Capítulo 9 - Subprogramas e Parâmetros: Subprogramas normalmente descrevem computação, duas formas para ganhar acesso a dados ou serem processados ou o acesso direto a variáveis não locais, eu, pela passagem de parâmetro. Aqueles que são passados por parâmetro, o acesso é pelo nome subprograma e são mais flexíveis do que o acesso direto.

Formas de Parâmetros: Variáveis fictícias pois não são variáveis no sentido tradicional, sendo vinculadas ao armazenamento na maioria dos casos, essa vinculação é feita por meio de uma variante do programa.

Métodos de Passagem de Parâmetro: Maneira como são transmitidos os parâmetros Do subprograma.

Passagem por Valor: O valor do parâmetro real é usado para inicializar o formal correspondente, agindo como uma variável local no subprograma. Normalmente é por cópia, o acesso é mais eficiente, vantagem para escalares, é rápida. Desvantagem, é necessário armazenamento adicional para o parâmetro formal e o parâmetro real deve ser copiado para a área de armazenamento.

Passagem por Resultado: Modelo para parâmetros do modo de entrada, nesse caso, nenhum valor é passado para o subprograma. Ajudado como variável local, o parâmetro formal antes do controle ser transmitido de volta para o chamador, é transmitido para o parâmetro real de volta. Tem as mesmas vantagens e desvantagens da passagem por valor, se os valores retornados por cópia exigem armazenamento extra, a dificuldade se dá pela transmissão de um caminho de acesso, resultando na implementação de cópia de dados.

Passagem por Valor-Resultado: Modo de entrada e saída, onde os valores reais são copiados. O parâmetro real é copiado para o formal no subprograma, requerendo armazenamentos múltiplos, sendo uma desvantagem.

Passagem por Referência: Implementação para parâmetros no modo de entrada e saída, não copia os valores de cada lado para o outro, como no anterior, transmite um caminho de acesso, apenas um endereço para o subprograma chamado. Fornecendo um caminho de acesso para a coluna, podendo acessar o parâmetro real, que é compartilhado com o subprograma chamado. É vantajoso em termos de espaço, não sendo necessário duplicar as cópias, desvantagens, o acesso aos parâmetros formais são mais lentos, a comunicação de uma volta para o subprograma chamado é necessário, podendo ter chamadas erradas no parâmetro real.

Exemplos de Passagens (Valor e Referência)

Esse código em C demonstra a passagem de parâmetros para funções e o conceito de passagem por valor.

1. Inclusão de bibliotecas:
   ```c
   #include <stdio.h>
   ```
   Esta linha inclui a biblioteca padrão de entrada e saída em C, que fornece funcionalidades para entrada de dados do usuário (como com `scanf()`) e para exibição de dados (como com `printf()`).

2. Declaração da função `AlteraValor()`:
   ```c
   void AlteraValor(int val);
   ```
   Esta linha declara uma função chamada `AlteraValor()` que não retorna nada (`void`) e recebe um parâmetro do tipo inteiro (`int val`). Esta função será definida posteriormente.

3. Função `main()`:
   ```c
   int main() {
       // código
   ```

```
    return 0;
}
```

Esta é a função principal do programa, onde a execução começa e termina. Ela retorna um valor inteiro (`int`) para indicar se o programa terminou com sucesso ou com erro.

4. Variável `val`:
```c
int val = 50;
```
Aqui, uma variável inteira chamada `val` é declarada e inicializada com o valor 50.

5. Impressão do valor antes da chamada da função:
```c
printf("Valor antes da funcao: %d\n", val);
```
Esta linha imprime o valor da variável `val` antes de ser alterada pela função `AlteraValor()`.

6. Chamada da função `AlteraValor()`:
```c
AlteraValor(val);
```
Aqui, a função `AlteraValor()` é chamada passando o valor de `val` como argumento. Note que é passado o valor da variável `val`, não a própria variável. Isso significa que a função `AlteraValor()` trabalhará com uma cópia do valor de `val`, não com `val` em si.

7. Impressão do valor depois da chamada da função:
```c
printf("Valor depois da funcao: %d\n", val);
```
Esta linha imprime o valor de `val` após a chamada da função `AlteraValor()`, mas como veremos adiante, o valor não será alterado aqui.

