Universidade Federal do Amazonas
Faculdade de Tecnologia
Departamento de Eletrônica e Computação

Programação de Grandes Sistemas

Lucas Cordeiro
lucascordeiro@ufam.edu.br

Características de Sistemas de Tempo Real

- **Grande e Complexo** (variedade de atividades e requisitos que surgem a partir da interação com o mundo real)
- Controle concorrente dos componentes do sistema
- Facilidades para controlar o hadware
- Extremamente confiável e seguro
- Aparatos de tempo real
- Eficiência na execução

- Revisão do suporte da linguagem para o desenvolvimento de grandes sistemas
- Ilustrar o uso de módulos e pacotes para ajudar na **decomposição**, **abstração** e **compilação separada**
- Módulos e compilação separada em C (linguagens mais antigas, como Pascal, não suportam módulos)
- *Child packages* e POO em Ada 95
- POO em Java

Decomposição e Abstração

- **Decomposição** — A quebra sistemática de um sistema complexo em partes menores até que os componentes sejam isolados de uma tal maneira que possam ser entendidos e desenvolvidos por indivíduos ou grupos menores

  TOP DOWN DESIGN

- **Abstração** — Permite adiar a consideração detalhada sobre os componentes, mas ainda possibilita a especificação da parte essencial do componente
  BOTTOM UP DESIGN

Módulos

- Coleção de operações e objetos relacionados logicamente
- Encapsulamento — a técnica de isolar uma função do sistema dentro de um módulo com uma especificação precisa da interface (**estrutura estática**)
  - ocultação da informação (*information hiding*)
  - compilação separada
  - tipos de dados abstratos
- Ada e Java suportam módulos na forma de **package** enquanto que o suporte para módulo em C é **fraco**
- Como deveríamos decompor grandes sistemas em módulos?

A resposta para esta pergunta está no coração da engenharia de software!

Ocultação de Informação

- Em linguagens simples, todas as variáveis permanentes devem ser globais (uso impróprio e erros existem)
  - se dois ou mais procedimentos desejam compartilhar dados então os dados devem ser visíveis as outras partes do programa
- Um **módulo** suporta **visibilidade reduzida** permitindo que informações sejam ocultadas dentro do seu corpo
- A especificação e corpo de um módulo podem ser dadas separadamente
  - idealmente, a especificação deve ser compilada sem o corpo ser escrito
- Em Ada, existe uma especificação de pacote e um corpo de pacote; relacionamento formal; erros causados pela falta do corpo são detectados em tempo de compilação

Ocultação de Informação

- Em C, os módulos não são tão formalizados
  - programadores usam um `.h` para especificar a interface e um `.c` para o corpo
  - nenhum relacionamento formal
  - erros são detectados no tempo de *link*
- Java, tem o conceito de **pacote**
  - não existe sintaxe da linguagem para representar a especificação e corpo de um pacote
  - um pacote é um diretório onde classes relacionadas são armazenadas
  - para adicionar uma classe ao diretório, simplesmente coloque o nome do pacote (nome do caminho) no início do arquivo fonte

Exemplo de uma Fila em Ada (1)

```
package queuemod is
  -- assuma que o tipo Element está no escopo
  function Empty return Boolean;
  procedure Insert (E : Element);
  procedure Remove (E : out Element);
end queuemod;
package body queuemod is
  type Queue_Node_T;          --declaração mais adiante
  type Queue_Node_Ptr_T is access Queue_Node_T;
  type Queue_Node_T is
    record
      Contents : Element;
      Next : Queue_Node_Ptr_T;
    end record;
 type Queue_T is
    record
      Front : Queue_Node_Ptr_T;
      Back  : Queue_Node_Ptr_T;
    end record;
```

Exemplo de uma Fila em Ada (2)

```
type Queue_Ptr_T is access Queue_T;
Q : Queue_Ptr_T;
procedure Create is
begin
  Q := new Queue_T;
  Q.Front := null;   --não necessariamente um ponteiro
  Q.Back := null;    --são sempre inicializados como null
end Create;
function Empty return Boolean is
begin
  return Q.Front = null;
end Empty;
procedure Insert(E : Element) is
  New_Node : Queue_Node_Ptr_T;
begin
  New_Node := new Queue_Node_T;
  New_Node.Contents := E;
  New_Node.Next := null;
  if Empty then
    Q.Front := New_Node;
```

Exemplo de uma Fila em Ada (3)

```
    else
      Q.Back.Next := New_Node;
    end if;
    Q.Back := New_Node;
  end Insert;
  procedure Remove(E : out Element) is
    Old_Node : Queue_Node_Ptr_T;
  begin
    Old_Node := Q.Front;
    E := Old_Node.Contents;
    Q.Front := Q.Front.Next;
    if Q.Front = null then
      Q.Back := null;
    end if;
  end Remove;
  begin
    Create;
  end queuemod;
```

Exemplo de uma Fila em Ada (4)

- Tanto a especificação quanto o corpo de um pacote devem ser colocados dentro da mesma parte declarativa, embora outras entidades possam ser definidas entre as duas partes
- Qualquer pacote pode ser usado se este estiver dentro do escopo
  - para reduzir a necessidade de **nomeação excessiva**, o comando **use** pode ser fornecido

```
declare
  use queuemod;
begin
  if not Empty then
    Remove(E);
  end if;
end;
```

Exemplo de uma Fila em C (1)

- Isto define a interface funcional para o módulo

```
/* assuma que element está no escopo */
int empty();
void insertE(element E);
void removeE(element *E);
```

- O usuário do módulo simplesmente usa este arquivo. O corpo do módulo é dado abaixo

```
#include "queuemod.h" /* faz a especificação do módulo */
                      /* visível para o corpo */
struct queue_node_t {
  element contents;
  struct queue_node_t *next;
};
```

