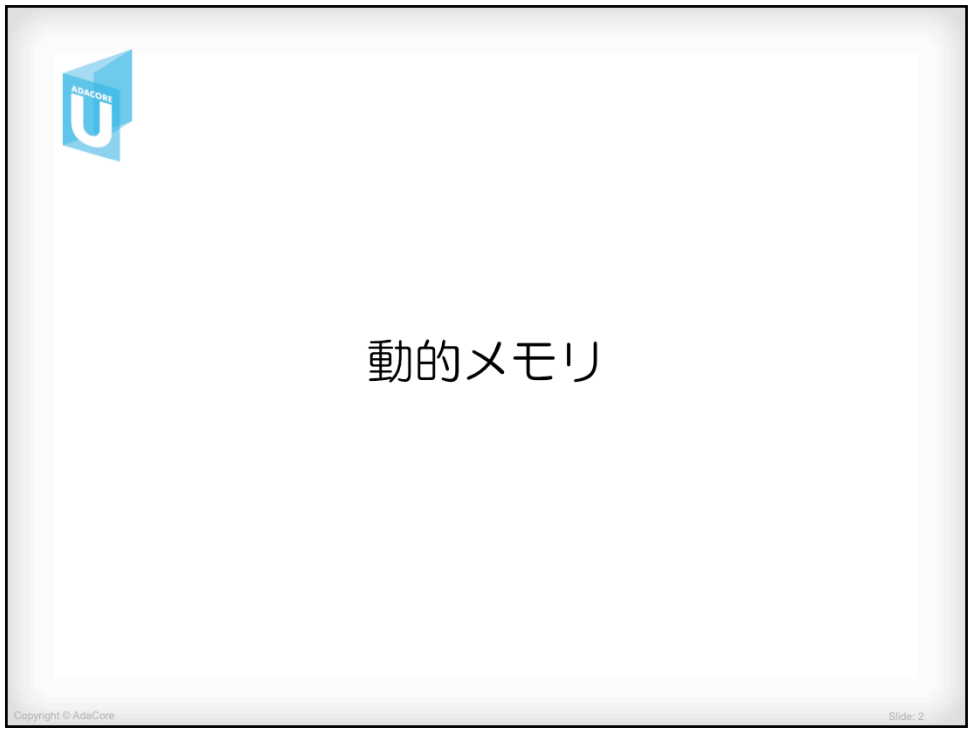
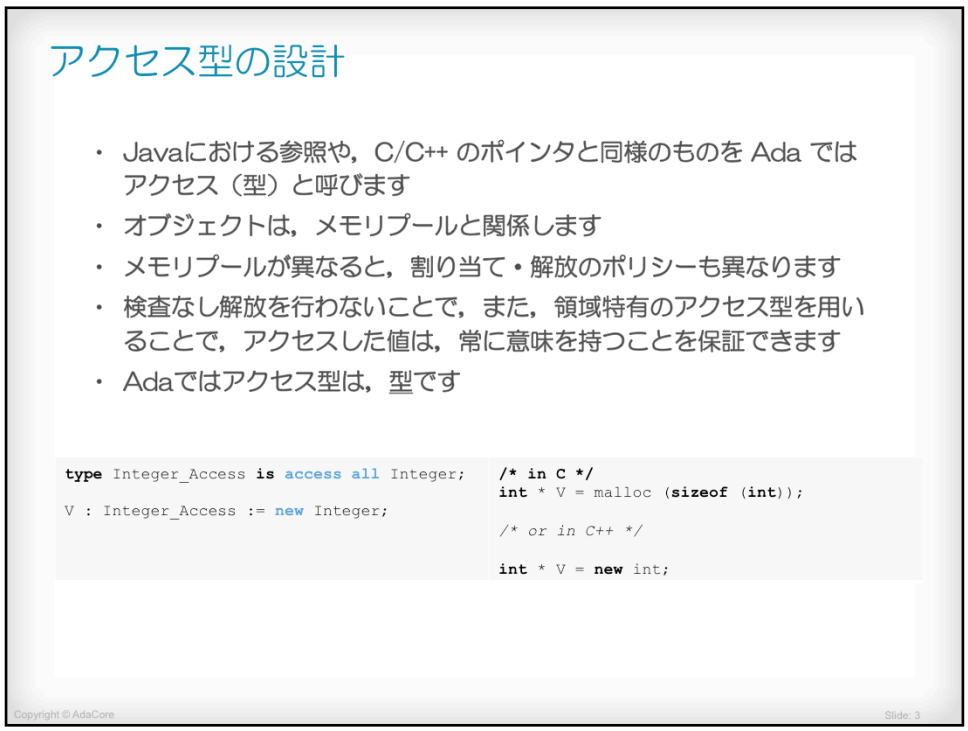
こんにちは．私の名前は，Martyn Pike です．Ada University の大規模プログラミングの講義にようこそ．

この講義のテーマは，Adaのアクセス型です．最初に一連のスライドによる学習を終えた後，幾つかのクイズを通して，学んだことを確認します．

スライドは，三つの部分に分かれています．最初は，メモリを割り当てるときの方法間の違いについて，二番目は，アクセス型の記述法の紹介，最後にアクセス型利用時の規則に従う方法について学びます．



アクセスされるオブジェクトは，実行時にどのようにメモリに割り当てられるかを理解することは，
複雑な Ada のアクセス型を学ぶときに重要です．



Ada のアクセス型は，Java の参照や C／C++ におけるポインタと同じ基本的な特徴を持っています．

記憶域プール中のオブジェクトが参照できること，メモリの割り当て・解放を異なった方法できるように，これらのアクセス型は設計されています．

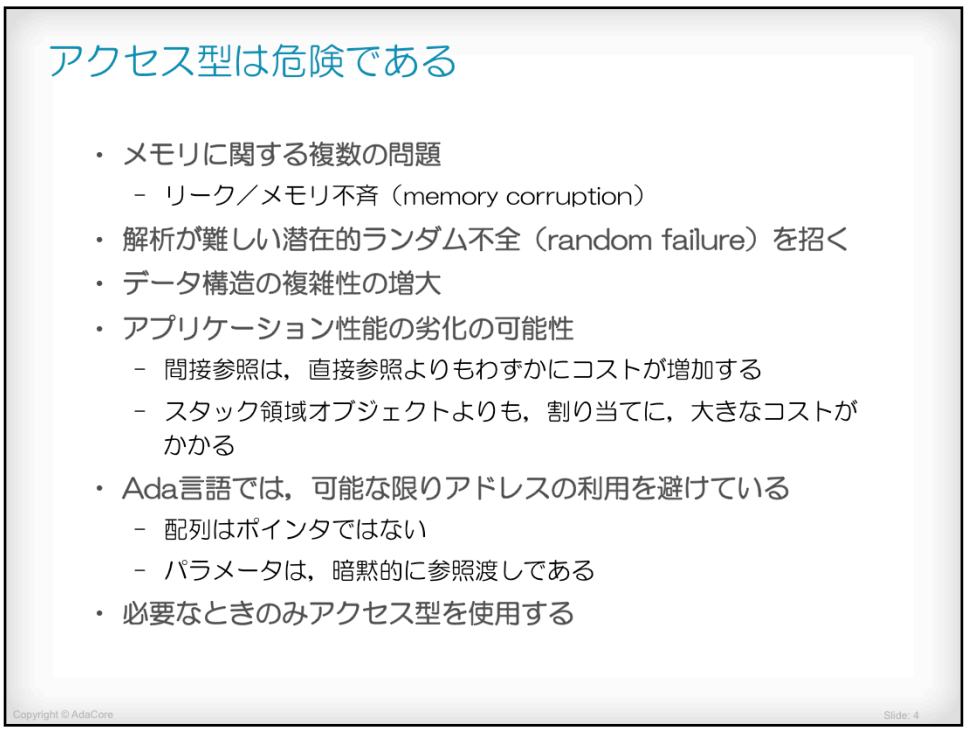
ポインタに共通する問題は，ポインタが示しているものが，常に正しく，かつ意味あるものであるということを保証できないことです．Ada では，この欠点を補うように，アクセス型を設計されています．

個々のアクセスが，異なったメモリ領域に及ばないことを保証するように，慎重に型付けしているというのが，Ada のアクセス型の主たる特徴です．

例にある左のコードでは，アクセス型宣言は，整数型オブジェクトのみを指し示すことができると宣言しています

次に，アクセス型のスタック領域オブジェクトの宣言です．ここでは，new キーワードを用いて，明示的な初期化として，オブジェクトをメモリ領域に割り当てています．

右側の，C 言語版では， malloc 関数を用いて，C++版 では new 演算子を用いて，同等の機能を実現しています．



Adaにおけるアクセス型のアプローチは，伝統的なポインタの問題を解決しますが，問題がなにもないというわけではありません．

様々な対処をしても，それでもやはり，プログラマがメモリリークやメモリ不斉を招くことがあります．

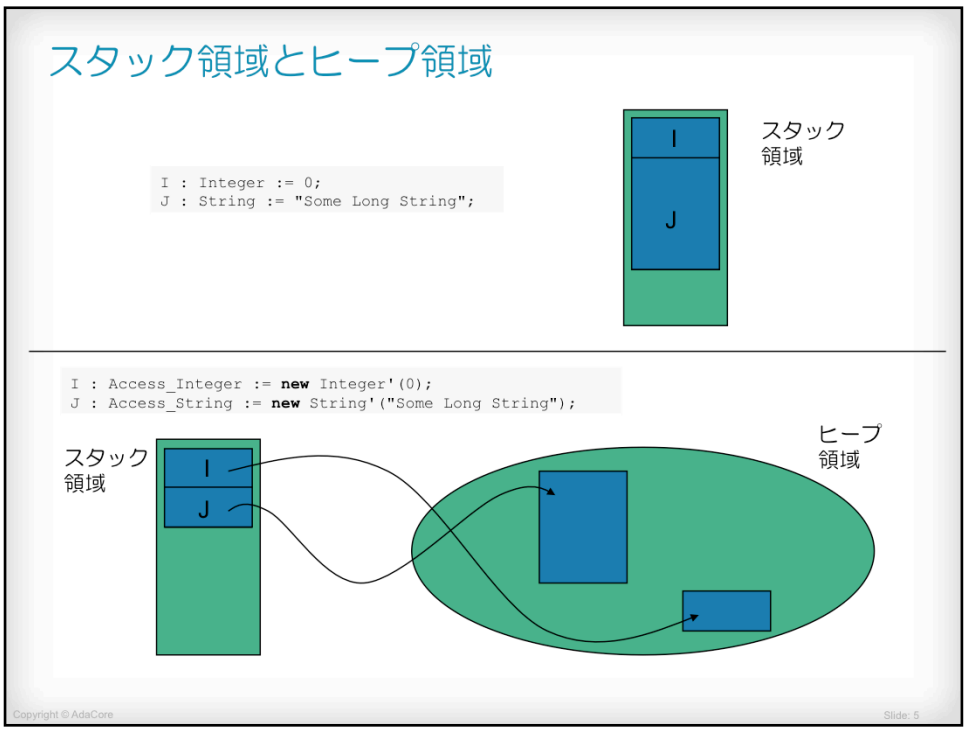
（ポインタにある）動的な特性のために，欠陥の分析やデバッグが難しくなります．また，ランダムなふるまいのように見える場合があります．

アクセス型を通して，オブジェクトにアクセスする場合，2 段階の手続きを経るため性能に影響がでる場合があります．このことは，直接アクセスよりわずかにコストが掛かることを示しています．

もちろん，実行時の動的なメモリ割り当ては，スタック領域上のオブジェクトを用いるよりも，より多くのコストが掛かります．これはメモリプールの管理とメモリの断片化といった問題に起因しています．

総じて Ada 言語ではメモリアドレスの使用を避けています．C 言語と異なり，配列はポインタではなく，パラメータは参照渡しとなります．

ポインタにまつわる先に挙げた危険性の幾つかは，とても深刻なものです．従って，いつもアクセス型を用いるのなら，必ず注意深く考えるようにする，どうしても必要だというとき以外は使用しないようにすべきです．