8. Definição da função `AlteraValor()`:
```c
void AlteraValor(int val) {
    // código
}
```
Esta é a definição da função `AlteraValor()` que foi declarada anteriormente. Aqui, `val` é uma variável local para esta função, que recebe uma cópia do valor passado como argumento.

9. Expressão dentro da função `AlteraValor()`:
```c
val = (val * 15);
```
Dentro desta função, o valor de `val` é multiplicado por 15 e atribuído de volta à variável `val`. No entanto, esta alteração só afeta a cópia local de `val` dentro desta função, não afeta a variável `val` na função `main()`.

10. Impressão do valor dentro da função:
```c
printf("Valor dentro da funcao: %d\n", val);
```
Esta linha imprime o valor de `val` após a alteração dentro da função `AlteraValor()`.

11. Impressão de uma mensagem:
```c
puts("Quando o valor e passado por referencia, o valor da variavel dentro da main NAO e alterado apos a execucao da expressao.");
```

```
Esta linha imprime uma mensagem explicativa sobre o comportamento do programa.

No geral, o código ilustra o conceito de passagem de parâmetros por valor em C, onde uma cópia do valor da variável é passada para a função, em vez da própria variável. Isso significa que quaisquer alterações feitas nos parâmetros dentro da função não afetarão os valores das variáveis originais fora da função.

Este código em C é semelhante ao anterior, mas utiliza a passagem de parâmetros por referência.

1. Inclusão de bibliotecas:
   ```c
   #include <stdio.h>
   ```
   Essa linha inclui a biblioteca padrão de entrada e saída em C, como no exemplo anterior.

2. Declaração da função `AlteraValor()`:
   ```c
   void AlteraValor(int *val);
   ```
   Aqui, declaramos a função `AlteraValor()`, que recebe um ponteiro para um inteiro (`int *val`) como parâmetro. Isso significa que a função aceita o endereço de uma variável inteira como argumento.

3. Função `main()`:
   ```c
   int main() {
       // código
       return 0;
   }
   ```
   A função principal do programa, onde a execução começa e termina, similar ao exemplo anterior.

4. Variável `val`:
   ```c
   int val = 50;
   ```
   Declaração e inicialização de uma variável inteira `val` com o valor 50, assim como no exemplo anterior.

5. Impressão do valor antes da chamada da função:
   ```c
   printf("Valor antes da funcao: %d\n", val);
   ```
   Impressiona o valor de `val` antes de ser alterado pela função `AlteraValor()`.

6. Chamada da função `AlteraValor()` com referência ao valor:
   ```c
   AlteraValor(&val);
   ```
   Aqui, é passado o endereço de `val` (referência ao valor) como argumento para a função `AlteraValor()`. Isso significa que a função terá acesso direto à variável `val` na memória.

7. Impressão do valor depois da chamada da função:
   ```c
   printf("Valor depois da funcao: %d\n", val);
   ```
```

Imprime o valor de `val` após a chamada da função `AlteraValor()`. Como veremos adiante, o valor de `val` será alterado pela função.

8. Definição da função `AlteraValor()`:
```c
void AlteraValor(int *val) {
    // código
}
```
A definição da função `AlteraValor()`. Aqui, `val` é um ponteiro para um inteiro, o que significa que ele aponta para a localização na memória onde o valor de `val` está armazenado.

9. Expressão dentro da função `AlteraValor()`:
```c
*val = (*val * 15);
```
Dentro desta função, o valor apontado por `val` (ou seja, o valor de `val` na função `main()`) é multiplicado por 15 e atribuído de volta à mesma localização na memória.

10. Impressão do valor dentro da função:
```c
printf("Valor dentro da funcao: %d\n", *val);
```
Imprime o valor apontado por `val` após a alteração dentro da função `AlteraValor()`.

11. Impressão de uma mensagem:
```c
puts("Quando o valor e passado por referencia, o valor da variavel dentro da main e alterado apos a execucao da expressao.");
```
Uma mensagem explicativa sobre o comportamento do programa é impressa.

Este código demonstra a passagem de parâmetros por referência em C, onde as alterações feitas nos parâmetros dentro da função afetam diretamente os valores das variáveis originais fora da função. Isso é possível porque a função recebe o endereço de memória da variável, não apenas seu valor.

Capítulo 11 → Dados Abstratos e Encapsulamento

Abstração: Vinculação/representação de entidade com os atributos mais significativos. Permitindo a coleta de exemplares de entidades em grupos onde os atributos comuns não precisam ser considerados.