Exemplo de uma Fila em C (2)

```
struct queue_t {
  struct queue_node_t *front;
  struct queue_node_t *back;
} *Q; /*Q é agora um ponteiro para a estrutura queue_t*/
void create() {
  Q = (struct queue_t *) malloc(sizeof(struct queue_t));
  Q->front = NULL;
  Q->back = NULL;
};
int empty() {
  return (Q->front == NULL);
}
void insertE(element E) {
  struct queue_node_t *new_node;
```

Exemplo de uma Fila em C (3)

```
void insertE(element E) {
  struct queue_node_t *new_node;
  new_node = (struct queue_node_t *) malloc(sizeof(struct
                  queue_node_t));
  new_node->contents = E;
  new_node->next = NULL;
  if (empty) {
    Q->front = new_node;
  else
    Q->back->next = new_node;
  };
  Q->back = new_node;
}
```

Exemplo de uma Fila em C (4)

```
void removeE(element *E) {
  struct queue_node_t *old_node;
  old_node = Q->front;
  *E = old_node->contents;
  Q->front = Q->front->next;
  if (Q->front == NULL) {
    Q->back = NULL;
  }
  free(old_node);
}
```

- Implemente uma pilha em Ada95 usando os conceitos de orientação a objetos vistos nestes slides. As operações de **PUSH** e **POP** devem demorar o tempo *O(1)*
  - assuma que o tipo *Element* seja do tipo inteiro

Compilação Separada (1)

- Se uma unidade de biblioteca X deseja ter acesso a qualquer outra unidade de biblioteca Y, então X deve indicar Y usando a cláusula **with** (ou **#include** em C)

```
package Disptacher is
   --novos objetos visíveis
end Dispacther
with Queuemod;
package body Dispatcher is
   --objetos escondidos
end Dispatcher
```

- Dentro do contexto de gerenciamento de projeto, especificações podem ser elaboradas por um engenheiro senior (**especificação representa uma interface**)
- Compilação separada suporta programação **bottom-up**

Compilação Separada (2)

- Ada também suporta um projeto **top-down** (‘*stub*’ pode ser criado usando **is separate**)

```
procedure Main is
   type Reading is …
   procedure Convert (R: Reading; Cv : Out
   Control_Value) is separate;
begin
   loop
      Input(Rd); Convert(Rd, Cv); Output(Cv);
   end loop;
end;
…
separate (Main)
procedure Convert (R: Reading; Cv : Out
   Control_Value) is
   --corpo do procdimento
end Convert;
```

Tipos de Dados Abstratos (1)

- Um módulo pode definir um **tipo** e as **operações** sobre o tipo
  - o tipo deve ser declarado e instâncias do tipo passadas como um parâmetro para a operação
- Os detalhes do tipo devem ser escondidos do ponto de vista do usuário
  - ocultação de informação
- Para assegurar que o usuário não tenha informações sobre os detalhes do tipo, este é ou definido para ser **private** (como em Ada) ou sempre é passado como um **ponteiro** (como em C)
  - uma declaração incompleta do tipo é dado em um arquivo .h

Tipos de Dados Abstratos (2)

- Em C, a interface para o módulo `queuemod` se tornaria:

```
typedef struct queue_t *queue_ptr_t;
queue_ptr_t create();
int empty(queue_ptr_t Q);
void insertE(queue_ptr_t Q, element E);
void removeE(queue_ptr_t Q, element *E);
```

- Ada permite que parte da implementação apareça na especificação, mas para ser acessada somente a partir do corpo do pacote

```
package Queuemod is
  type Queue is limited private;
  procedure Create (Q : in out Queue);
  function Empty (Q : Queue) return Boolean;
  procedure Insert (Q : in out Queue; E : Element);
  procedure Remove(Q : in out Queue; E : out Element);
```

Tipos de Dados Abstratos (3)

```
private
  --nenhuma das declarações estão externamente visíveis
  type Queuenode;
  type QueuePtr is acess Queuenode;
  type Queuenode is
    record
      Contents: Element;
      Next : Queueptr;
    end record;
  type Queue is
    record
      Front : Queueptr;
      Back  : Queueptr;
    end record;
  end Queuemod;
package body Queuemod is
  --essencialmente o mesmo que o código original
end Queuemod;
```

Tipos de Dados Abstratos (4)

```
package Complex_Arithmetic is
  type Complex is private;
  function "+" (X,Y : Complex) return Complex;
  function "-" (X,Y : Complex) return Complex;
  function "*" (X,Y : Complex) return Complex;
  function "/" (X,Y : Complex) return Complex;
  function Comp (A,B : Float) return Complex;
  function Real_Part (X : Complex) return Float;
  function Imag_Part (X : Complex) return Float;
private
  type Complex is
    record
      Real_Part : Float;
      Imag_Part : Float;
    end record;
end Complex_Arithmetic;
```

Programação Orientada a Objetos

- POO possui (diferentemente dos tipos de dados abstratos):
  - extensibilidade de tipo (herança)
  - inicialização de objeto automática (construtores)
  - finalização de objeto automática (destrutores)
  - despacho em tempo de execução das operações (polimofirsmo)

- Ada 95 suporta as “funcionalidades” acima através do uso de *tagged types* e *class-wide programming*

- Java suporta POO através do uso de classes

Polimofirsmo

```
public abstract class OperacaoMatematica { public abstract
   double calcular(double x, double y); }
public class Soma extends OperacaoMatematica { public double
   calcular(double x, double y) { return x+y; } }
public class Subtracao extends OperacaoMatematica { public
   double calcular(double x, double y) { return x-y; } }
public class Contas {
    public static void mostrarCalculo(OperacaoMatematica
   operacao, double x, double y) {
        System.out.println("O resultado é: " +
   operacao.calcular(x, y));
    }
    public static void main(String args[]) {
         Contas.mostrarCalculo(new Soma(), 5, 5);
         Contas.mostrarCalculo(new Subtracao(), 5, 5);
    }
}
```

permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam

POO e Ada (1)