一般的なポインタの落とし穴を完全に理解するために，実行時のメモリ割り当てに関する基本的な違いを，再度確認することは有益です．

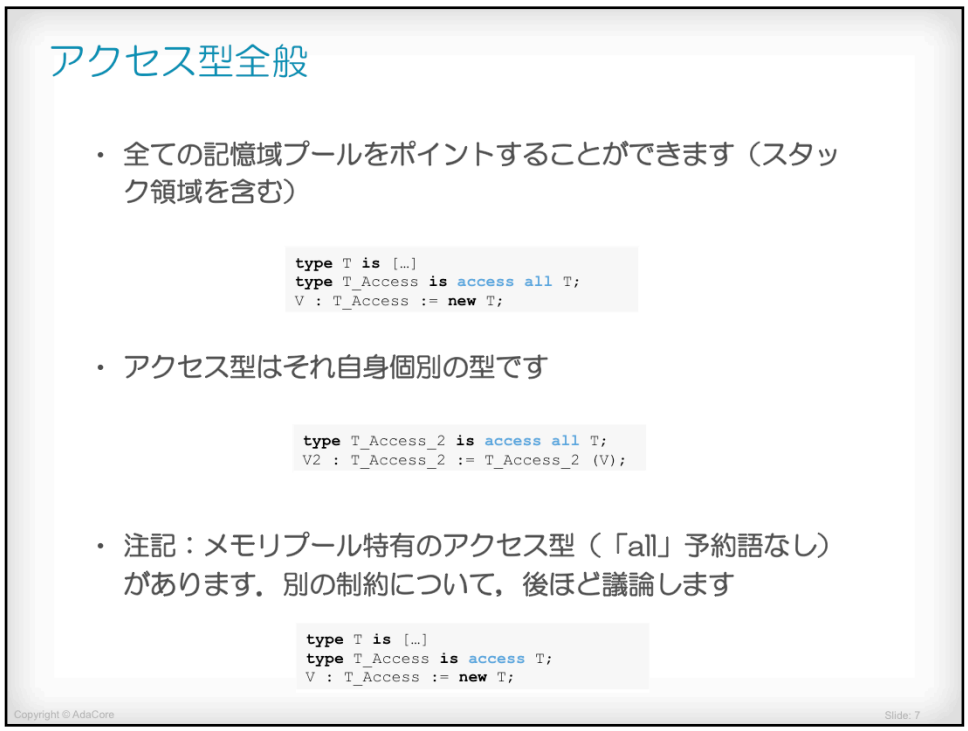
伝統的に，2 つのメカニズムがあります．最初はスタック領域で，二つ目はヒープ領域です．

Ada では，宣言部分に現れる全てのオブジェクトのためにスタック領域を使用します．最初の例にある I , J は，スタック領域において連続する場所を占めています．

二番目の例では，アクセス型のオブジェクト自身は，スタック領域中にいますが，そのオブジェクトを通してアクセスする実際のオブジェクトは，ヒープ領域のどこかにいることを示しています．

少し手を休めて，この図のイメージを心に刻んで下さい．アクセス型について考える時，このイメージをどれだけ強調しても，強調しすぎることはありません．

次に，Ada アクセス型の仕様に関する基礎的な設計原則と危険性のいくつかについて，見ていきます．



さて，ここでは，アクセス型の構文を知り，その意味について考えます．

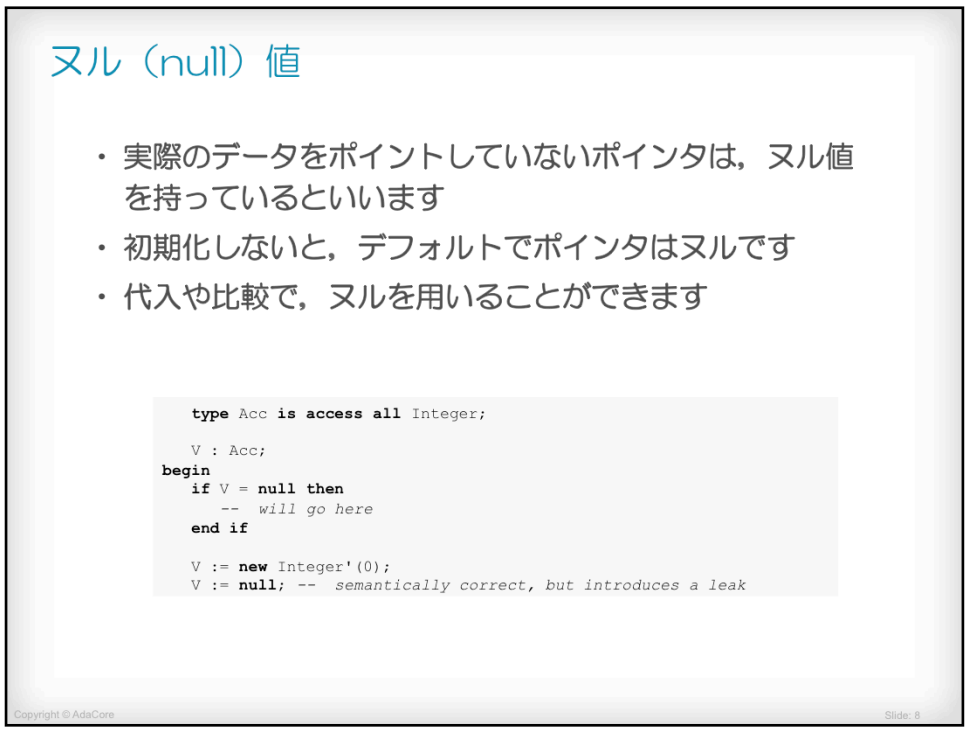
次の基本的な点から始めます．アクセス型は，個別の Ada の型であり，従って，名前・スコープ・属性を持ちます．

構文としては，「access all」キーワードを用い，次に，アクセスされるオブジェクトの型を示します．

例にあるコードの断片に示すように，T_Access や T_Access2 は T 型のオブジェクトにアクセスします，この両者は，Adaの型規則に適合する個別の型となります．

宣言から 「all」 キーワードを除いているメモリ領域特有のアクセス型宣言があります．「all」 を含まない場合に関しては，のちほど説明します．

最後のコードの断片は，その例になります．



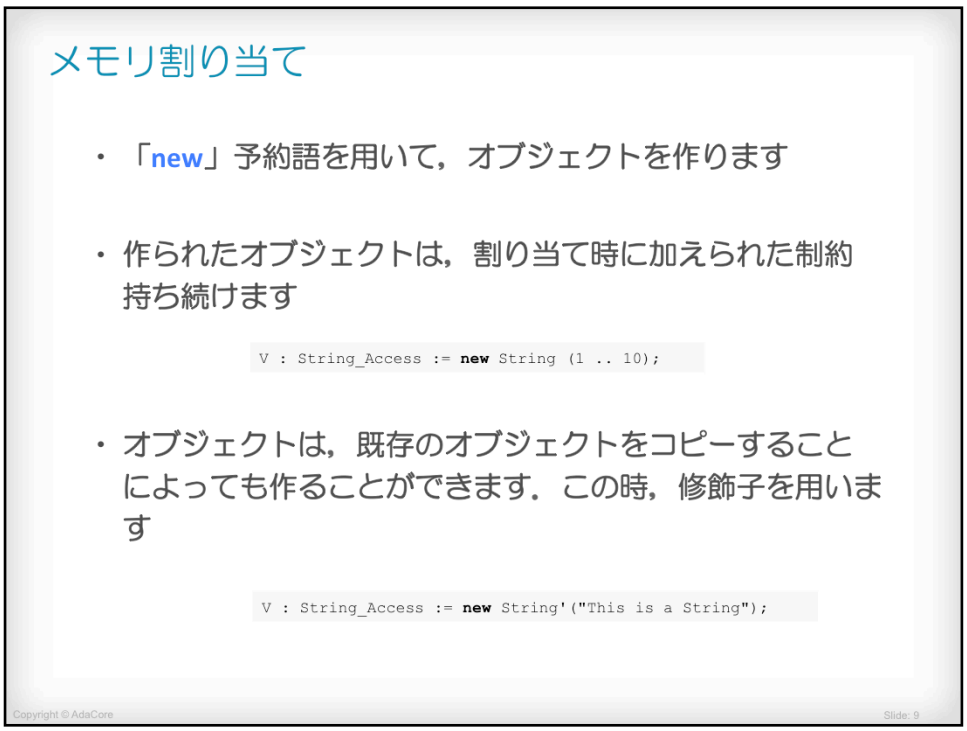
C 言語や C++ 言語においてポインタを初期化するときに，数値ゼロを用いるか，NULL リテラル定数を用いるかは，いつも大きな論争になります．

Ada プログラミング言語は，ヌル（null） と呼ぶ予約語を持ち，言語の一部として，あいまいさがないように規定しています．

このことによって，コンパイラは，どのように使用されるかについての明確な前提を行うことができ，もし，ターゲットとなるオブジェクトが指定されていなければ，デフォルトのヌル値をアクセス型オブジェクトに対して与えます．

ヌルキーワードは，作成しようとする機能の明確な仕様を提供するために，比較と代入でも用いることができます．

この例題は，使った場合に危険となるヌルの利用について示しています．例の最後の行は，意味的には正しいのですが，不適切な場合を示しています．（前の行で）割り当てた整数オブジェクトの場所がもはや不明になり，解放することができません．古典的なメモリリークです．



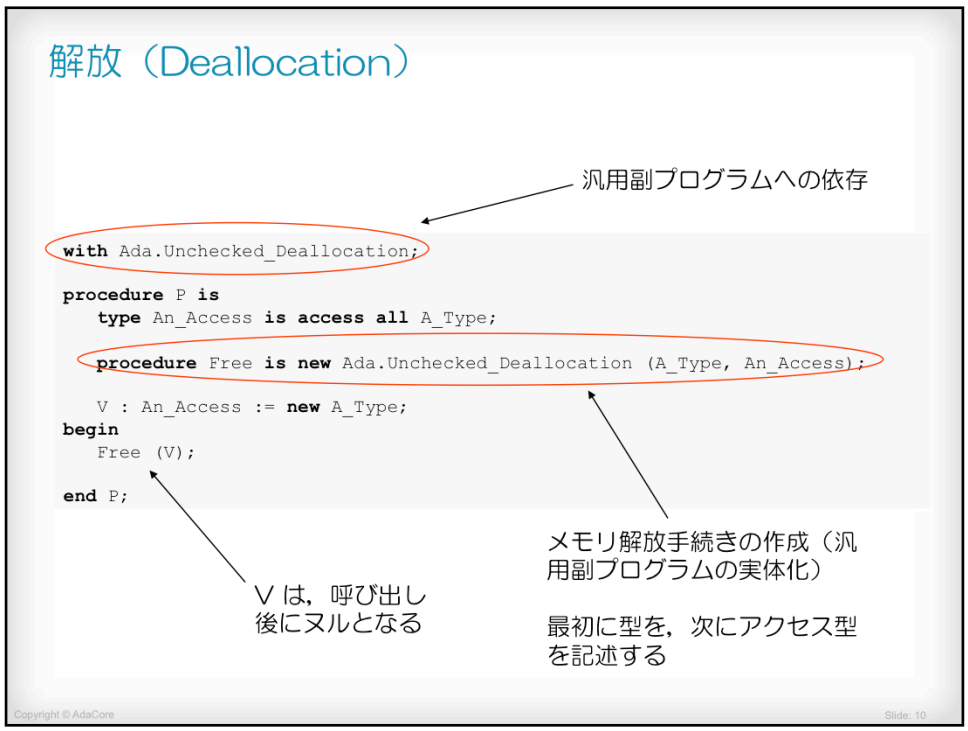
これまでの説明で，「new」予約語を用いました．この語は，ターゲットとなるオブジェクトで，明示的にアクセス型を初期化するのに用います．

これまでのスライドを思い起こすと，new は，記憶領域プールのオブジェクトに対して割り当てを行いました．

「new」予約語に続いて，アクセス型のターゲットの型，その型が必要とする全ての制約を記述します．

割り当てによりメモリが実際に消費され，その消費量は事前に知る必要があるということは，道理にかなったことです．

修飾子は，ターゲット型となる既存のオブジェクトをコピーするために用います．デフォルトとなるオブジェクトが利用可能であると便利です．また，レコード型や配列型に対しては，集成代入（aggregate assignment)を用いたいと思うでしょう．