Exemplo: Atributos de uma pessoa e, se dada é uma ave, a descrição do clado não precisa incluir os atributos de aves.
Os atributos são uma ferramenta contra a complexidade das programas, com o propósito de simplificar o processo de programação.

Abstração de Processos: São dados os programas, fornecendo uma maneira pela qual o programa especificado fica um processo sem fornecer os detalhes de como não são usados: Para ordenar um inteiro (sortentilesti late) é uma ordem de promoção de toda a ação al de fortaleza na não pra representação de dados específicas, sendo os subprogramas dando as operações para esse tipo. Um exemplo são os objetos.
Assim como a abstração de dados, a de processos também é uma ótima ferramenta contra a complexidade, sendo uma forma de tornar programas grandes e/ou complicados mais gerenciáveis.

Ponto Flutuante + pode Dados Alestratar: Em linguagem de alto nível, pontos flutuantes têm um conceito chave na abstração de dados, ocultando de informação, o real armazenamento é oculto ao usuário, sendo que eles não podem fazer nenhuma alteração, além das permitidas.

Condições do contexto do usuário:

As declarações do tipo e as predefinições de operação em um objeto do tipo, que fornecem sua interface, são contidas em uma única unidade sintática. A interface do tipo não depende da representação dos objetos nem da implementação das operações. A implementação do tipo e de suas operações podem estar na mesma unidade sintática ou em uma unidade separada. Outras unidades de programa podem criar variáveis do tipo definido.
A representação dos objetos do tipo é ocultada das unidades de programa que os usam, então as únicas operações diretas possíveis Nesses objetos, as que são fornecidas na definição do tipo. A principal vantagem em empacotar é fornecer um método de organizar um programa em unidades lógicas possíveis de serem compiladas separadamente.

* Tipos de Dados Encapsulados em Ada: Fornecem encapsulamento podendo ser usado para definir tipos de objetos abstratos únicos, inclusive a habilidade de ocultar a sua representação.
Encapsulamento: Recebendo o nome de pacotes em Ada, há 2 tipos. O pacote de especificação, fornecendo a interface de encapsulamento e o pacote de corpo, que fornece a implementação da maioria das entidades mencionadas no pacote anterior. Ambas as partes que se pertencem recebem o mesmo nome.
Ocultamento de Informação: Há 2 abordagens que ocultam a representação das clientes na pacote de especificação, uma é incluir a Recar no pacote induzindo uma chamada privada incluindo a palavra reservada "private" sempre aparecendo no final do pacote. A segunda maneira é definir o tipo de dados abstratos como ponteiros e definir a estrutura apresentada no pacote de corpo.

Construções de Encapsulamento: Organizar os programas em coleções de código e dados logicamente relacionados, cada um sendo compilado sem a recompilação do resto do programa.
Em C, os arquivos funcionam como uma biblioteca, são os arquivos de cabeçalho, podendo ser declarados como parâmetros para tipos struct.

O encapsulamento desempenha um papel fundamental na construção de sistemas de programação de grande porte. Aqui estão alguns pontos-chave relacionados ao encapsulamento:

1. Módulos como Unidades de Programa:
   - Em sistemas de programação de grande porte, o código é geralmente organizado em módulos.
   - Um módulo é uma unidade de programa que possui um nome e pode ser implementada como uma entidade independente.
   - Cada módulo geralmente tem um único objetivo e uma interface bem definida com outros módulos.

2. Modularidade e Reutilização:
   - Um módulo bem projetado é altamente modular, o que significa que ele é composto por componentes distintos e separados.
   - A modularidade promove a reutilização do código, pois os módulos podem ser facilmente integrados em diferentes partes de um sistema ou em sistemas diferentes.

3. Abstração e Encapsulamento:
   - A chave para a modularidade é a abstração. Abstração refere-se à capacidade de concentrar-se nos aspectos essenciais de um conceito ou entidade, enquanto esconde os detalhes desnecessários.
   - Um módulo encapsula seus componentes, o que significa que ele oculta os detalhes internos de implementação dos componentes e fornece uma interface clara e concisa para interagir com esses componentes.

4. Componentes Exportáveis e Escondidos:

- Os componentes exportáveis de um módulo são aqueles que são visíveis externamente ao módulo. Isso inclui tipos, constantes, variáveis, procedimentos, funções, etc., que são projetados para serem usados por outros módulos.
- Por outro lado, os componentes escondidos (ocultos) são utilizados apenas para auxiliar na implementação dos componentes exportáveis. Eles não são acessíveis fora do módulo e ajudam a manter a coesão interna do módulo.