- Baseado em extensões de tipo (tagged types) e polimofirsmo dinâmico (class-wide types)
- Exemplos de *tagged types*

```
type Coordinates is tagged
  record
    X : Float;   Y : Float;
  end record;
procedure Plot(P : Coordinates);
```

- Este tipo pode então ser estendido:

```
type Three_D is new Coordinates with
  record
    Z : Float;
  end record;
procedure Plot(P : Three_D); -- sobre-escreve Plot
Point : Three_D := (X => 1.0, Y => 1.0, Z => 0.0);
```

POO e Ada (2)

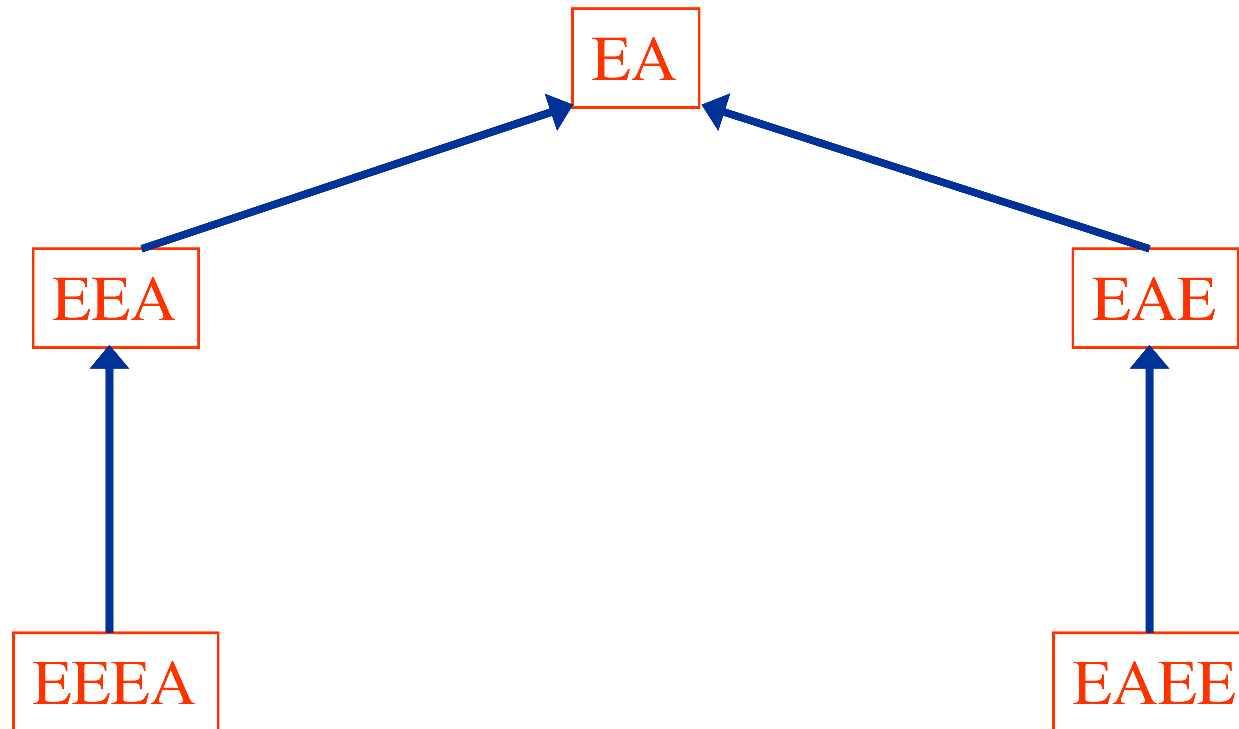
- Os campos da classe `Three_D` no exemplo acima estão diretamente visíveis para os usuários do tipo
- Ada permite que estes **atributos sejam completamente encapsulados** usando tipos **private**

```
package Coordinate_Class is
  type Coordinates is tagged private;
  procedure Plot(P: Coordinates);
  procedure Set_X(P: Coordinates; X: Float);
  function Get_X(P: Coordinates) return Float;
  --semelhantemente para Y
private
  type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Coordinate_class;
```

POO e Ada (3)

```
type A is record … end record;  -- normal record type
type EA is tagged record … end record; -- tagged type
procedure Op1(E : EA; Other_Param : Param);
  -- primitive operation
 procedure Op2(E : EA; Other_Param : Param);
  -- primitive operation
type EEA is new EA with record … end record; -- inherit EA
procedure Op2(E : EEA; Other_Param : Param); -- override Op2
procedure Op3(E : EEA; Other_Param : Param);
-- add new primitive operation
type EEEA is new EEA with record … end record;
...
type EAE is new EA with record … end record;
...
type EAEE is new EAE with record … end record;
...
```

POO e Ada (4)



Hierarquia de tipo que tem
origem em EA chamada EA'Class

Programação Class-wide (1)

- Ada é uma linguagem que considera **fortemente** os tipos dos dados
- Um mecanismo é necessário para que um objeto de qualquer membro da hierarquia possa ser passado como parâmetro

```
procedure Generic_Call(X : EA'Class) is
begin
  OP1(X,Param);
end Generic_Call;
```

- Um sub-programa pode receber uma coordenada como parâmetro sem se preocupar se esta coordenada é de 2D ou 3D

Programação Class-wide (2)

```
procedure General_Plot(P : Coordinates'Class);
```

- Qualquer chamada para uma operação de primitiva em um tipo **class-wide** resultará na operação correta para o tipo atual sendo chamado

```
procedure General_Plot(P : Coordinate'Class) is
begin
  --faça algo
  Plot(P);
  --dependendo no valor atual de P, um dos procedimentos
  --de Plot definidos será chamado (isto é, 2D ou 3D)
```

Resultados “despachados” em tempo de execução

Pacotes Filho (1)

- Adiciona mais flexibilidade a facilidade de empacotamento de um único nível

```
package Coordinate_Class is
  type Coordinates is tagged private;

  procedure Plot(P: Coordinates);

  procedure Set_X(P: Coordinates; X: Float);
  function Get_X(P: Coordinates) return Float;
  -- similarly for Y
private
  type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Coordinate_Class;
```