割り当てたオブジェクトが不要になったときに，それらを記憶域プールに返す機構が，プログラム言語には必要です．

これは，メモリリークへ対処するための基本的な戦略です．Ada 言語は，賢く手際のよい手段を用いています．

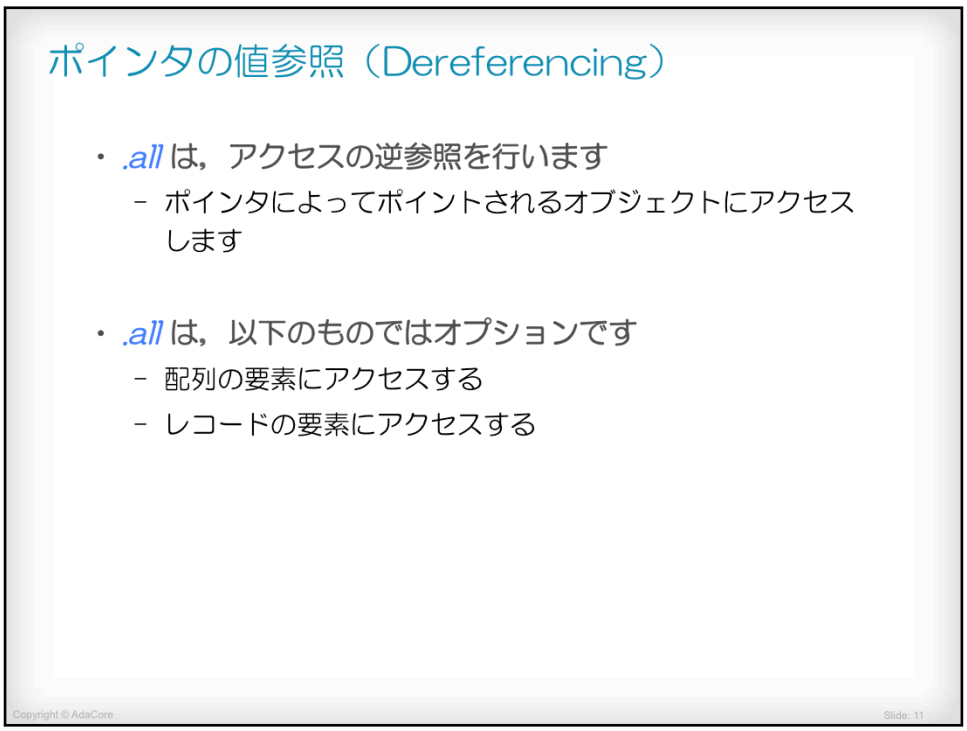
いつ，どうやってオブジェクトが解放されるべきかを，開発者が完全に理解していることを，Ada 言語は要求しています．記憶域プールの各オブジェクトは全て，解放するために汎用体ライブラリを実体化することによって，（オブジェクトの解放を）行います．

開発者は，このことをやり過ぎと思うかもしれませんが，この労力は無駄になりません．また，アクセス型は，そうせざる終えないときのみ使用するということを思い出して下さい．

汎用手続きは，コールされた Unchecked_Deallocation であり，標準の Ada ライブラリから利用可能です．それは 2 つの汎用パラメータとともに，実体化されます．最初のパラメータは，ターゲットの型であり，二番目がアクセス型名になります．

このコード例には，A_Type というターゲット型とともに An_Access というアクセス型が宣言されています．次に，Free と云う名前で， Unchecked_Deallocation を実体化するのに．このアクセス型を（ターゲット型とともに）用いています．

アクセス型オブジェクトは，割り当てられたターゲットオブジェクトとともに明示的に初期化されます．（ここでの）唯一のボディ部の文は，Free の呼び出しであり，ターゲットオブジェクトを解放し，アクセス型オブジェクトを再度ヌルにセットしています．



一度，アクセス型オブジェクトにターゲットオブジェクトが与えられると，開発者は，アクセス型オブジェクトの値参照によって，ターゲットオブジェクトへのアクセスが可能になります．

「.all」予約語をアクセス型のオブジェクト識別子に加えることで，値参照を行います．これによってターゲットオブジェクトにアクセス出来るようになります．

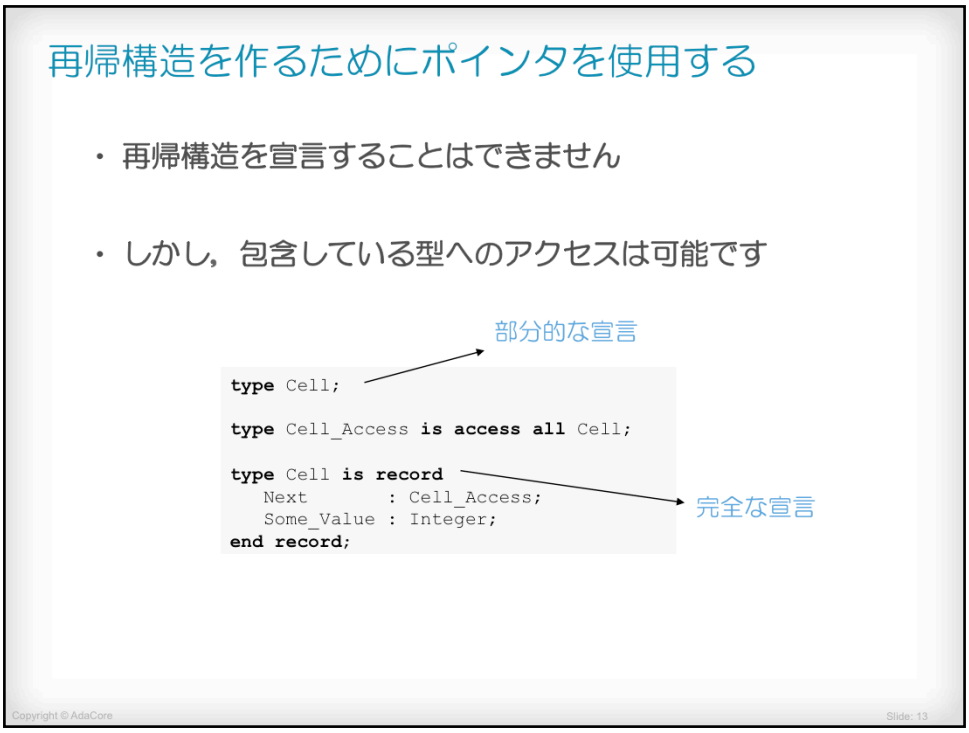
これは，C 言語や C++ 言語におけるアスタリスクと同様のものです．

「.all」 接尾辞は，ある環境下では，オプションです．即ち，配列の要素や，レコードの要素へアクセスする場合です．

値参照の例

```
type R is record
   F1, F2 : Integer:
end record;

type A_Int is access all Integer;
type A_String is access all String;
type A_R is access all R;

V_Int    : A_Int := new Integer;
V_String : A_String := new String'("abc");
V_R      : A_R := new R;

[…]

V_Int.all := 0;
V_String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V_R.all := (0, 0);
V_R.F1 := 1; -- similar to V_R.all.F1 := 1;
```


Adlib



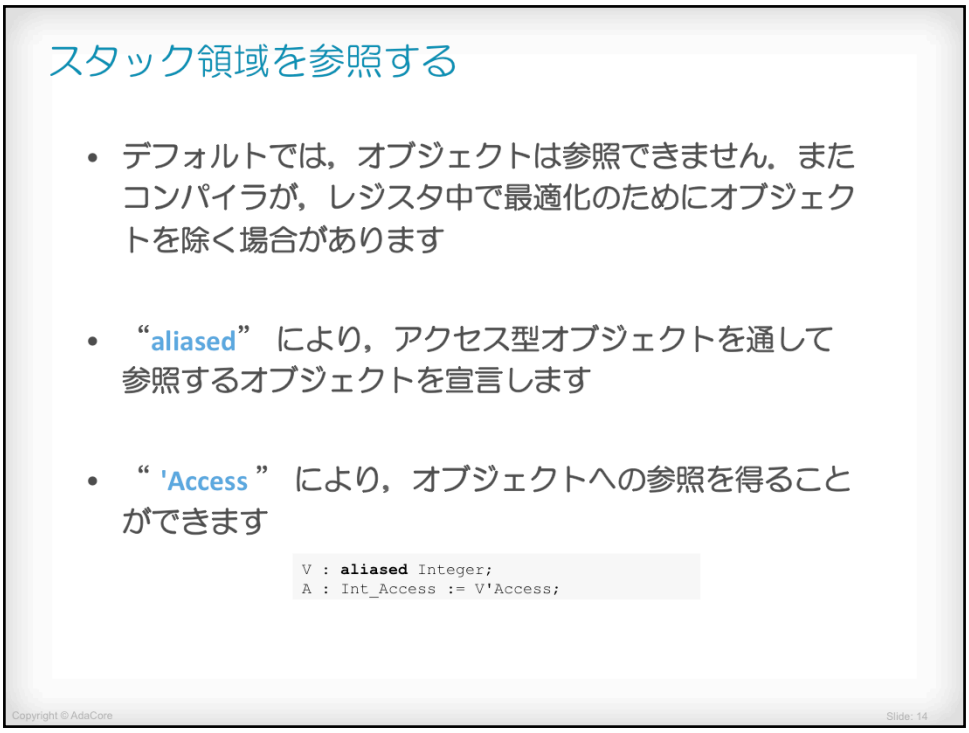
再帰データ構造は，プログラミングにおいて，大変に有益です．特に，連結リストやそれと同様のデータ構造において，役立ちます．

包含するスコープへのアクセス型オブジェクトを組み込むような賢い方法を使うことなしに，再帰データ構造を宣言することはできません．

スライドにある例は，伝統的な単一の連結リストです．最初に，Cell という型を部分的に宣言します．次に，その型へのアクセス型を宣言します．

次に，レコード型としての完全な宣言中に，一つのフィールドとして，Cell アクセス型の実体を組み込みます．

再帰的なデータ構造の最後（のデータ）を識別するために，比較における「null」予約語を用いると，容易になります．



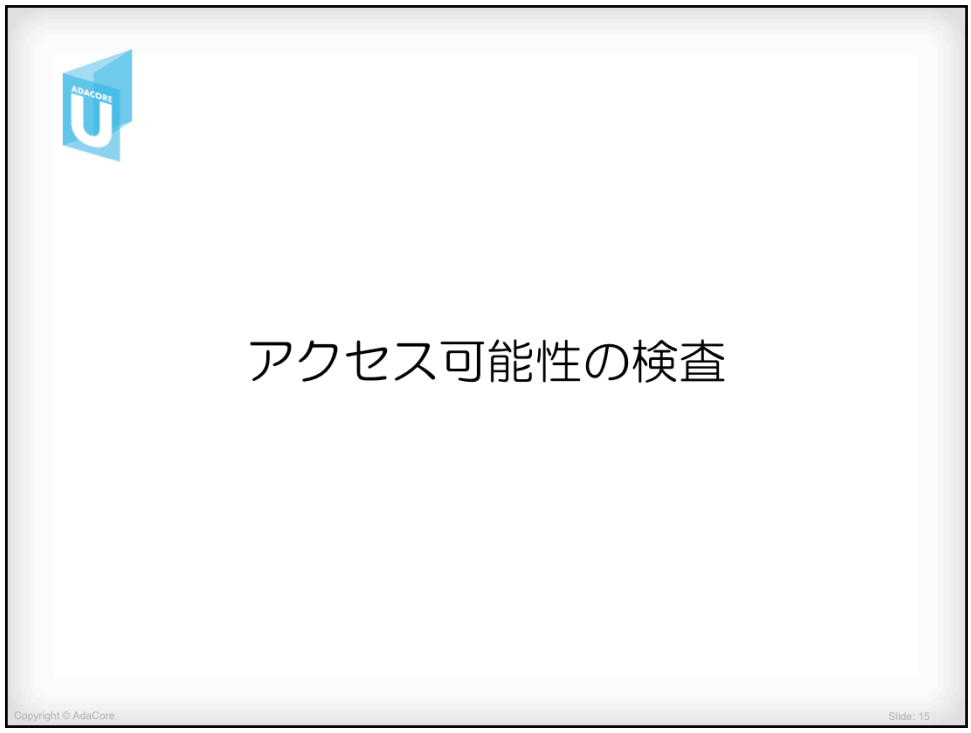
これまで，動的に割り当てられたヒープオブジェクトにアクセスために使用するアクセス型について見てきました．