Em resumo, o encapsulamento em módulos de programação é fundamental para promover a modularidade, reutilização e manutenção do código. Ele permite que os detalhes internos de implementação sejam ocultos, enquanto uma interface clara e concisa é fornecida para interagir com os componentes do módulo.

No exemplo fornecido em Ada, um pacote chamado `conversao_metrica` é definido para encapsular informações relacionadas à conversão de unidades métricas. Vou explicar cada parte:

1. Declaração do Pacote (`package conversao_metrica is ... end conversao_metrica;`):
   - O pacote `conversao_metrica` é definido com a lista de informações declarativas entre as palavras-chave `is` e `end`.
   - Este pacote encapsula as informações sobre as constantes de conversão entre unidades métricas.

2. Informações Declarativas:
   - Dentro do pacote, são declaradas constantes que representam fatores de conversão entre diferentes unidades métricas.
   - As constantes declaradas são `pol_cm`, `pe_cm`, `jarda_cm`, e `milha_km`, cada uma representando um fator de conversão específico.

3. Encapsulamento de Amarrações:
   - O pacote pode ser visto como um conjunto encapsulado de amarrações, onde as amarrações são as constantes declaradas dentro do pacote.
   - As constantes dentro do pacote estão encapsuladas, o que significa que elas são acessíveis apenas dentro do escopo do pacote.

4. Uso do Pacote:
   - Para utilizar as constantes definidas no pacote `conversao_metrica`, os programas podem fazer uma declaração de uso ou importação desse pacote.
   - Por exemplo, se um programa precisar usar o fator de conversão de polegadas para centímetros, pode fazer referência à constante `pol_cm` definida dentro do pacote.

Em resumo, os pacotes em Ada fornecem uma forma de encapsular informações relacionadas e fornecer uma interface coesa para acessá-las. Eles permitem organizar e estruturar o código de forma modular e promovem a reutilização, pois as informações encapsuladas podem ser facilmente compartilhadas entre diferentes partes do programa.

Ocultamento de informação, ou information hiding, é uma prática fundamental na programação que visa ocultar os detalhes internos de implementação de um módulo ou componente, fornecendo apenas uma interface externa clara e concisa para interagir com ele. Em Ada, essa prática é facilitada pela divisão de um pacote em duas partes distintas: a declaração do pacote e o corpo do pacote.

1. Declaração do Pacote:
   - A declaração do pacote contém apenas os componentes exportáveis, ou seja, aqueles que são destinados a serem utilizados por usuários externos ao pacote.
   - Essa parte do pacote fornece apenas as informações necessárias aos usuários para que eles possam utilizar o pacote de forma adequada.
   - Os componentes exportáveis incluem tipos, constantes, variáveis, procedimentos e funções que são destinados a serem utilizados fora do pacote.
   - A declaração do pacote permite ao compilador realizar verificações de tipo e fornecer mensagens de erro mais precisas durante a compilação.

2. Corpo do Pacote:
- O corpo do pacote contém todas as declarações dos componentes escondidos, além dos corpos de procedimentos e funções exportáveis.
- Os componentes escondidos são aqueles que são utilizados internamente pelo pacote para implementar os componentes exportáveis, mas que não são destinados a serem utilizados fora do pacote.
- Isso inclui variáveis auxiliares, procedimentos auxiliares e outras implementações internas que não são necessárias para os usuários externos do pacote.
- O corpo do pacote não é acessível externamente e não é visível durante a compilação para usuários que utilizam o pacote.

Ao dividir um pacote em declaração e corpo, o Ada permite um forte encapsulamento de informações, garantindo que apenas os detalhes necessários para o uso adequado do pacote sejam expostos aos usuários externos. Isso promove a modularidade, reutilização e manutenção do código, ao mesmo tempo em que protege a integridade e a coerência interna do pacote.

Neste exemplo em Ada, é definido um pacote chamado `trig` que encapsula funções trigonométricas como seno e cosseno. Vou explicar cada parte:

1. Declaração do Pacote (`package trig is ... end trig;`):
- O pacote `trig` é declarado com duas funções: `sin` e `cos`, que calculam o seno e o cosseno de um número, respectivamente.
- Essas funções são os componentes exportáveis do pacote, ou seja, elas são destinadas a serem utilizadas externamente ao pacote.