Pacotes Filho (2)

```
package Coordinate_Class.Three_D is
-- "." indicates that Three_D is a child of Coordinate_Class
  type Three_D is new Coordinates with private;

  -- new primitive operations
  procedure Set_Z(P: Coordinates; Z: Float);
  function Get_Z(P: Coordinates) return Float;

  procedure Plot(P: Three_D); -- overrides the Plot subprogram

private

  type Three_D is new Coordinates with
  record
    Z : Float;
  end record;
end Coordinate_Class.Three_D;
```

- Permite acesso aos componentes privados dos pais sem usar a interface dos pais
- Reduz **re-compilação**

Tipos Controlados (1)

- Com estes tipos, é possível definir sub-programas que são chamados (automaticamente) quando objetos do tipo:
  - são criados (initialize)
  - deixam de existir (finalize)
  - são atribuídos um novo valor (adjust)
- Para ganhar acesso a estas funcionalidades, o tipo deve ser derivado a partir do `Controlled`, um tipo pré-definido declarado na biblioteca do pacote `Ada.Finalization`
- Este define procedimentos para `Initialize`, `Finalize` e `Adjust`
- Quando um tipo é derivado a partir do `Controlled`, estes procedimentos podem ser sobre-escritos

Tipos Controlados (2)

```
with Ada.Finalization; use Ada.Finalization;
package Tracked_Things is
  type Thing is new Controlled with
    record
      Identify_Number : Integer;
      … --other data
    end record;
  procedure Initialize(Object: in out Thing);
  procedure Adjust(Object: in out Thing);
  procedure Finalize(Object: in out Thing);
end Tracked_Things;
package body Tracked_Things is
  The_Counter : Integer := 0;
  Next_One : Integer := 1;
```

Tipos Controlados (3)

```
  procedure Initialize(Object: in out Thing) is
    begin
      The_Count := The_Count + 1;
      Object.Identify_Number := Next_One;
      Next_One := Next_One + 1;
    end Initialize;
  procedure Adjust(Object: in out Thing) is
    renames Initialize;
  procedure Finalize(Object: in out Thing) is
  begin
    The_Count := The_Count - 1;
  end Finalize
end Tracked_Things;
```

POO e Java

- Baseado na construção de classes
- Cada classe encapsula dados (variáveis de instância) e operações sobre os dados (métodos, incluindo métodos construtores e destrutores)
- Cada classe pode pertencer a um pacote
  - pode ser local para o pacote ou visível para outros pacotes (que neste caso é nomeado public)
- Outras classes modificadoras são abstract e final
- Semelhantemente, métodos e variáveis de instância possuem modificadores como sendo
  - public (visível fora da classe)
  - protected (visível somente dentro do pacote ou em uma sub-classe)
  - private (visível somente para a classe)

Exemplo de Java (1)

```
import somepackage.Element; // import element type
package queues; // package name
//if there is no named package, then the system assumes
//an unamed package
class QueueNode // class local to package
{
  Element data;
  QueueNode next;
}
//abstract: no objects can be created
//final: the class cannot be extended
public class Queue // class available from outside the package
{

  QueueNode front, back; // instance variables

  public Queue() // public constructor
  {
    front = null;
    back = null;
  }
```

Exemplo de Java (2)

```
  public void insert(Element E) //  visible method
  {
    QueueNode newNode = new QueueNode();
    newNode.data = E;  newNode.next = null;
    if(empty()) {front = newNode;}
    else { back.next = newNode; }
    back = newNode;
  }
  public Element remove() //visible method
  {
    if(!empty()) { Element tmpE = front.data;
      front = front.next; if(empty)) back = null; }
    // garbage collection will free up the QueueNode object
    return tmpE;
  }
  public boolean empty() // visible method
  { return (front == null); }
}
```

Herança e Java (1)

```
package coordinate;
public class Coordinate  // Java is case sensitive
{
  float X, Y;

  public Coordinate(float initial_X, float initial_Y) // constructor
  { X = initial_X;
    Y = initial_Y; }

  public void set(float F1, float F2)
  {  X = F1;
     Y = F2; }

  public float getX()
  { return X; }

  public float getY()
  { return Y; }

  public void plot() {
   // plot a two D point}
}
```

Herança e Java (2)

```
package coordinate;
public class ThreeDimension extends Coordinate {
  // subclass of Coordinate

  float Z; // new field

  public ThreeDimension(float initialX, float initialY,
         float initialZ) // constructor
  { super(initialX, initialY); // call superclass constructor
    Z = initialZ;
  }

  public void set(float F1, float F2, float F3) //new method
  {  super.set(F1, F2); // call superclass set
    Z = F3;
  }

  public float getZ() // new method
  { return Z;}

  public void plot() {//overridden method
   /* plot a three D point */}
}
```

Herança e Java (3)

- Diferente do Ada, todas as chamadas de método são despachantes

```
{
  Coordinate A = new Coordinate(0f, 0f);
  A.plot();
}
```

Plotaria uma coordenada de duas dimensões, onde

```
{
  Coordinate A = new Coordinate(0f, 0f);
  ThreeDimension B = new ThreeDimension(0f, 0f, 0f);

  A = B;
  A.plot();
}
```

Plotará uma coordenada de 3-D mesmo embora `A` fosse originalmente declarado para ser do tipo `Coordinate`. Isto acontece porque `A` e `B` são tipos de referências. Atribuindo `B` para `A` somente a referência tem sido alterada e não o objeto em si

A Classe do Objeto

- Todas as classes em Java são sub-classes implícitas da classe “raiz” `Object`

```
public class Object {
  ...
  public boolean equals(Object obj);
  // methods to support monitors
  public final void wait()throws IllegalMonitorStateException,
      InterruptedException;
  public final void wait(long millis)throws
     IllegalMonitorStateException, InterruptedException;
  //the maximum time to wait in millis + additional time in nanos
  public final void wait(long millis, int nanos) throws
     IllegalMonitorStateException, InterruptedException;
  public final void notify() throws IllegalMonitorStateException;
  public final void notifyAll() throws IllegalMonitorStateException;
  //override for finalization
  protected void finalize() //is only called when garbage collection
    throws Throwable; //is about to take place
  //there are no destructor methods as in, say, C++
}
```

Exercício (1)

- Crie um tipo de dado abstrato (TDA) chamado **Retângulo**. O TDA tem atributos **comprimento** e **largura**, cada um com valor default igual a 1. Ele tem funções "membro" que calculam o **perímetro** e a **área** do retângulo. Forneça as funções *set* e *get*, tanto para o comprimento como para a largura. A função *set* deve verificar se o comprimento e a largura são números de ponto flutuante maiores que 0.0 e menores que 20.0.

