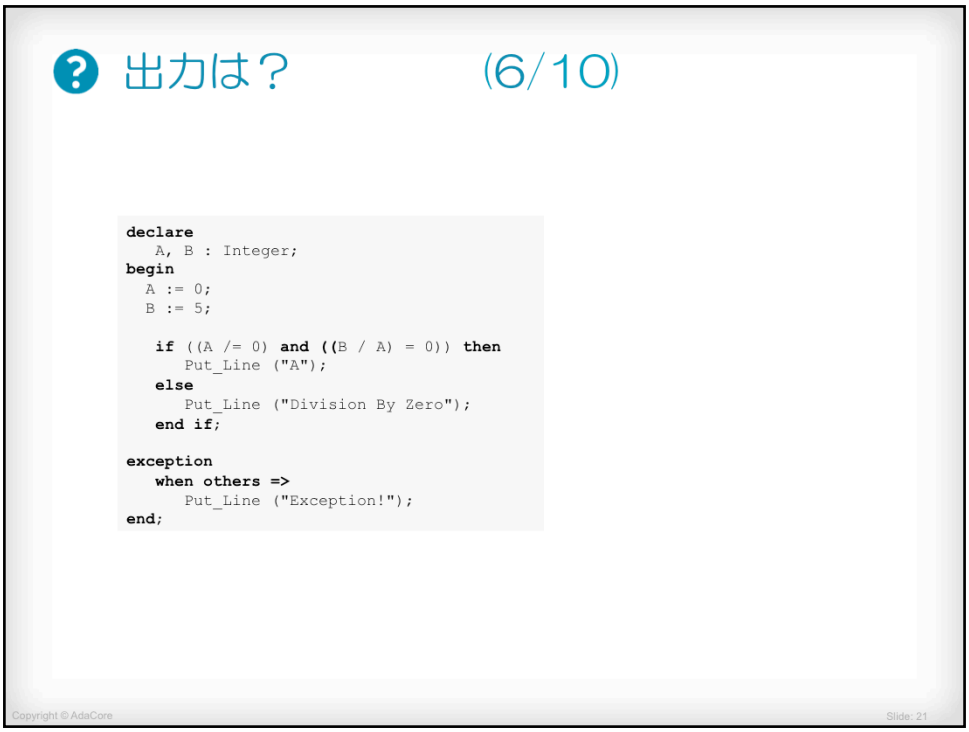
My_Blockで宣言された例外ハンドラから raise 文が取り除かれたことに注意して下さい．

何が出力されるとお考えですか．



変更によって，My_Block の宣言部分において，不正な初期値が A に代入されたことで，CONSTRAINT_ERROR が生じるということは分かると思います．

My_Block のボディ部で定義されている例外ハンドラは，スコープ中にありません．それで，外側のスコープ中のハンドラに伝搬されます．手続き E は，例外ハンドラを宣言しているので，“last chance handler” を出力します．



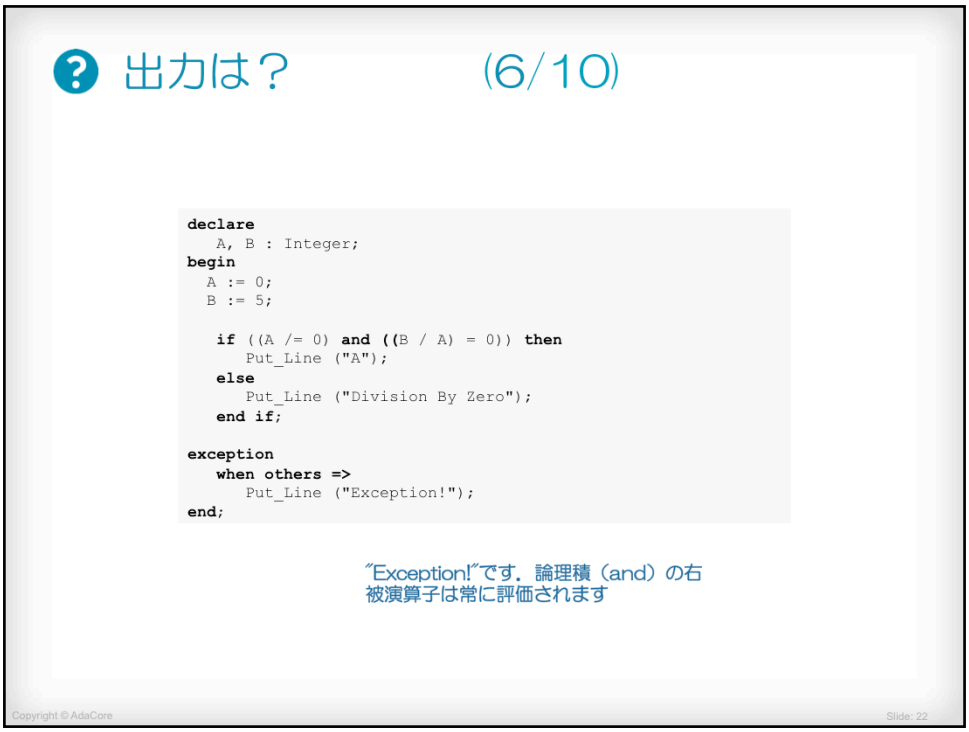
問題 6 では，代入文ではなく，条件式で送出される例外についての知識を問います．