2. Corpo do Pacote (`package body trig is ... end trig;`):
- O corpo do pacote `trig` fornece a implementação das funções declaradas na declaração do pacote.
- Dentro do corpo do pacote, é definida uma constante `pi`, que representa o valor de π.
- Para as funções `sin` e `cos`, são fornecidas implementações que calculam o seno e o cosseno do argumento `x`, respectivamente.
- Os detalhes específicos de implementação de como calcular o seno e o cosseno podem variar, mas não são relevantes para os usuários externos do pacote.

3. Encapsulamento de Informações:
- Os detalhes internos de como as funções `sin` e `cos` são calculadas são ocultados dos usuários externos do pacote.
- Os usuários externos só precisam saber que podem chamar as funções `sin` e `cos` fornecidas pelo pacote `trig`, sem se preocupar com os detalhes de implementação.

Este exemplo demonstra como o encapsulamento de informações é alcançado em Ada, permitindo que apenas a interface pública do pacote seja acessível externamente, enquanto os detalhes de implementação são mantidos internamente ao pacote. Isso promove a modularidade, reutilização e manutenção do código, ao mesmo tempo em que protege a integridade e a coerência interna do pacote.

Neste contexto, o tipo racional representa um número racional na forma de uma tupla de dois inteiros, onde o primeiro inteiro representa o numerador e o segundo inteiro representa o denominador. O tipo data também é representado como uma tupla de dois inteiros, possivelmente representando um par de valores para dia e mês, por exemplo.

A expressão (1, 12) é uma tupla de dois inteiros, que corresponderia a um possível valor para o tipo racional, onde 1 seria o numerador e 12 seria o denominador. No entanto, essa expressão não corresponde diretamente ao tipo data, que representa uma data no formato (dia, mês).

É possível comparar um valor do tipo data com um valor do tipo racional, mas isso não faria sentido sem uma definição explícita de como realizar a comparação entre esses tipos.

Quanto aos desafios enfrentados ao representar um tipo por meio de outro tipo:

1. A representação pode conter valores que não correspondem a qualquer valor do tipo desejado. Por exemplo, uma tupla (3, -2) poderia ser uma representação válida para um número racional, mas não faria sentido em um contexto de datas.

2. Comparações podem fornecer resultados incorretos, já que os tipos possuem semânticas diferentes. Uma comparação direta entre uma data e um número racional não teria um significado claro.

3. Os valores de um tipo podem ser confundidos com valores do outro tipo, a menos que um novo tipo seja declarado para cada representação. Isso garante que não haja ambiguidade sobre qual tipo está sendo manipulado.

Em resumo, ao implementar tipos abstratos em um programa, é importante definir operações específicas para esse tipo e garantir que a representação interna seja encapsulada, evitando confusão e ambiguidade com outros tipos de dados.

Esse trecho de código exemplifica a utilização de pacotes em Ada para representar e manipular informações sobre turmas de alunos.

1. Declaração do Pacote (`package tipo_turma is`):
   - Define um pacote chamado `tipo_turma`, que encapsula tipos e procedimentos relacionados ao gerenciamento de turmas.

2. Declaração de Tipos (`type id_aluno is integer;` e `type Turma is limited private;`):
   - Define um tipo `id_aluno` como um inteiro.
   - Define um tipo `Turma` como privado e limitado. Isso significa que somente os membros do pacote podem acessar diretamente os detalhes da implementação desse tipo.

3. Procedimentos (`procedure inclui_aluno (t: in out Turma; aluno: in id_aluno);` e `procedure cria_turma (t: in out Turma; prof: in integer; sala: in integer);`):
   - `inclui_aluno`: Um procedimento para adicionar um aluno à turma.
   - `cria_turma`: Um procedimento para criar uma nova turma, especificando o professor e a sala.

4. Seção `private`:
   - Dentro do `private`, a especificação do tipo `Turma` é detalhada.
   - `tam_max` é uma constante que define o tamanho máximo da turma.
   - `Turma` é definida como um tipo de registro (`record`), contendo:
     - `sala`: Um inteiro representando o número da sala.
     - `professor`: Um inteiro representando o identificador do professor.
     - `tam_classe`: Um inteiro limitado ao intervalo de 0 a `tam_max`, inicializado como 0.
     - `lista_classe`: Um array de tamanho `tam_max` contendo os identificadores dos alunos.