Exercício (2)

- Crie um TDA Retângulo mais sofisticada que a que você criou no exercício anterior. Este TDA armazena só as coordenadas cartesianas dos quatro cantos do retângulo. O construtor chama uma função *set* que recebe quatro conjuntos de coordenadas e verifica se cada uma delas está no primeiro quadrante sem que nenhum valor de coordenada *x* ou *y* seja maior que 20.0. A função *set* também especifica se as coordenadas fornecidas de fato especificam um retângulo. As funções membro calculam o **comprimento**, **largura**, **perímetro** e **área**. O comprimento é a maior das duas dimensões. Inclua uma função quadrado, que determina se um retângulo é um quadrado.

Reusabilidade

- Produção de software é um negócio caro
  - software parece sempre ser construído do zero
  - engenheiro de hardware usa componentes bem definidos e testados
- Programação **modular** e **orientada a objetos** fornecem os fundamentos para a reusabilidade de software
  - um módulo para ordenação de inteiros não pode ser usado para odernar reais ou registros
- Ada fornece o conceito de **módulo genéricos**
- C++ fornece o conceito de **templates**
- Java usa a noção de **interfaces**

Programação Genérica em Ada (1)

- Um **generic** é um template a partir do qual componentes podem ser instanciados
  - a instanciação especifica o tipo atual
- Uma medida de reusabilidade do **generic** pode ser derivada a partir das **restrições impostas pelos parâmetros**
  - se o tipo instanciado tem que ser um vetor **uni-dimensional** do tipo inteiro, então o **generic não é resusável**
  - se **qualquer tipo** pode ser usado na instanciação então um **alto nível de reuso** pode ser obtido
- Parâmetros genéricos são definidos de tal maneira que especifica quais operações são aplicadas a eles dentro de um corpo genérico

Programação Genérica em Ada (2)

- Em algumas situações, a lógica é independente do tipo

```
procedure Swap(X, Y: in out Float) is
  T :  Float;
begin
  T := X; X := Y; Y := T;
end;
```

- O mecanismo **generic** evita este tipo de problema

```
generic
  type Item is private;
procedure Exchange(X, Y: in out Item);
procedure Exchange(X, Y: in out Item) is
  T :  Item;
begin
  T := X; X := Y; Y := T;
end;
```

A especificação e o corpo devem ser separados

Programação Genérica em Ada (3)

```
generic
  type Element is private;
package Queuemod_Template is
  type Queue is limited private;
  procedure Create (Q : in out Queue);
  function Empty(Q : Queue) return Boolean;
  procedure Insert(Q : in out Queue; E : Element);
  procedure Remove(Q : in out Queue; E : out Element);
private
  type Queuenode;
  type Queueptr is acess Queuenode;
  type Queue is
    record
      Contents : Element;
      Next : Queueptr;
    end record;
  type Queue is
    record
      Front : Queueptr;
```

Limited private restringe as operações para somente aquelas declaradas na especificação do pacote

Programação Genérica em Ada (4)

```
      Back : Queueptr;
    end record;
end Queuemod_Template;

package body Queuemod_Template is
  --the same as before
end Queuemod_Template;
```

- Uma instanciação deste genérico cria o pacote verdadeiro:

```
declare
  package Integer_Queue is new Queuemod_Template(Integer);
  type Processid is
    record
      ...
    end record;
  package Process_Queues is new Queuemod_Template(Processid);
  Q1, Q2 : Integer_Queues.Queue;
  Pid : Process_Queues.Queue;
```

Programação Genérica em Ada (5)

```
package Process_Queues is new Queuemod_Template(Processid);
  Q1, Q2 : Integer_Queues.Queue;
  Pid : Process_Queues.Queue;
  P : Processid;
  use Integer_Queues;
  use Process_Queues;
begin
  Create(Q1);
  Create(Pid);
  ...
  Insert(Pid, P);
  ...
end;
```

Cada um destes pacotes define um tipo de dados abstrato para uma fila, mas os tipos de elementos são distintos

Interfaces em Java

- Interfaces em Java melhoram a **reusabilidade** do código
- Uma **interface** é uma forma especial de classe que **define** a especificação de um conjunto de **métodos** e **constantes**
- Elas são por definição **abstratas** assim nenhuma instância da interface pode ser declarada
- **Uma** ou **mais classes** podem **implementar uma interface**, e objetos implementando interfaces podem ser passados como argumentos para métodos definindo o parâmetro a ser do tipo da interface
- Interfaces permitem relacionamentos a serem construídos entre as classes fora da hierarquia da classe

Exemplo de Interface (1)

- Considere um algoritmo “genérico” para ordenar vetores de objetos
  - o que existe de comum entre todas as classes que podem ser ordenadas é que elas suportam um operador ‘<’ ou ‘>’
  - esta característica é então encapsulada em uma interface

```
package interfaceExamples;
//any class which implements the Ordered interface must
//compare its object ordering with the argument passed
public interface Ordered {
  boolean lessThan (Ordered O); //return whether it is less
}                                //than the object
```

- `lessThan` recebe como parâmetro qualquer objeto que implementa a interface `Ordered`