同様にスタック領域に割り当てられるアクセスオブジェクトに対しても，アクセス型を使用することができます．

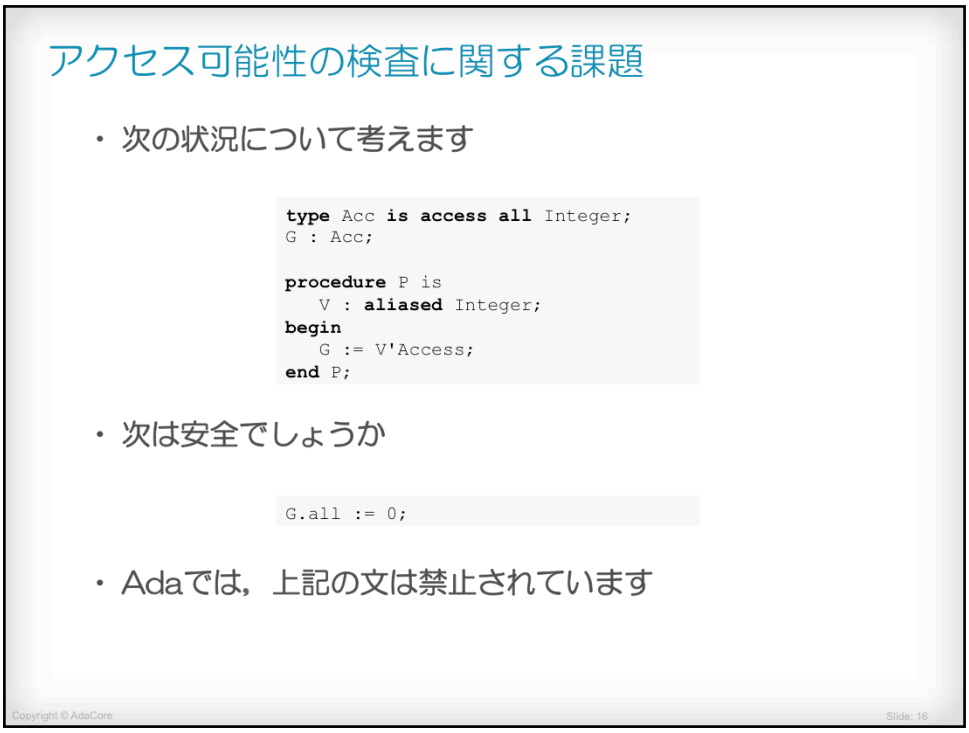
最初のポイントは次です．スタック領域を用いて実体化する全てのオブジェクトには，何の前提もありません．実際，コンパイラはこれらのオブジェクトを配置するのに，自由にレジスタを用いることができます．

従って，アクセス型を用いて，これらオブジェクトにアクセスしようとすると問題となります．

2 つめのポイントは次です．オブジェクトを宣言するとき，オブジェクトに対して「alias」 キーワードを付加します．そのオブジェクトを利用したい全てのアクセス型オブジェクトを，Object'Access という形式で初期化します．



講義の最後となるこのセクションでは，アクセス可能性に関係した規則について説明します．このセクションが終わったあと，確認のクイズを始めます．



アクセス可能性レベル規則は，何らかの理由で不正となっているターゲットオブジェクトにアクセスしないように設計されています．この規則を破ることがないように，十分に注意しなくてはなりません．

ここでは，整数に対するアクセス型を宣言し，G として，アクセス型オブジェクトの実体を宣言している場合について考えます．

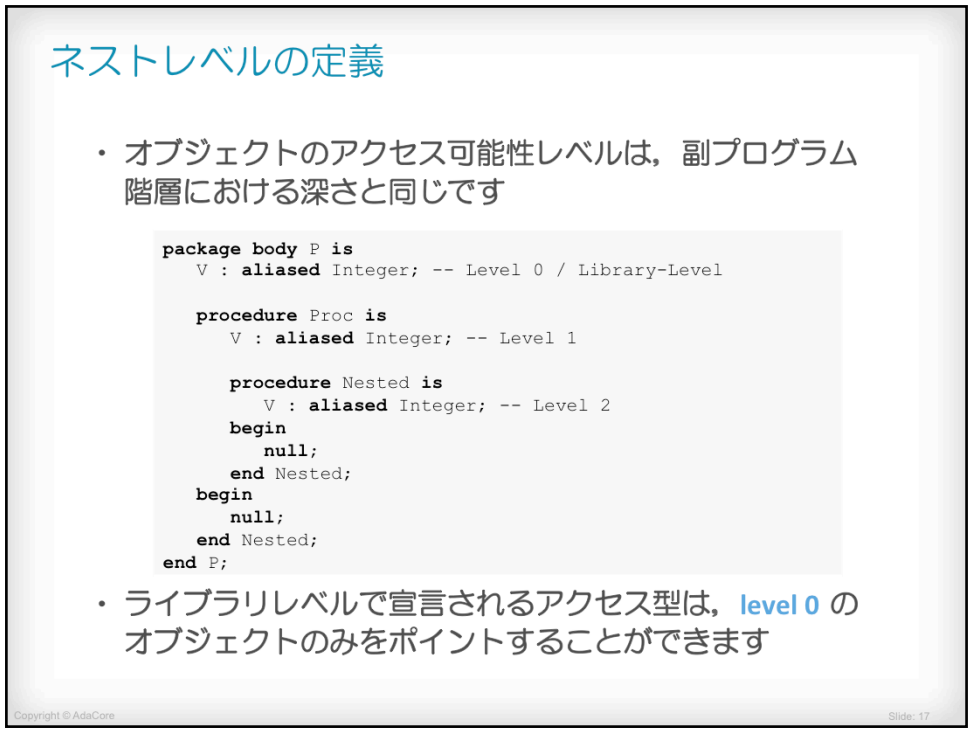
手続き P は，別名化した整数 V スタック領域のオブジェクトを宣言し，G がそのオブジェクトにアクセスできるように宣言しています．

ここでは，手続き P の外側で，G.all を用いています．これは安全でしょうか．

Ada では，アクセス可能性レベルに関する規則を破っているため，このコードを禁止しています．

G の有効期間は，V より長いため，手続き P が終了したとき，スタック領域のオブジェクトである V は，もはや正当でありません．副プログラムが終了したときには，関係するスタック領域は，もはや存在しないからです．

それゆで，G の値参照は，もはやスコープ中には存在しないターゲットオブジェクトにアクセスしようとしていることになります．



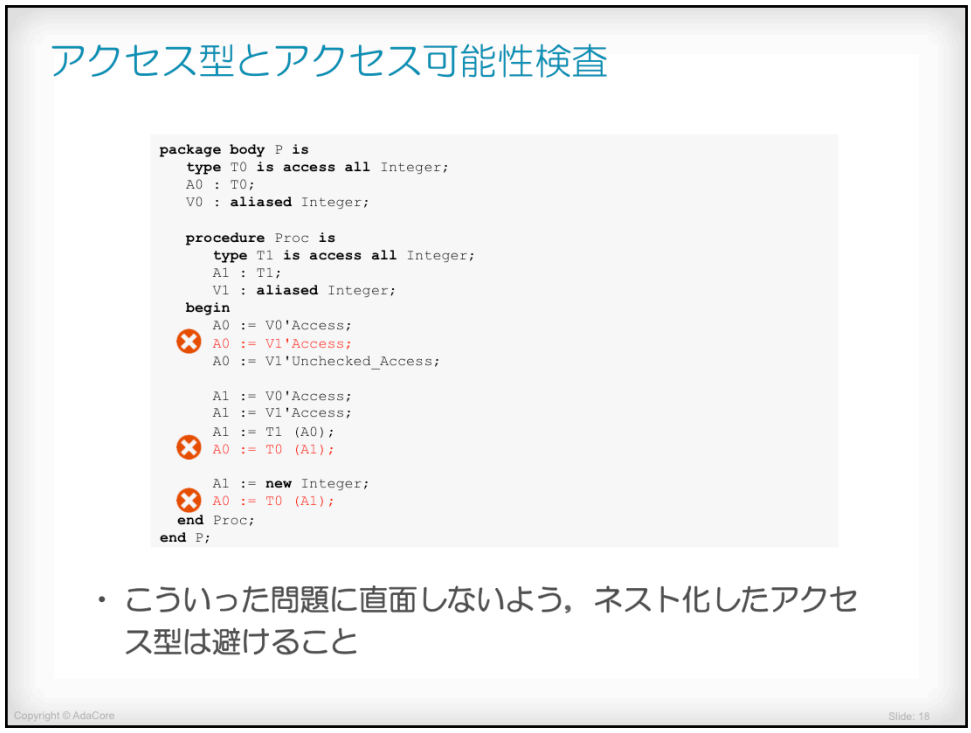
理解すべき重要な概念に，ネストレベルがあります．ネストレベルは，プログラムのソースコードの構造化に用いる入れ子と同様にスコープのレイヤによって定まります．

Ada では，アクセス可能性レベル（accessibility level）という用語を用います．入れ子において，もっとも外側のスコープであるレベル 0 と比較して，あるコードを包含しているスコープのレベルを示す用語です．

このコードの例では，レベル 0 からレベル 2 までの 3 つのアクセス可能性レベルがあります．

規則は次です．ライブラリレベルのコードは，アクセルレベル 0 のオブジェクトにアクセスできます．他のレベルにいるコードは，アクセスオブジェクト自身のスコープ中に存在し続けるアクセスターゲットオブジェクトにアクセスできます．通常，このことは，（アクセスオブジェクト）自身を包含しているスコープの全てを意味します．

各包含されているスコープでは，最後に，スタックは解放されるため，この規則には意味があります．包含している（側の）スコープ中にいるアクセスオブジェクトは，解放されたスタック領域中のオブジェクトをターゲットとすることになり，不正なアクセスとなるからです．



ネスト化したアクセス型を極力避けるというのが，これまでのスライドで学ぶべき重要な点です．

例では，Level 0 において，整数ターゲットオブジェクトに対するアクセス型の宣言があります．

A0 として実体を作り，次に，別名化した整数スタック領域オブジェクトを宣言しています．

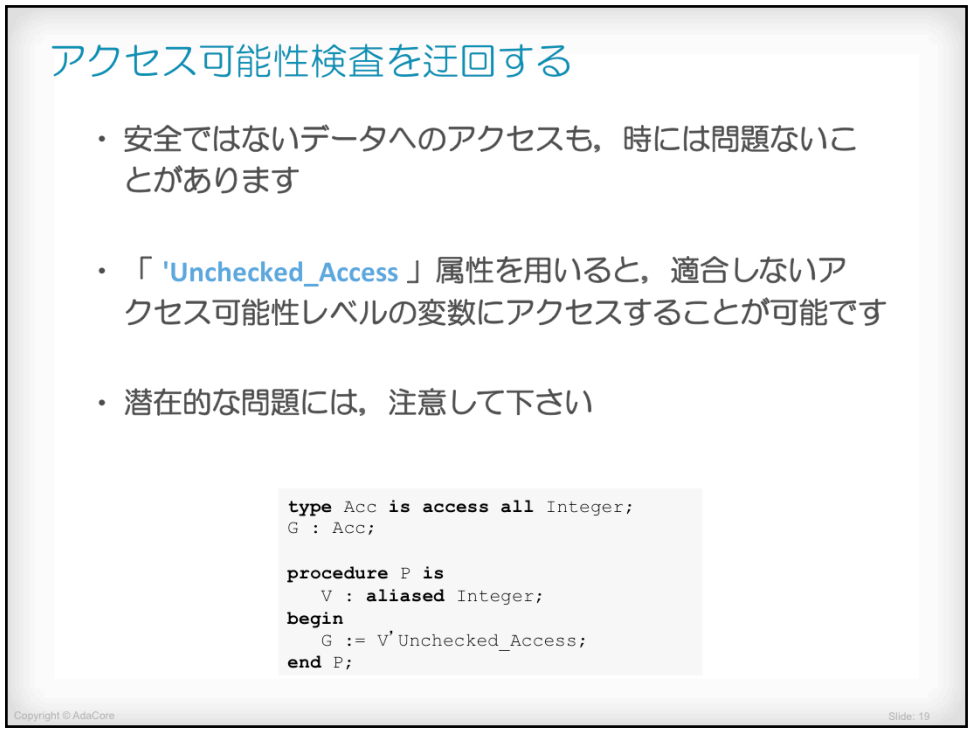
手続き Proc では，整数ターゲットオブジェクトに対するアクセス型を宣言しています．次に，この実体と別の別名化したスタック領域オブジェクトを宣言しています．

アクセス型オブジェクトは，「’ Access」 属性を用いることによって，別名化したスタック領域オブジェクトにアクセスできることを思い出して下さい．

ターゲットオブジェクト V1 に対して，より高いレベルの A0 が宣言されているということが，最初の誤りです．

二番目のエラーも同様です．アクセス型に型変換していますが，ターゲットオブジェクトは，アクセス型オブジェクトよりも低いレベルで宣言されています．

三番目のエラーも同様で，ターゲットオブジェクト A1 が，アクセス型オブジェクト A0 よりもより低いレベルであるためです．



先のスライドの例で，「‘Unchecked_Access」と呼ばれる 'Access 属性の変種を用いていることに気がついたかもしれません．

データへの安全ではないアクセスを行ったときに，開発者の責任で，いつなら，あるいはどうして問題がないかを判断しなくてはなりません．

Ada は，「‘Unchecked_Access」 属性によって，アクセス可能性レベル規則を意図的に破ることを認めています．また，アクセス可能性レベルが不適合であるスタック領域のオブジェクトにアクセスすることも認めています．