5. Corpo do Pacote (`package body tipo_turma is`):
   - Implementação dos procedimentos declarados no pacote.
   - `inclui_aluno`: Código para adicionar um aluno à turma.
   - `cria_turma`: Código para criar uma nova turma.

O encapsulamento em linguagens de programação, especialmente em linguagens orientadas a objetos como Java, Python, e C++, refere-se à prática de ocultar os detalhes internos de um objeto e permitir o acesso apenas a certas operações ou métodos definidos para esse objeto. Aqui está uma explicação mais detalhada em relação aos objetos e classes:

1. Objetos e Classes:
   - Uma classe é um modelo ou um plano de construção para criar objetos. Ela define os atributos (dados) e os métodos (operações) que os objetos de sua classe terão.
   - Um objeto é uma instância de uma classe. Ele contém dados (atributos) e métodos que operam nesses dados. Cada objeto é uma entidade independente com seu próprio conjunto de características e comportamentos.

2. Objetos Simples:

- No contexto de um módulo, um objeto pode ser uma variável oculta que é acessível apenas dentro do módulo, juntamente com as operações (funções ou métodos) que manipulam essa variável.
- Essa abstração permite que o módulo esconda os detalhes de implementação da variável e exponha apenas as operações relevantes para manipulá-la.
- Alterações na representação da variável (ou seja, sua estrutura interna) não afetam o código fora do módulo, desde que as operações expostas permaneçam consistentes.

3. Vantagens do Encapsulamento:
- Abstração: Os detalhes internos de um objeto são ocultados, permitindo que os usuários vejam apenas o que é necessário para usar o objeto corretamente.
- Segurança: O encapsulamento protege os dados dentro de um objeto, permitindo que apenas os métodos autorizados acessem e modifiquem esses dados.
- Modularidade: Ao dividir um programa em módulos encapsulados, é possível gerenciar e manter partes do código de forma independente, facilitando a colaboração e a manutenção do código.

4. Tempo de Vida do Objeto:
- Cada objeto tem um tempo de vida que geralmente é gerenciado automaticamente pelo sistema de execução da linguagem.
- O tempo de vida começa quando o objeto é criado (instanciado) e termina quando o objeto é destruído (coletado pelo coletor de lixo, em linguagens que possuem esse recurso).
- O tempo de vida de um objeto geralmente segue as mesmas regras que as variáveis locais em relação ao escopo e à vida útil.

Em resumo, o encapsulamento é uma prática fundamental na programação orientada a objetos, que promove a modularidade, a segurança e a reusabilidade do código, ao ocultar os detalhes internos dos objetos e expondo apenas as operações necessárias para interagir com eles.

Este código em Ada demonstra a utilização de um pacote genérico para definir classes de objetos similares, neste caso, relacionados a turmas de alunos. Vamos analisar cada parte:

1. Declaração do Pacote Genérico (`generic package tipo_turma is`):
- Define um pacote genérico chamado `tipo_turma`, que será parametrizado com tipos ou valores específicos quando for instanciado.

2. Parâmetros Genéricos:
- Neste exemplo, não há parâmetros genéricos explícitos, mas em pacotes genéricos mais complexos, poderiam ser fornecidos tipos ou valores que personalizam o comportamento do pacote para diferentes contextos de uso.

3. Declaração de Tipos e Procedimentos:
- Define um tipo `id_aluno` como um inteiro.
- Declara os procedimentos `inclui_aluno` e `cria_turma`, que serão usados para operar em instâncias de turmas.

4. Corpo do Pacote (`package body tipo_turma is`):
- No corpo do pacote, são fornecidas implementações específicas para os procedimentos declarados no pacote genérico.
- Define a constante `tam_max`, o tipo `Turma` e as implementações dos procedimentos `inclui_aluno` e `cria_turma`.

5. Tipo de Registro `Turma`:
- Define um tipo de registro chamado `Turma`, que representa uma turma de alunos. Ele possui campos para a sala, o professor, o tamanho da classe e uma lista de alunos.

6. Procedimentos Implementados:

- `inclui_aluno`: Este procedimento é responsável por incluir um aluno na turma. A implementação real não está presente no exemplo, mas provavelmente envolveria adicionar o aluno à lista de alunos da turma.
- `cria_turma`: Este procedimento é responsável por criar uma nova turma. A implementação real também não está presente no exemplo, mas provavelmente envolveria inicializar os campos da turma, como sala e professor.