Exemplo de Interface (2)

```
import interfaceExamples.*;
class ComplexNumber implements Ordered {
  protected float realPart;
  protected float imagPart;

  public boolean lessThan(Ordered O) // interface implementation
  {
    ComplexNumber CN = (ComplexNumber) O; // cast the parameter

    if((realPart*realPart + imagPart*imagPart) <
        (CN.getReal()*CN.getReal() + CN.getImag()*CN.getImag()))
    { return true; }
    return false;
  }

  public ComplexNumber (float I, float J) // constructor
  { realPart = I; imagPart = J; }

  public float getReal() { return realPart;}

  public float getImag() { return imagPart; }

}
```

Exemplo de Interface (3)

```
package interfaceExamples;
public class ArraySort
{
  public static void sort (Ordered oa[], int size) //sort method
  {
    Ordered tmp;
    int pos;

    for (int i = 0; i < size - 1;  i++) {
      pos = i;
      for (int j = i + 1; j < size; j++) {
        if (oa[j].lessThan(oa[pos])) {
          pos = j;
        }
      }
      tmp = oa[pos];
      oa[pos] = oa[i];
      oa[i] = tmp;
    }
  }
```

A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência

Exemplo de Interface (4)

```
  public static Ordered largest(Ordered oa[], int size)
    // largest method
  {
    Ordered tmp;
    int pos;

    pos = 0;
    for (int i = 1; i < size; i++) { // assumes size >=1
      if (! oa[i].lessThan(oa[pos])) {
        pos = i;
      }
    }
    return oa[pos];
  }
}
```

Exemplo de Interface (5)

```
{

  ComplexNumber arrayComplex[] = { // say
                    new ComplexNumber(6f,1f),
                    new ComplexNumber(1f, 1f),
                    new ComplexNumber(3f,1f),
                    new ComplexNumber(1f, 0f),
                    new ComplexNumber(7f,1f),
                    new ComplexNumber(1f, 8f),
                    new ComplexNumber(10f,1f),
                    new ComplexNumber(1f, 7f)
  };
  // array unsorted
  ArraySort.sort(arrayComplex, 8);
  // array sorted
}
```

Tratamento de Exceção em C (1)

- Uma das formas mais primitivas é o **unusual return value** ou **error return**

```
if (functional_call(parameters) == AN_ERROR)
  /* error handling code */
} else {
  /* normal return code /*
}
```

- C/Real-Time POSIX possui uma variável inteira (`errno`) que é ajustada pelo sistema

```
#include <errno.h>
ret = function_call(parameters);
if (errno == AN_ERROR)
  /* error handling code */
} else {
  /* normal return code /*
}
```

Tratamento de Exceção em C (2)

- Um conjunto de macros pode ser definido para ajudar a estruturar o programa

```
#define NEW_EXCEPTION(name) …
/* code for declaring an exception */
#define BEGIN …
/* code for entering an exception domain */
#define EXCEPTION …
/* code for beginning exception handlers */
#define END …
/* code for leaving an exception domain */
#define RAISE(name) …
/* code for raising an exception */
#define WHEN(name) …
/* code for handler */
#define OTHERS …
/* code for catch all excpetion handlers*/
```

Tratamento de Exceção em C (3)

```
NEW_EXCEPTION(sensor_high);
NEW_EXCEPTION(sensor_low);
NEW_EXCEPTION(sensor_dead);
/* other declarations */

BEGIN
  /* statement which may cause the above exception */
  /* to be raised; for example */
  RAISE(sensor_high);

EXCEPTION
  WHEN(sensor_high)
  /* take some corrective action */
  WHEN(sensor_low)
  /* take some corrective action */
  when(OTHERS)
  /* sound an alarm */
END;
```

Tratamento de Exceção em Ada (1)

- Considere um sensor de temperatura com leituras no intervalo entre 0 e 100°C
  - se o valor calculado estiver fora desse intervalo, o sistema de suporte lança uma exceção *Constraint_Error*
  - o tratador associado habilita qualquer ação corretiva a ser executada

```
declare
  subtype Temperature is Integer range 0..100;
begin
  --read temperature sensor and calculate its value
exception
  --handler for Constraint_Error
end;
```

- A granularidade do bloco pode ser inadequada!

Tratamento de Exceção em Ada (2)

- Alguns dos seguintes cálculos podem lançar a exceção do tipo `Constraint_Error`

```
declare
  subtype Temperature is Integer range 0..100;
  subtype Pressure is Integer range 0..50;
  subtype Flow is Integer range 0..200;
begin
  --read temperature sensor and calculate its value
  --read pressure sensor and calculate its value
  --read flow sensor and calculate its value

  --adjust temperature, pressure, and value
  --according to requirements
exception
  --handler for Constraint_Error
end;
```

Tratamento de Exceção em Ada (3)

- Diminuir o tamanho do bloco e/ou aninhá-los

```
declare
   subtype Temperature is Integer range 0..100;
   subtype Pressure is Integer range 0..50;
   subtype Flow is Integer range 0..200;
begin
   begin
      --read temperature sensor and calculate its value
   exception
      --handler for Constraint_Error for temperature
   end;
   begin
      --read pressure sensor and calculate its value
   exception
      --handler for Constraint_Error for pressure
   end;
...
end;
```

Tratamento de Exceção em Ada (4)

- Ada fornece um nome de tratador **when others** para capturar exceções que não foram previamente listadas

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead: exception
  use Text_Io;
begin
  //statements that may cause exceptions
  exception
    when Sensor_High | Sensor_Low =>
      --take some corrective action
    when E: others =>
      Put(Exception_Name(E));
      Put_Line(" caugth. The following information is
available");
      Put_Line(Exception_Information(E));
      --sound an alarm
end;
```

Tipos de Exceção em Ada

- **Constraint_Error**: É lançado após uma tentativa de atribuir um valor fora do intervalo do objeto
  - quando um acesso a um vetor está fora dos limites
  - quando um ponteiro nulo é acessado
- **Storage_Error**: É lançado quando o alocador de memória não atende a uma demanda de armazenagem
  - devido as limitações físicas da máquina terem sido esgotadas
- **Program_Error**: É lançado após uma tentativa de chamada de um subprograma, para ativar uma tarefa
  - ou para a elaboração de uma instanciação genérica, se o corpo da unidade correspondente não foi ainda elaborado
- **Tasking_Error**: É lançado quando exceções surgem durante uma comunicação intra-tarefa

Um Exemplo Completo em Ada (1)