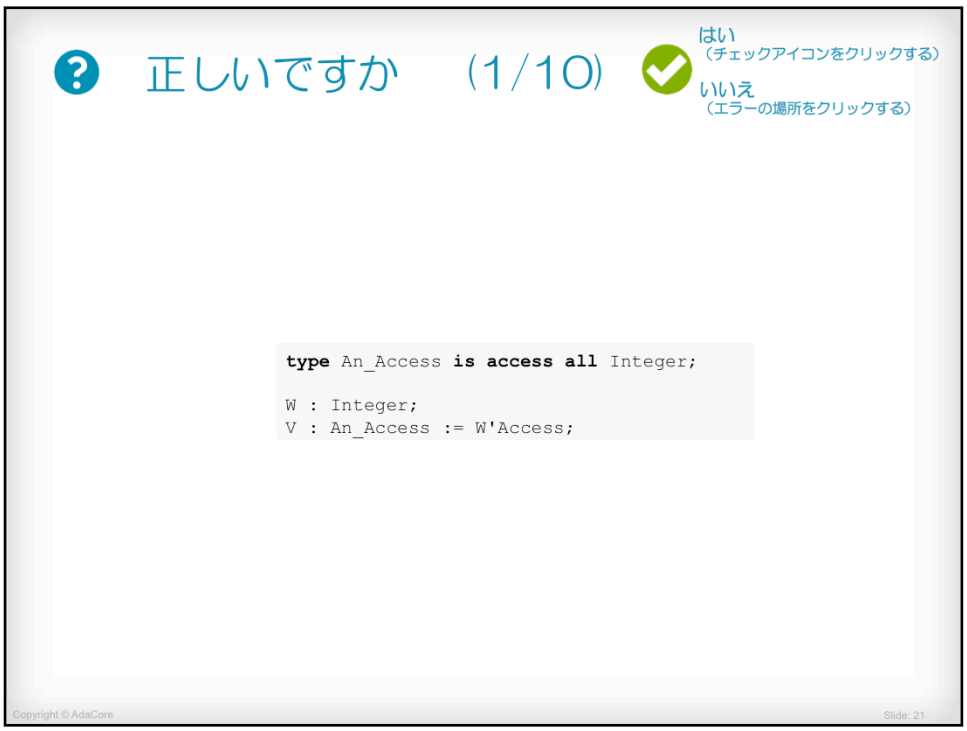
この属性を利用することは危険です．危険性を上手に伝えることはできませんが，このメッセージを受け止めてくれることを望んでいます．

もし，プログラムソースコードを読んだときに 「‘Unchecked_Access」 を利用する場所に出会ったら，よりよい方法を知らないプログラマによって利用されているのではないかと，注意するようにしてください．

いよいよこの講義の最後になりました.

既に，Ada 言語のアクセス型について十分な知識を得たと思います．理解を試すためのちょっとしたクイズをやりましょう.

幸運を！



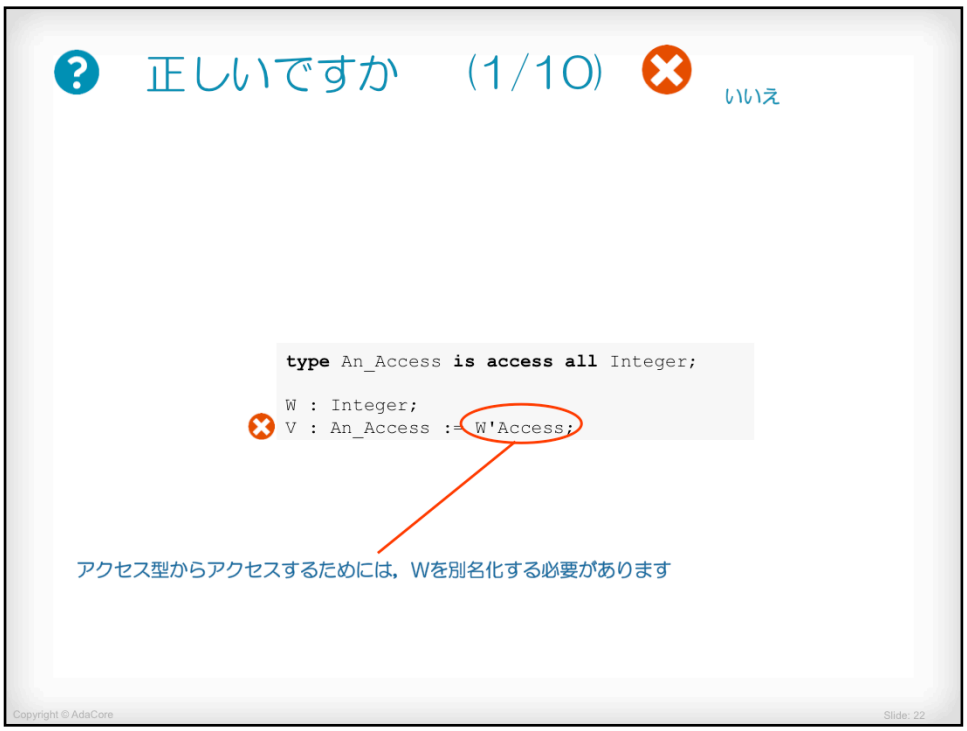
ゆっくりとはじめます．次のコードが正しいかどうかを判断して下さい．

アクセス型は，整数ターゲット型に対して宣言されています．

整数 W を宣言し，'Access属性を，明示的に An_Access 型のオブジェクトである V を初期化するために用いています．

このコードは正しいでしょうか．

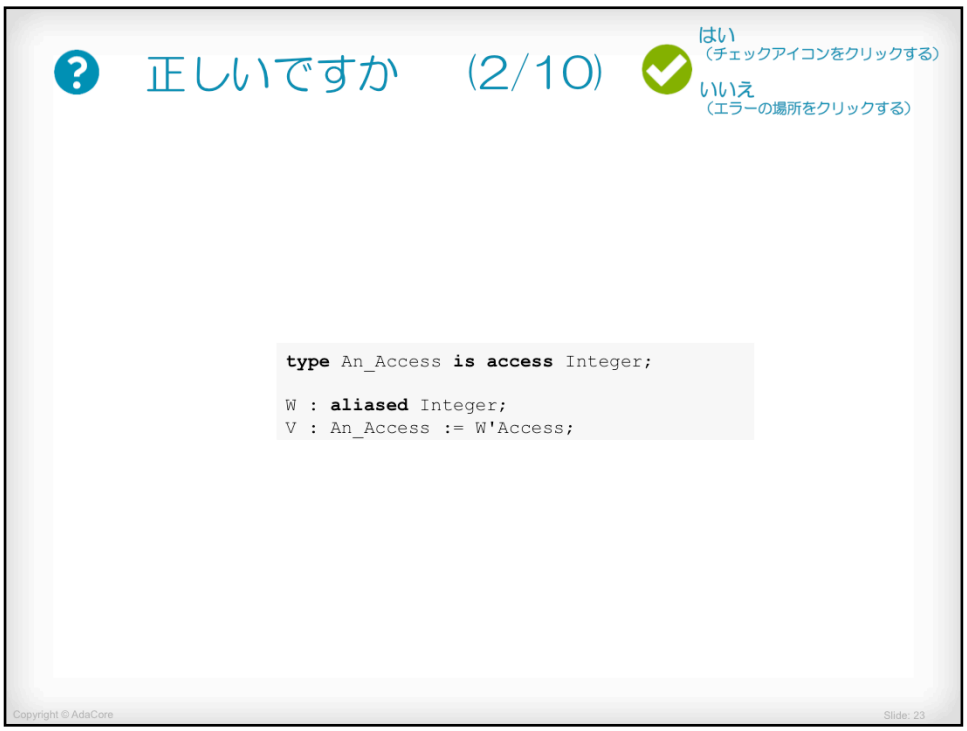
もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



このコードは間違っています．

'Access 属性の利用は，「aliased」 予約語を用いて宣言されたスタック領域オブジェクトに制限されています．

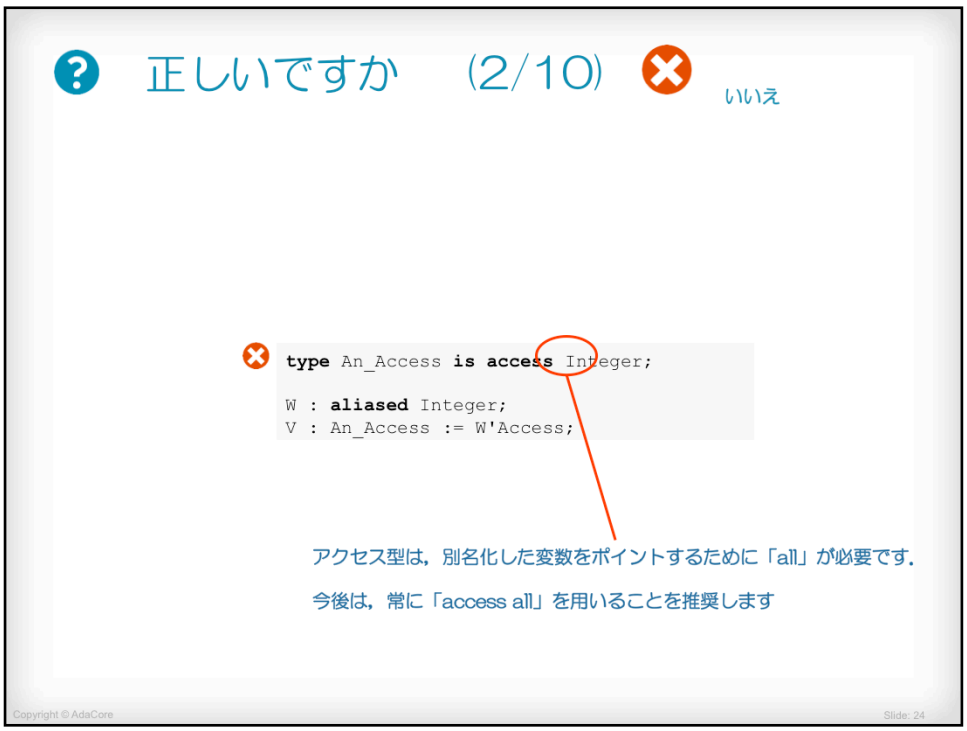
この例において，W は，「aliased」 予約語を用いて宣言されていないので，アクセス型からアクセスすることができません．



この 2 番目の問題では，先のコードを少し変更しました．もう一度同じ質問をします．

このコードは正しいでしょうか．

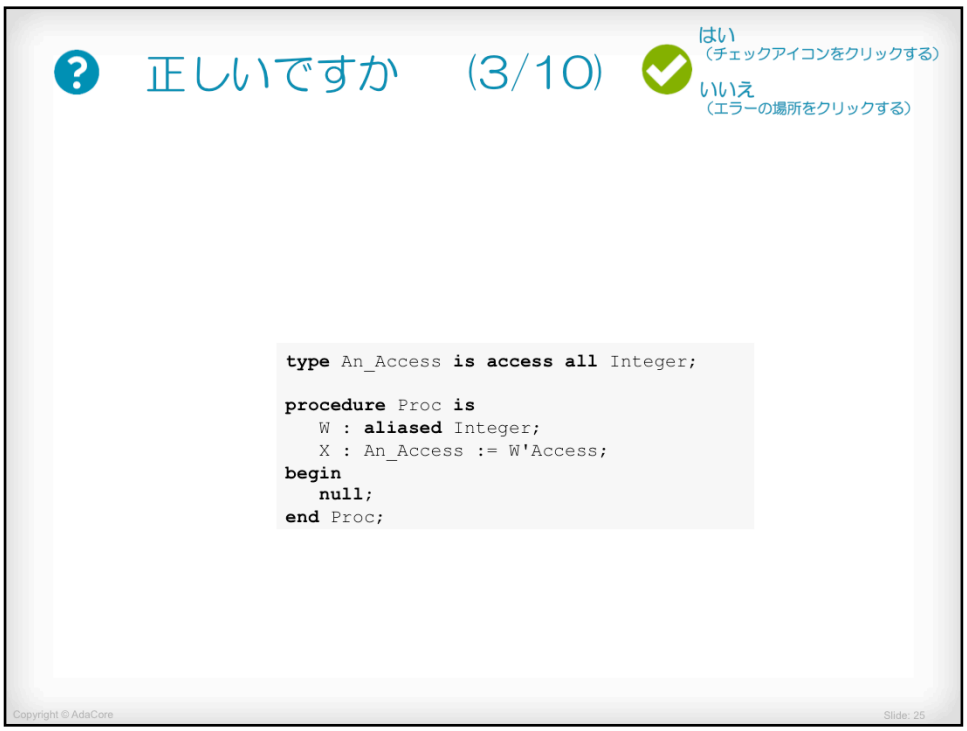
もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