No geral, este exemplo em Ada ilustra como usar um pacote genérico para definir classes de objetos similares (no caso, turmas de alunos) e como implementar as operações relacionadas a esses objetos no corpo do pacote.

A diferença entre um tipo abstrato e uma classe de objetos reside principalmente na forma como as operações são definidas e acessadas:

1. Tipos Abstratos:
   - Em tipos abstratos, as operações (funções e procedimentos) têm um parâmetro adicional, muitas vezes chamado de "self" ou "this", que é uma referência ao próprio objeto.
   - Ao chamar uma operação em um tipo abstrato, todos os argumentos, incluindo o objeto em si, precisam ser explicitamente passados.
   - Isso significa que para cada operação chamada, é necessário especificar o objeto em que a operação será realizada, tornando as chamadas de função mais explícitas e exigindo que o programador gerencie explicitamente o contexto do objeto.

2. Classes de Objetos:
   - Em classes de objetos, os procedimentos e funções são definidos dentro do escopo da classe e operam implicitamente nos dados do objeto.
   - Cada instância (objeto) da classe possui seu próprio conjunto de procedimentos e dados associados.
   - Ao chamar um procedimento ou função em um objeto, não é necessário especificar explicitamente o objeto como um parâmetro, pois o procedimento ou função já tem acesso implícito aos dados do objeto.
   - Isso simplifica a chamada de métodos, tornando o código mais conciso e geralmente mais fácil de entender.

3. Considerações Adicionais:
   - Tipos abstratos são frequentemente utilizados em linguagens como Ada e Modula-2, enquanto classes de objetos são uma característica central de linguagens orientadas a objetos como Java, C++, Python, e C#.
   - Tipos abstratos são mais semelhantes aos tipos de dados pré-definidos da linguagem, mas permitem ao programador definir novos tipos com comportamentos personalizados.
   - Classes de objetos oferecem um encapsulamento mais forte, pois os dados e operações relacionadas são agrupados em um único objeto, facilitando a manutenção e o reuso de código.

Em resumo, a principal diferença entre tipos abstratos e classes de objetos reside na forma como as operações são acessadas e no nível de encapsulamento proporcionado. Enquanto tipos abstratos exigem a passagem explícita do objeto como parâmetro em cada chamada de função, classes de objetos oferecem uma abordagem mais orientada a objetos, na qual os métodos operam implicitamente nos dados do objeto.

Neste exemplo em Ada, um pacote genérico chamado `tipo_pilha` é definido para criar abstrações genéricas sobre pilhas. Vou explicar cada parte:

1. Declaração do Pacote Genérico (`generic ... package tipo_pilha is`):
   - Define um pacote genérico chamado `tipo_pilha`, que é parametrizado por um tipo `Elem` e um valor `tam`.
   - `Elem` é um tipo genérico, cujo tipo real será especificado quando o pacote for instanciado.
   - `tam` é um parâmetro que define o tamanho da pilha.

2. Declaração de Tipos e Procedimentos:

- Define um tipo privado `Pilha`, que contém uma matriz de elementos do tipo `Elem` e um inteiro que representa o topo da pilha.
- Declara os procedimentos `empilha` e `desempilha` para operar em instâncias de pilha.

3. Corpo do Pacote (`package body tipo_pilha is`):
- No corpo do pacote, são fornecidas implementações específicas para os procedimentos declarados no pacote genérico.
- `empilha`: Implementa a operação de inclusão de um elemento na pilha.
- `desempilha`: Implementa a operação de retirada de um elemento da pilha.

4. Instanciação do Pacote Genérico:
- O pacote genérico pode ser instanciado para produzir pilhas de tipos específicos com tamanhos específicos.
- `Pilha_Inteiros` é uma instância do `tipo_pilha` que cria uma pilha de inteiros com tamanho 100.
- `Pilha_Turmas` é outra instância do `tipo_pilha` que cria uma pilha de instâncias da classe `Turma` (presumivelmente definida em outro lugar) com tamanho 60.

Dessa forma, o conceito de abstrações genéricas em Ada permite criar estruturas e algoritmos que funcionam de forma independente do tipo de dados específico que eles manipulam, facilitando a reutilização do código e aumentando a flexibilidade do sistema.