```
generic
  Size : Natural := 100;
  type Item is private;
package Stack is
  Stack_Full, Stack_Empty : exception;
  procedure Push (X: in Item);
  procedure Pop (X: out Item);
end Stack;


package body Stack is
  type Stack_Index is new Integer range 0..Size-1;
  type Stack_Array is array (Stack_Index) of Item;
  type Stack is
    record
      S : Stack_Array;
      Sp : Stack_Index := 0;
    end record;
  Stk : Stack;
```

Um Exemplo Completo em Ada (2)

```
  procedure Push(X : in Item) is
  begin
    if Stk.Sp = Stack_Index'Last then
      raise Stack_Full;
    end if;
    Stk.Sp := Stk.Sp + 1;
    Stk.S(Stk.Sp) := X;
  end Push;

  procedure Pop(X: out Item) is
  begin
    if Stk.Sp = Stack_Index'First then
      raise Stack_Empty;
    end if;
    X := Stk.S(Stk.Sp);
    Stk.Sp := Stk.Sp - 1;
  end Pop;
end Stack;
```

Um Exemplo Completo em Ada (3)

```
with Stack;
with Text_IO;
procedure Use_Stack is
  package Integer_Stack is new Stack(Item => Integer);
  X : Integer;
  use Integer_Stack;
begin
  ...
  Push(X);
  ...
  Pop(X);
  ...
exception
  when Stack_Full =>
    Text_Io. Put_Line("stack overflow!");
  when Stack_Empty =>
    Text_Io. Put_Line("stack underflow!");
end Use_Stack;
```

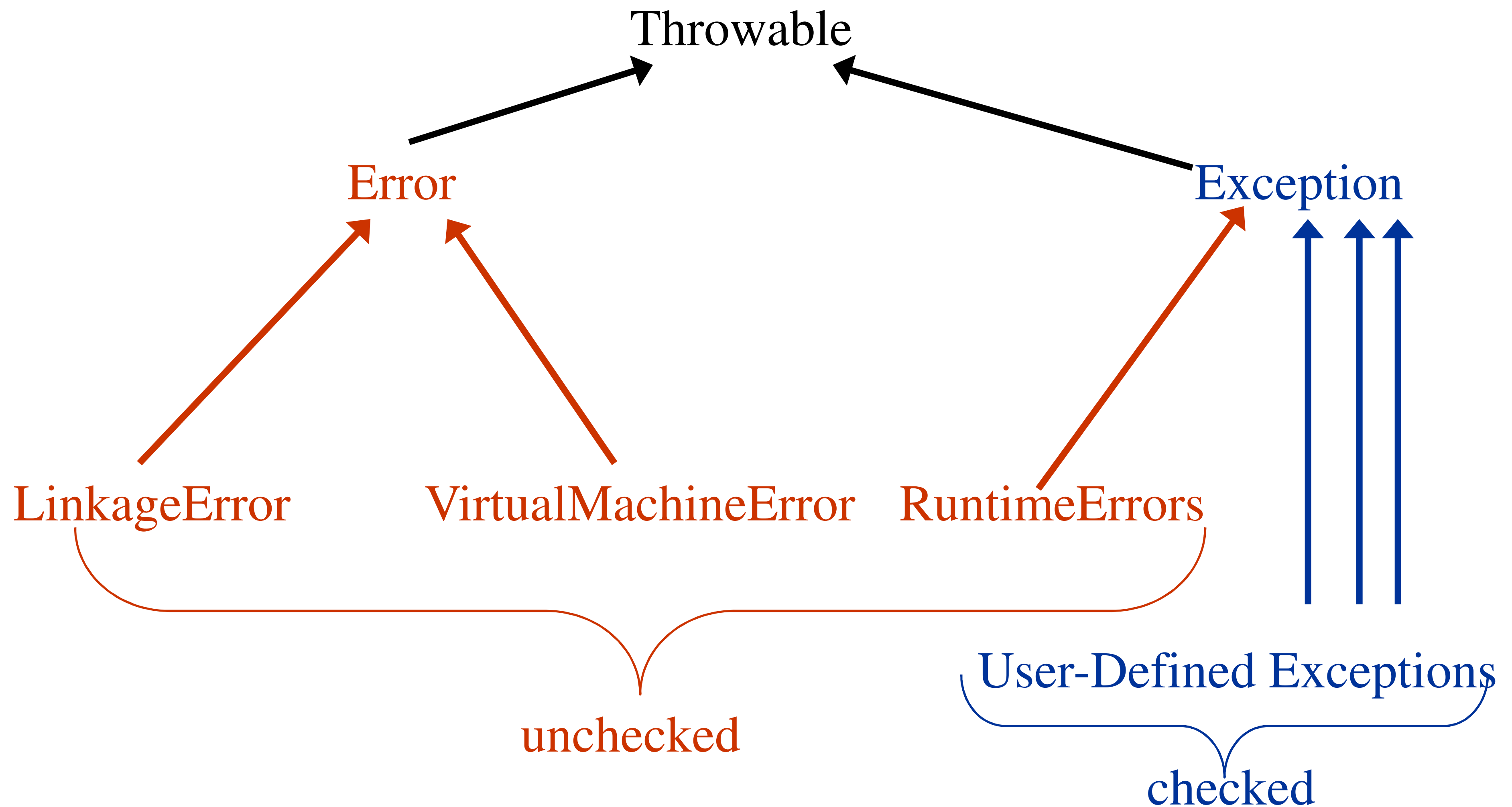
Tratamento de Exceção em Java

- Em Java, o domínio de um tratador de exceção deve ser explicitamente indicado e o bloco é considerado para ser guardado