少し（これまでとは）違ったコードの例を用います．ここでは，全ての例外を受け取り ”Exception!（例外!）” と出力する例外ハンドラを持つブロックを定義しています．

このブロックの宣言部では，2 つのスタック領域オブジェクトを宣言していて，それぞれ，ボディ部で 0 と 5 を代入しています．

次に条件式があり，もし，条件式が真であれば，文字 A を出力します．そうではない場合，”Division by Zero（ゼロ割り）” を出力します．

コードのふるまいをよく考えて，何が出力されるかを決めて下さい．



文字列，”Exception!" が出力されるというのが正解です．

条件式を見ると，評価される右被演算子が，B/Aを含んでいることに気がつくと思います．この場合，5 を 0 で割ることになるので，CONSTRAINT_ERROR が発生します．

この例題は，代入文と同様に条件式においても例外送出が可能であることを示しています．

（実行時ではなく）コンパイル中にお使いのコンパイラがこの例外に関して警告を出すと良いのですが．

次は，ユーザが定義した例外に対する Ada 言語のスコープと可視性に関する知識を問う問題になります．

この例題では，パッケージ P を定義しています．この中で，Call という名前を持つ手続きインターフェイスがあります．

P のパッケージボディ部中には，Call の実現があり，My_Exception という名前のユーザ定義の例外を明示的に送出します．

Main の手続きは，with 節とともに P を指定し，P が提供するインターフェイスを使用します．また，My_Excepiton 例外に対する例外ハンドラを定義しています．

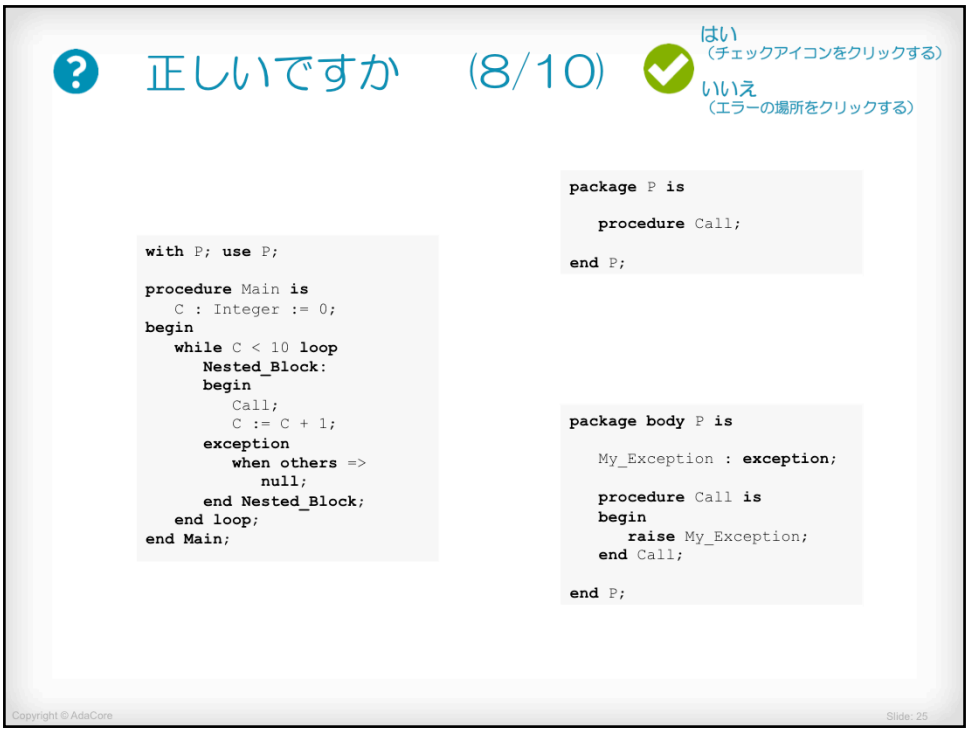
このコードは正しいでしょうか．

このコードが正しいと思うときは，「はい」のアイコンをクリックしてください．間違いと思えば，その場所をクリックしてください．

このコードは，正しくありません．Main 手続きは，コンパイルできません．My_Exception は，Main からは不可視です．なぜならば，P のパッケージボディ部で定義されているからです．