このコードは，間違っています．

アクセス型オブジェクトが，スタック領域オブジェクトをターゲットとするためには，アクセス型を，「all」 予約語と共に宣言しなければなりません．

柔軟性を増すために，開発者は，つねに アクセス型宣言において，access all メソッドを使用することを勧めます．

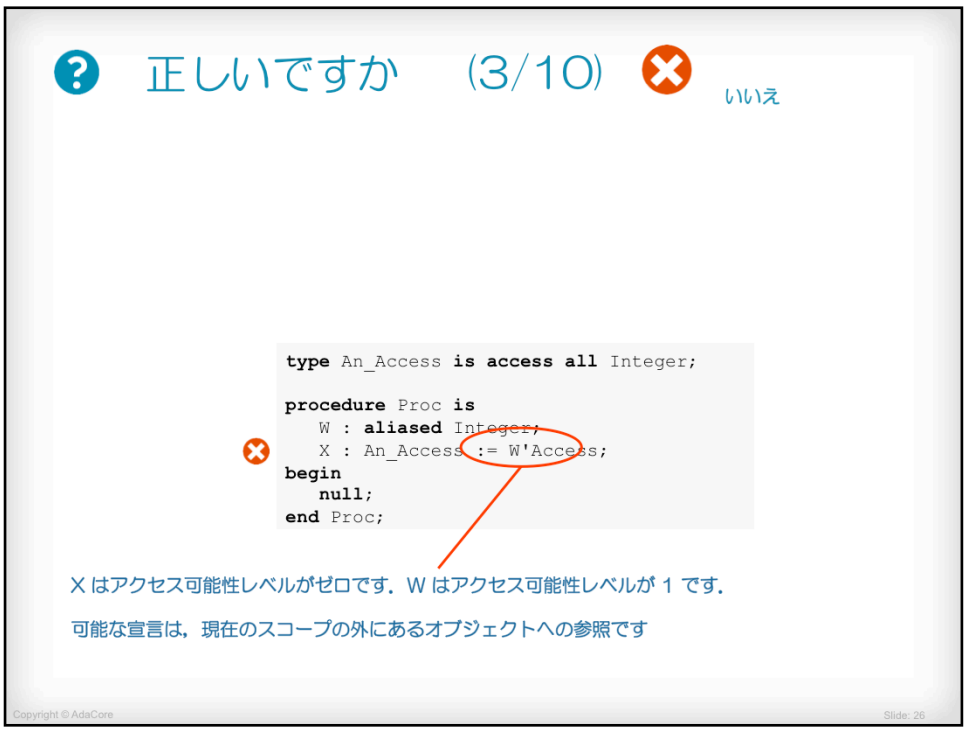


この問題でも，整数ターゲット型に対する An_Access アクセス型を，引き続き利用します．

宣言部には，二つのスタック領域オブジェクトがあり，ボディ部には空の手続き Proc があります．

W は，別名化した整数型であり，X は，W の’Access 属性を利用し，明示的に初期化したアクセス型の実体です．

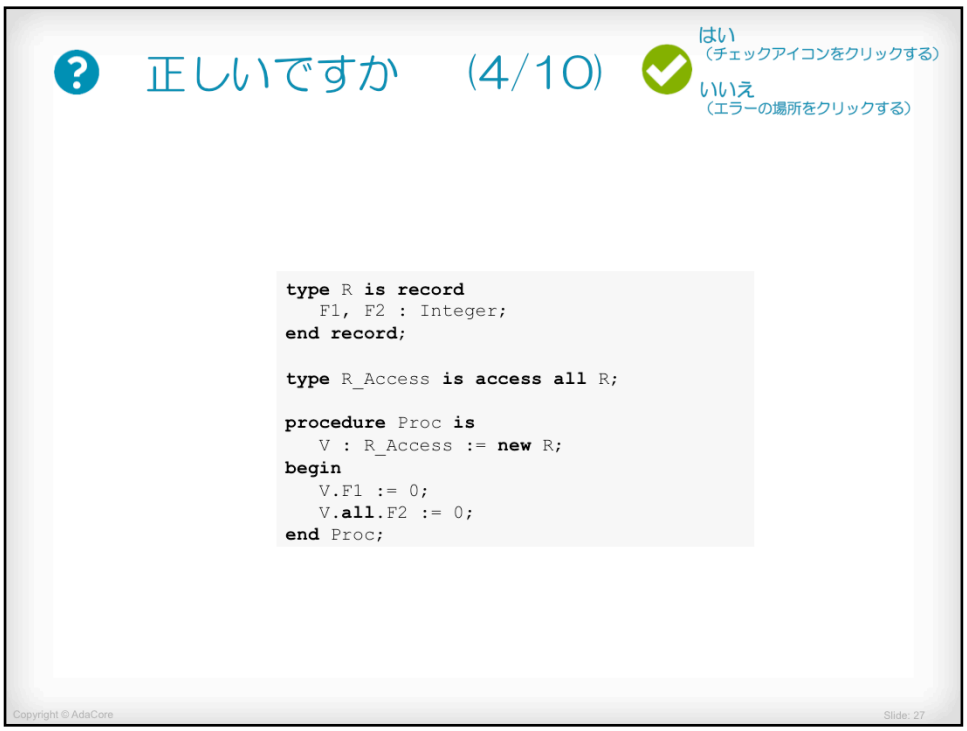
もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



残念なことにこのコードは正しくありません．アクセス可能レベル規則を破っているため，コンパイルすることができません．

X のアクセス可能性レベルは 0 です．その型が宣言されているレベルからそうなります（一行目）．一方で，W のアクセス可能性レベルは 1 です．

この宣言で可能になるのは，現在のスコープの外側にあるオブジェクトへの参照になります．

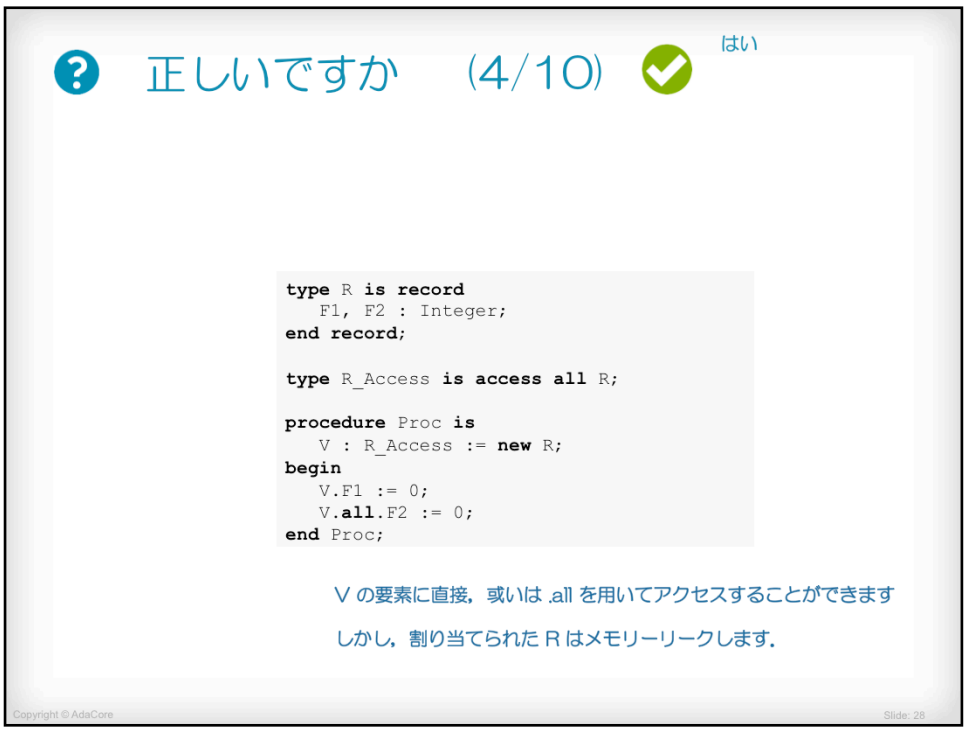


ここでは，レコード型 R と関連するアクセス型を見ます．

手続き Proc は，動的に割り当てられた R の実体にアクセスするために，明示的に初期化されたアクセス型オブジェクトを宣言しています．

手続き Proc のボディ部で，R の 2 つの要素のうち，一つは 「.all」 予約語を用いて，他方は用いていません．

もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



このコードは構文的には正解です．アクセス型オブジェクトは，.all 接尾辞のあるなしに関わらずターゲットオブジェクトの要素を参照することが可能です．

しかし，このコードには明らかなメモリーリークがあります．動的に割り当てられた R の実体は，記憶域プールに返ることができないからです．Proc　手続きが終了するとき，V スタック領域オブジェクトは失われます．その場合，ソースコードの他の部分は，V がターゲットとしていたオブジェクトを解放することができません．

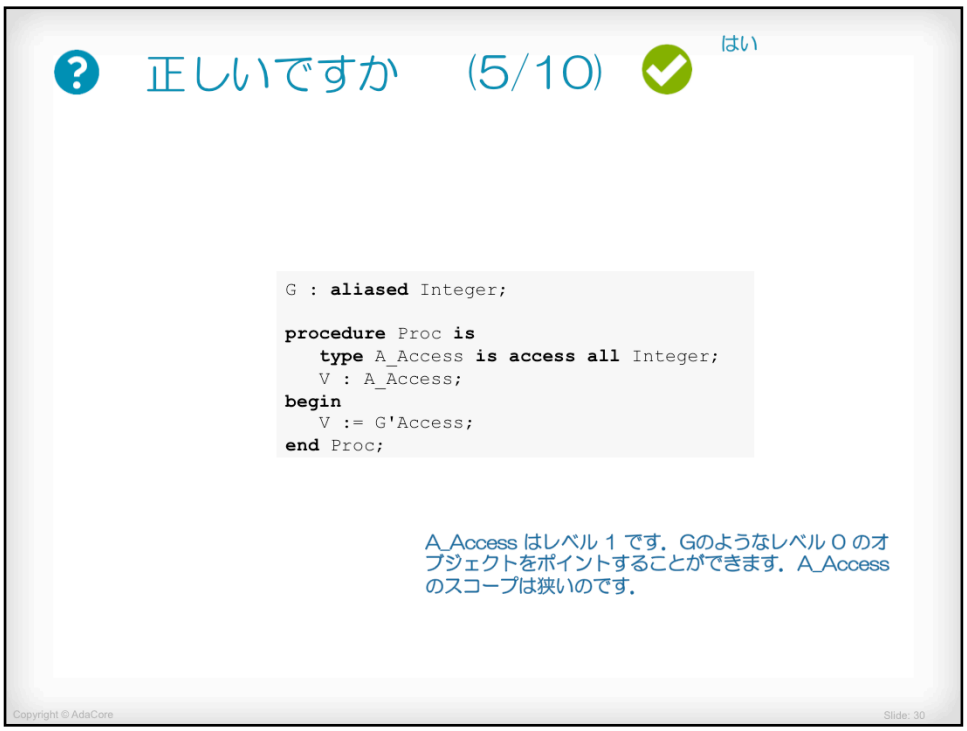
5 番目の問題は，G と呼ぶ，レベル 0 の整数別名化スタック領域オブジェクトの宣言に関するものです．

手続き Proc は，レベル 1 であり，スタック領域を含む全ての記憶域から指し示すことのできる整数オブジェクトへのアクセス型を宣言しています．

このアクセス型の実体は，オブジェクト V として宣言されています．唯一のボディ部の文は，レベル 0 のスタック領域オブジェクト G に対して，’ Access 属性を用いて，V のターゲットとしています．

このコードは正しいでしょうか．

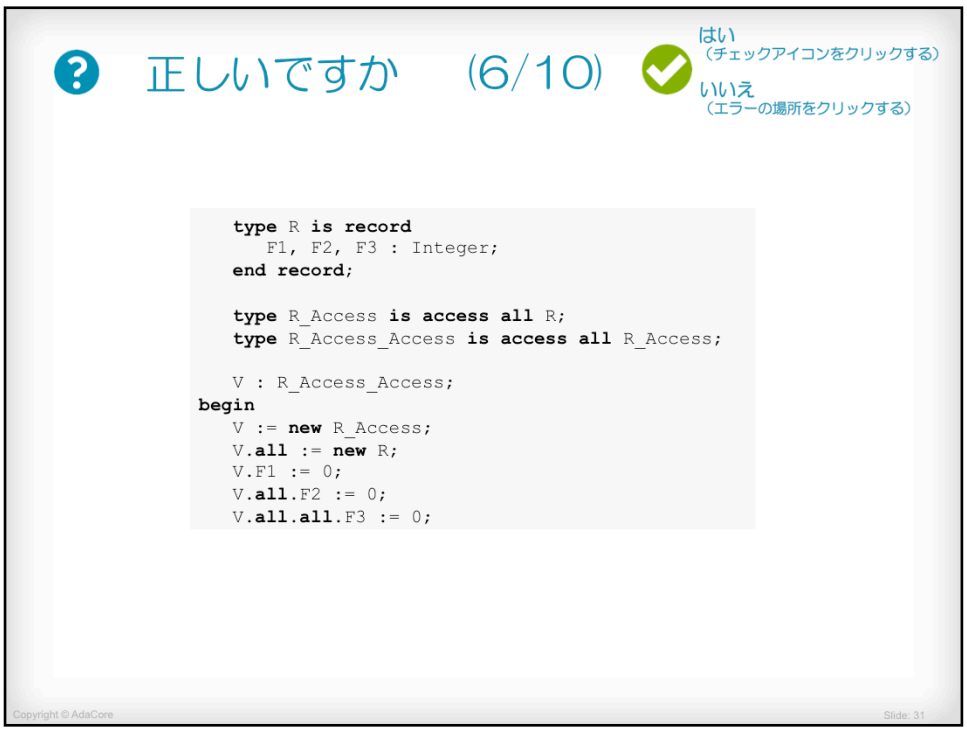
もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



これは正しいコードです．アクセス可能性規則により，包含しているスコープ中で宣言されたスタック領域オブジェクトへのアクセスが可能です．

アクセス型 A_Access は，レベル 1 に居ます．これは，G のような，スコープに関して低いレベルであるレベル 0 として宣言されたオブジェクトを，ターゲットとすることができます．

規則によれば，包含しているスコープ中で宣言されたオブジェクトは，包含されているスコープからアクセス可能ということを思い出して下さい．



次の問題は，アクセス型をターゲット型としている別のアクセス型，という少し複雑なケースについて考えます．

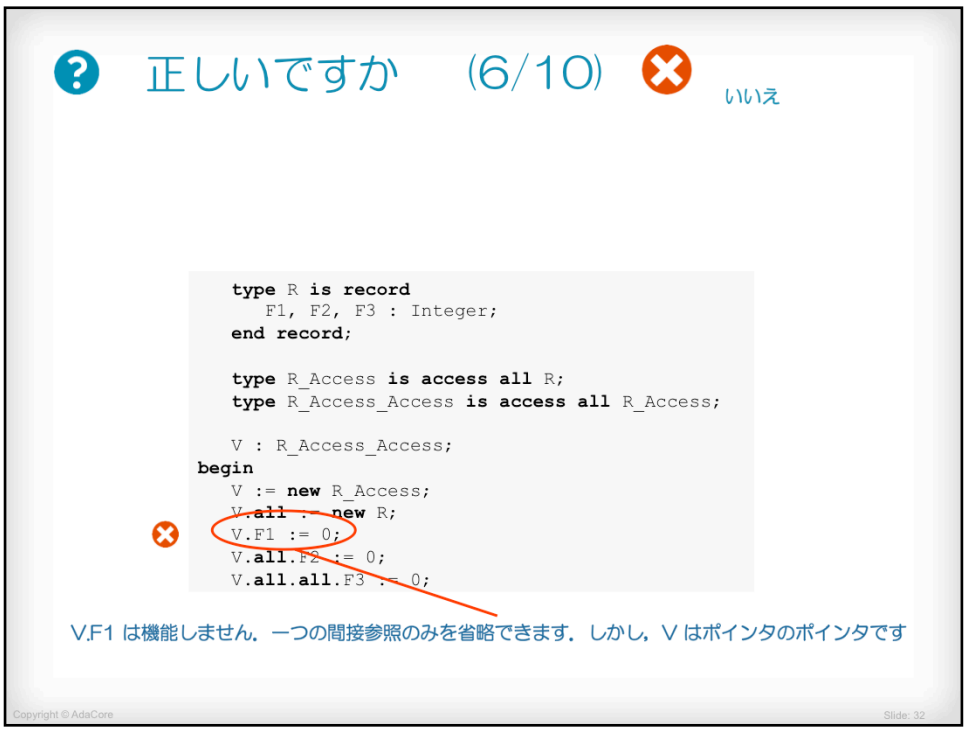
ここには，R という単純なレコード型宣言があり，ターゲット型が R であるアクセス型宣言が続きます．

次に，この宣言されたアクセス型 R_Access をターゲット型とするアクセス型を宣言します．

更に，R アクセス型を二重にターゲットとするスタック領域オブジェクトを作ります．

ボディ部には，V にアクセスしたり，操作したりする様々な文があります．

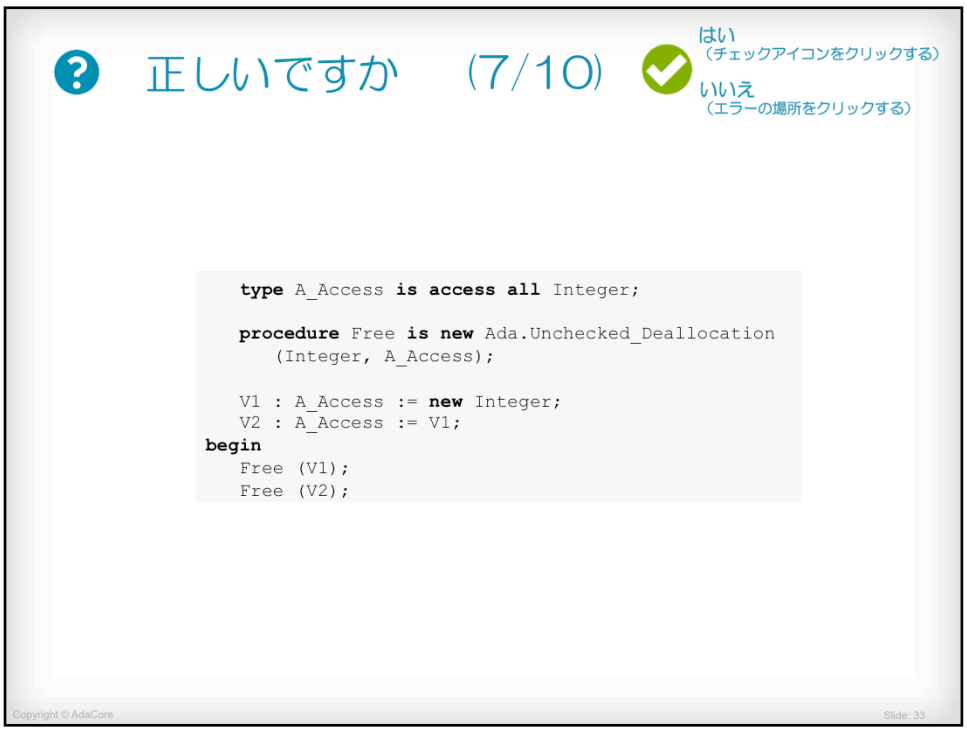
もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



ほとんどの文は合っています．しかし，コンパイルエラーを生じる文が一つあります．

V.F1 で始まる行は，不正です．は二重にターゲット化したアクセス型において，「.all」を省略することはできないからです．

「.all」 は，ターゲットオブジェクトの要素にアクセスするときにのみ省略できることを思い出して下さい．この場合，ターゲットオブジェクトは，それ自身アクセス型であり，何ら要素を持っていません．



7 番目の問題は，割り当てられたオブジェクトを記憶域プールに戻すことに関する知識を問う問題です．

このコードは，アクセス型を宣言することから始まっています．このアクセス型は，スタック領域を含む全ての記憶域プールの全ての整数オブジェクトをターゲットにしています．

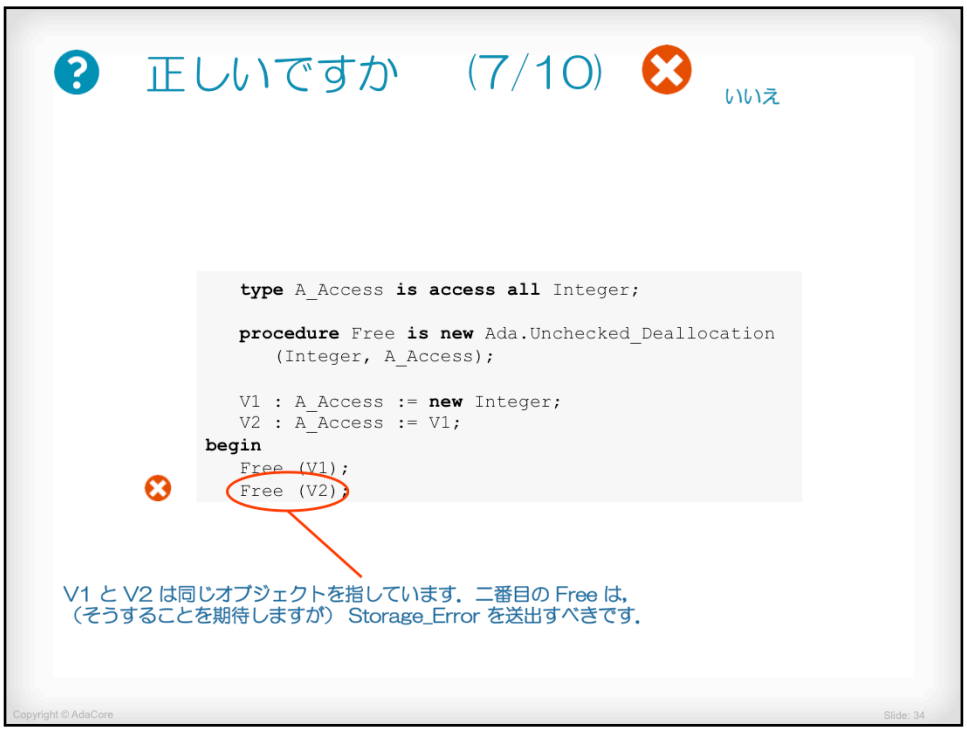
次に，汎用手続き Unchecked_Deallocation にターゲット型とアクセス型を引数として与え，Free として，宣言しています．

アクセス型の二つのスタック領域オブジェクトがあります．最初は，明示的に整数型で割り当て初期化を行った形で宣言されています．二番目は，最初のターゲットオブジェクトを使って，明示的に初期化行った形で宣言されています．

ボディ部には，V1 を引数にした Free 呼び出しと，V2 を引数にした Free 呼び出しがあります．

このコードは正しいでしょうか．

もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



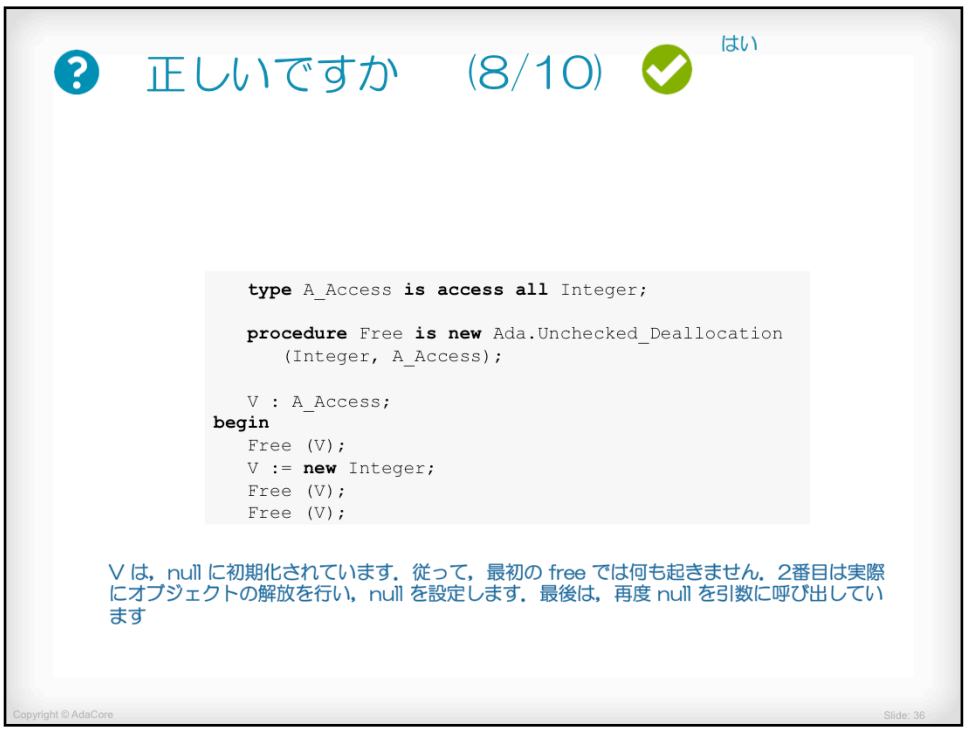
このコードをコンパイルできます．しかし，実行時に，二番目の Free 呼び出しは，STORAGE_ERROR 例外を送出します．