このコードをコンパイルするためには，My_Exception の宣言を，Pのパッケージ仕様部に移動すべきです．

この例は，ユーザ定義の例外は，他の全てのユーザ定義 Ada エンティティと同様に，名前付けと可視性の規則に従わないといけないということを示しています．



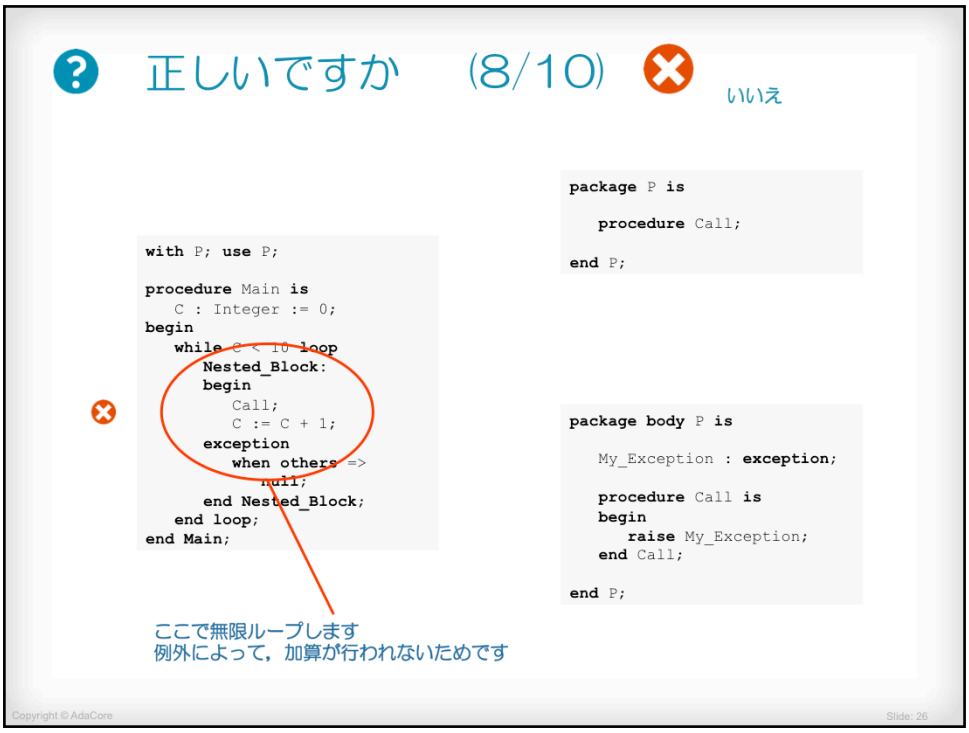
問題 8 は，Call という名前の手続きを含む P のパッケージ仕様から始めます．

P のパッケージボディは，My_Exception というユーザ定義の例外を持ち，Call のボディ部の唯一の命令文は，My_Exception の送出です．

手続き Main は，スタック領域オブジェクト C を宣言しており，ループインデクスとして使用しています．ループは，入れ子のブロックを含んでおり，何もしない例外ハンドラを含んでいます．

ブロックのボディ部は，パッケージ P が持つ手続き Call の呼び出しを含んでいます．また，ループインデクス C を 1 増加させます．

このコードが正しいと思うときは，「はい」のアイコンをクリックしてください．間違いと思えば，その場所をクリックしてください．



このコードは正しくありません．

Call によって明示的に送出された例外は，Nested_Block 内で定義されたハンドラに伝搬されます．プログラムフローは，C を 1 増加する部分を回避するので，ループ条件式は，常に真となります．結果として，ループは決して終了せず，無限ループとなります．

これは，コードインスペクションを行ったとしても，例外を含む実行時誤りを特定することが，如何に難しいかを示しています．例外伝搬と例外ハンドラの中身によっては，プログラムフローは，きわめて危うくなります．