V1 と V2 は，両方とも同じオブジェクトを指しています．最初の Free が完了したとき，ターゲットオブジェクトは，記憶域プールに返ります．しかし，V2 は，依然として（記憶域プールに戻ってしまった）オブジェクトを指したままです．

前回の問題にあったコードをもとに，別のコードを作ります．ボディ部の文を変えています．Free の呼び出しの後，確保した整数を V へ代入する文が続きます．

次に，2 つの連続する Free 呼び出しが続きます．

もし，このコードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



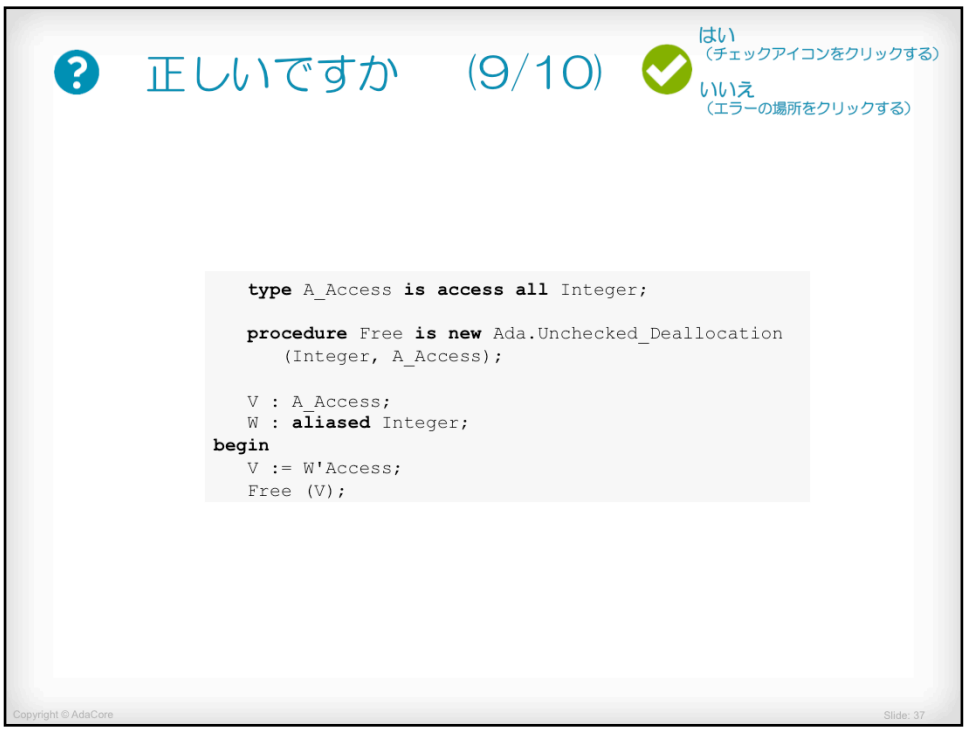
これは正しいコードです．

明示的に初期化されていない V を引数にして，Free を呼び出すことは可能です．全てのアクセス型オブジェクトは，デフォルトでは，null に初期化されます．

V が，いったんターゲットオブジェクトを割り当てられている場合，Free に渡されると，ターゲットオブジェクトを記憶域プールに戻します．

Free は，null をアクセス型オブジェクトにセットします．従って，次の連続する 2 つの Free 呼び出しは，規則に適合し，正当です．

記憶すべき重要なことは，null とともに呼び出す Free には，意味がないということです．



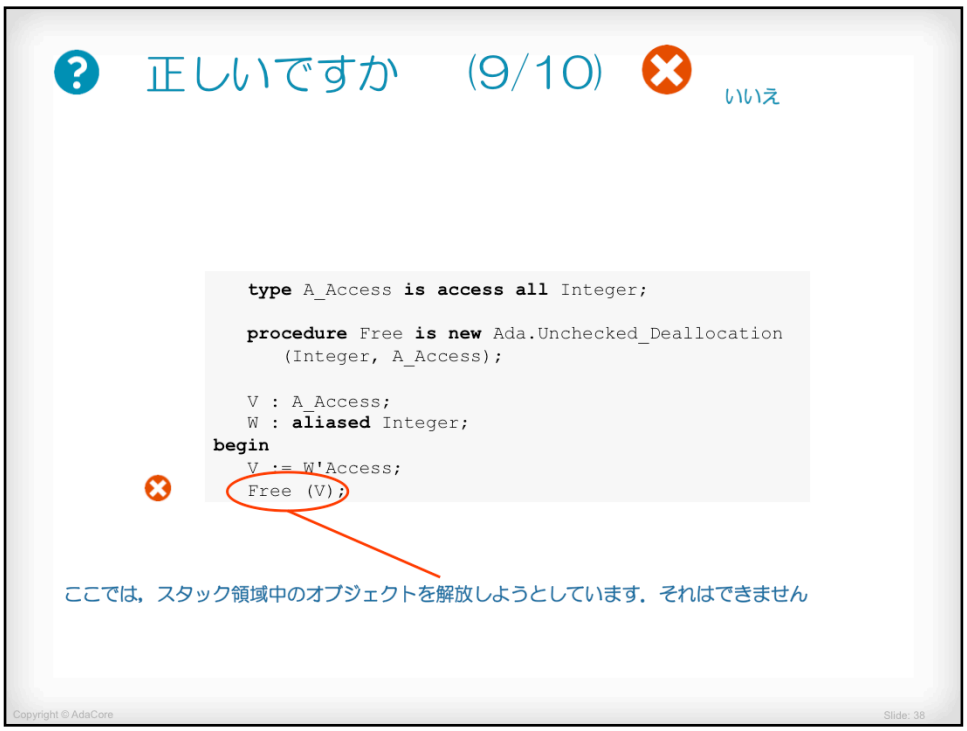
最後から 2 番目の問題は，再度 A_Access アクセス型を取り上げます．このアクセス型は，スタック領域を含む全ての記憶域プールの整数オブジェクトをターゲットにします．

Unchecked_Deallocation を，アクセス型とターゲット型の Free 手続きとして実体化します．

次にアクセス型 V の実体と，別名化整数 W の実体を宣言しています．

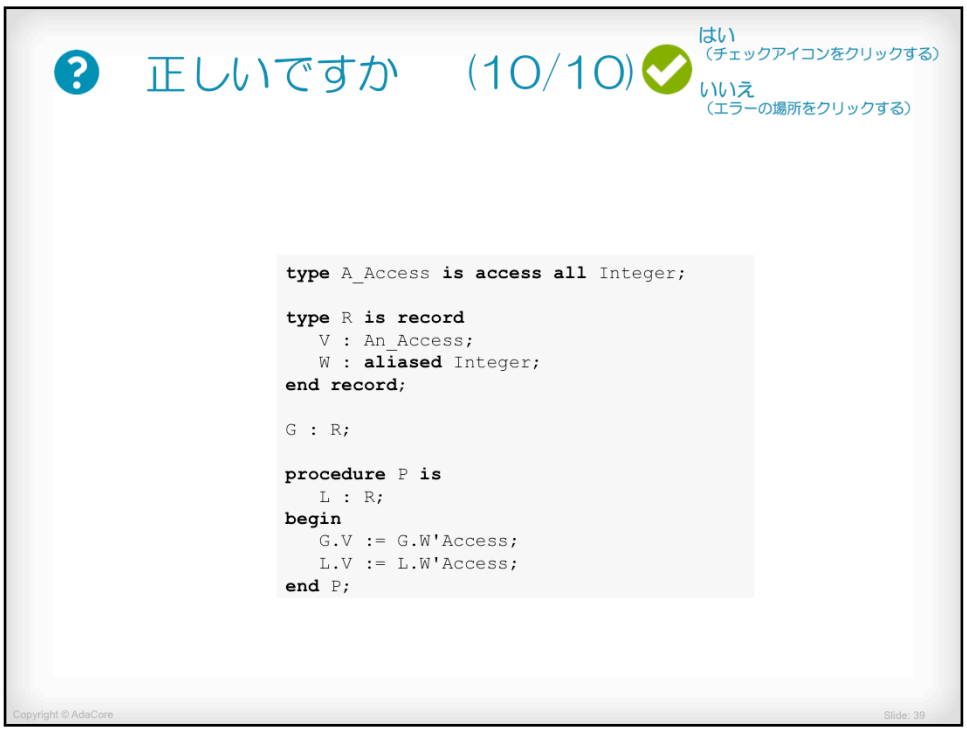
ボディ部で，Vは，Wへのアクセスを割り当てています．次に，Free にこの V を引数として渡します．

もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



このコードは不正です．

スタック記憶域プール中のターゲットオブジェクトを解放することは不正です．この場合，w はスタック領域オブジェクトなので，v はスタック記憶域プール中のオブジェクトを指していることになります．



最後の問題では，スタック領域を含む全ての記憶域プール中の整数ターゲットオブジェクトに対するアクセス型を宣言しています．

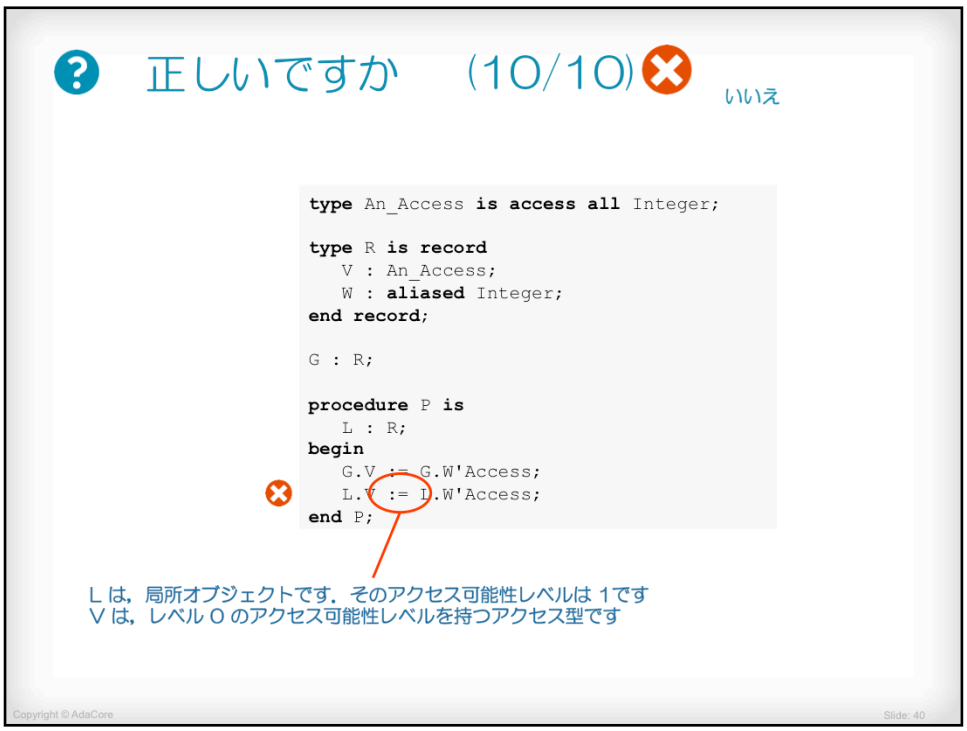
次に，アクセス型の実体と別名化整数の実体を要素とするレコード型を宣言しています．

レコード型の実体として，オブジェクト G があります．

手続き P は，その内部に L と呼ぶ，別のレコード型実体を持っています．

P のボディ部では，二つの文があります．ここでは，アクセス型の要素を別名化整数要素へのターゲットオブジェクトとしています．

もし，コードが正しいと思えば，「はい」のアイコンをクリックして下さい．或いは，間違っていると思うコードの場所をクリックして下さい．



このコードは正しくありません．

2　番目のボディ部の文は，アクセス可能性規則を破っています．入れ子の中で，低い位置から，アクセス型宣言を用いようとしています．

V　と名付けられた要素は，レベル 0 のアクセス可能レベルであるアクセス型です．

Ada 大規模プログラムのアクセス型のコースに参加いただきありがとうございました.

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.