次は，例外ハンドラ自身の中に記述されたコードのタイプについての理解を確認する問題です．

例外の送出は，正常なプログラムフローから，例外ハンドラにとぶので，見えない goto 文と同様だと云うことを思い出して下さい．

ハンドラは，自分自身にジャンプしてきたときコンテキストを十分に把握し，正しくふるまう必要があります．

ここでは，整数の範囲と整数の要素からなる A_Type という配列型を宣言しているパッケージ P を示しています．

Safe_Get という関数があり，配列型のオブジェクトと添字を受け取り，整数要素を返却します．

パッケージボディ部では， Safe_Get の実現中に例外ハンドラがあります．

このコードが正しいと思うときは，「はい」のアイコンをクリックしてください．間違いと思えば，その場所をクリックしてください．

このコードは間違っている，と考えたならば，正解です．

Safe_Get のボディ部は，返り値を持っていますが，局所的なスコープにある例外ハンドラには，返り値がありません．もし，Safe_Get のボディ部内で例外が発生したら，局所的なスコープを持つハンドラに伝搬します．結果として（返り値がないので） PROGRAM_ERROR　を実行時に発生させます．

コンパイル中に，お使いのコンパイラがこの例外に関して警告を出すと良いのですが．

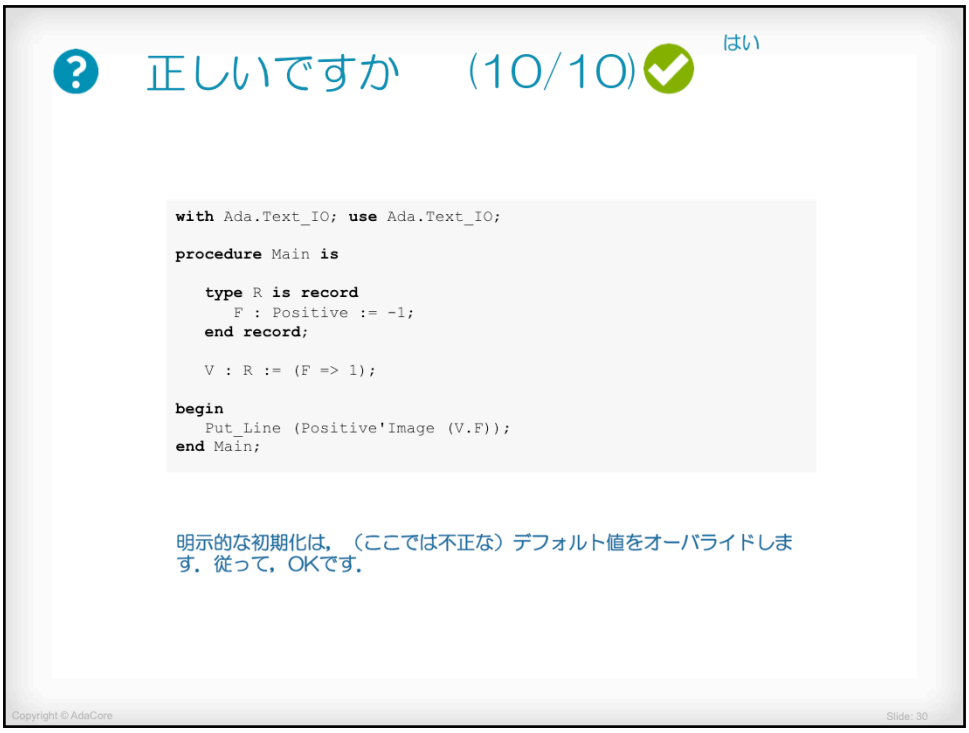
このクイズの最後の問題となりました．少し扱いにくい問題です．

手続き Main 中で，R という新しいレコード型を定義しています．このレコード型は単一の F という Positive 型要素を持ち，デフォルトで -1 の値を持ちます．

レコード型のスタック領域オブジェクトが　V　として作られ，集成代入式を用いて，要素 F に 1 をセットしています．

Ada.Text_IO パッケージから，Put_Line を呼び出し，スタック領域オブジェクト V の要素 F の’ Image 属性を使います．

このコードが正しいと思うときは，「はい」のアイコンをクリックしてください．間違いと思えば，その場所をクリックしてください．



扱いにくい問題ですが，あなたが正解と考えたならばうれしく思います．

このコードに問題はありません．R の実体である V において，その要素 F の明示的な初期化は，不正な初期値を重ね書きします．

Ada 大規模プログラムの例外に関するコースに参加いただきありがとうございました.

また，Ada 言語が提供する既定義の例外とは異なった例外についても，如何に送出されるか，また，正しく扱う方法について学びました.

Ada プログラム言語を学ぶ上で，この講義が貴重なーステップとなったこと，Ada University における他のコースも引き続き受けられることを期待しています.

ありがとうございました.