```
try {
   //statements which may raise exceptions
}
catch (ExceptionType e) {
   //handler for e
}
```

- Todas as exceções são sub-classes da classe pré-definida `java.lang.Throwable`
  - A linguagem também define outras classes: `Error`, `Exception`, and `RuntimeException`

A Hierarquia da Classe `Throwable`



Tipos de Exceção em Java

- Objetos derivados a partir de `Error` descrevem erros internos e esgostamento de recursos
  - existe muito pouco que o programa possa fazer quando eles são lançados
  - nenhuma suposição pode ser feita a respeito da integridade do sistema
- Objetos derivados a partir da hierarquia de `Exception` representam erros que os programas podem tratar
- `RuntimeErrors` são aquelas exceções que são lançadas como resultado de um erro do programa
  - *bad cast* (`ClassCastException`)
  - erro de limites de vetor (`IndexOutOfBoundException`)
  - acesso de ponteiro nulo (`Null-PointerException`)

Exemplo em Java (1)

```
public class IntegerConstraintError extends Exception {
  private int lowerRange, upperRange, value;

  public IntegerConstraintError(int L, int U, int V) {
    super(); // call constructor on parent class
    lowerRange = L;
    upperRange = U;
    value = V;
  }

  public String getMessage() {
    return ("Integer Constraint Error: Lower Range " +
    java.lang.Integer.toString(lowerRange) + " Upper Range " +
    java.lang.Integer.toString(upperRange) + " Found " +
    java.lang.Integer.toString(value));
  }
}
```

Exemplo em Java (2)

```
import exceptionLibrary.IntegerConstraintError;

public class Temperature {
  private int T;

  public Temperature(int initial) throws IntegerConstraintError {
    // constructor
    ...;
  }

  public void setValue(int V) throws IntegerConstraintError {
    ...;
  }

  public int readValue() {
    return T;
  }

  // both the constructor and setValue can throw an
  // IntegerConstraintError
}
```

Exemplo em Java (3)

```
class ActuatorDead extends Exception  {
  public String getMessage()
  { return ("Actuator Dead");}
}
class TemperatureController  {
  public TemperatureController(int T)
         throws IntegerConstraintError {
    currentTemperature = new Temperature(T);
  }
  Temperature currentTemperature;
  public void setTemperature(int T)
         throws ActuatorDead, IntegerConstraintError {
     // check Actuator
     currentTemperature.setValue(T);
  }
  int readTemperature() {
    return currentTemperature.readValue();
  }
}
```

Lançando uma Exceção (1)

```
import exceptionLibrary.IntegerConstraintError;
class Temperature {
  int T;

  void check(int value) throws IntegerConstraintError {
    if(value  > 100 || value  < 0) {
      throw new IntegerConstraintError(0, 100, value);
    };
  }

  public Temperature(int initial) throws IntegerConstraintError
    // constructor
  { check(initial); T = initial; }

  public void setValue(int V) throws IntegerConstraintError
  { check(V); T = V; }

  public int readValue()
  { return T; }
}
```

Lançando uma Exceção (2)

```
// given TemperatureController TC

try {
  TemperatureController TC = new TemperatureController(20);

  TC.setTemperature(100);
  // statements which manipulate the temperature
}
catch (IntegerConstraintError error) {
  // exception caught, print error message on
  // the standard output
  System.out.println(error.getMessage());
}
catch (ActuatorDead error) {
  System.out.println(error.getMessage());
}
```

Catching All

```
try {
   // statements which might raise the exception
   // IntegerConstraintError or ActuatorDead
}
catch(Exception E) {
   // handler will catch all exceptions of
   // type exception and any derived type;
   // but from within the handler only the
   // methods of Exception are accessible
}
```

- Uma chamada para `E.getMessage` despachará a rotina apropriada para o tipo de objeto lançado
- **`catch`**`(Exception E)` é equivalente ao Ada's **`when others`**

Um Exemplo Completo em Java (1)

```
public class FullStackException extends Exception {
  public FullStackException () {
  }
}
public class EmptyStackException extends Exception {
  public EmptyStackException () {
  }
}
public class Stack {
  public Stack (int capacity) {
    stackArray = new Object(capacity);
    stackIndex = 0;
    stackCapacity = capacity;
  }
}
```

Um Exemplo Completo em Java (2)

```
public void push(Object item) throws FullStackException {
  if (stackIndex == stackCapacity)
           throw new FullStackException();
  stackArray[stackIndex++] = item;
}
public synchronized Object pop()throws
                            EmptyStackException {
  if (stackIndex == 0)
           throw new EmptyStackException();
  stackArray[--stackIndex] = item;
}

protected Object stackArray[];
protected int stackIndex;
protected int stackCapacity;
```

Um Exemplo Completo em Java (3)

```
public class UseStack {
  public static void main(...) {
    Stack S = new Stack(100);
    try {
      ...
      S.push(SomeObject);
      ...
      SomeObject = S.pop();
      ...
    }
    catch (FullStackException F) { ... }
    catch (EmptyStackException E) { ... }
  }
}
```

- Note que o Java fornece uma classe de `Stack` padrão no dentro do pacote `utils`

- Módulo suporta: ocultação de informação, compilação separada e tipos de dados abstratos
- **Ada** e **C** tem uma estrutura de **módulo estático**
- **C** informalmente suporta **módulos**; **Java** tem uma estrutura de módulo dinâmica chamada de **classe**
- **Pacotes** em Ada (e Java) e classes em Java possuem **especificações bem definidas** que atuam como a interface entre o módulo e o resto do programa
- **Compilação separada** possibilita que bibliotecas de **componentes pré-compilados** sejam construídos
- A **decomposição** de um sistema maior em módulos é a essência da programação de **grandes sistemas**

- O uso de **tipos de dados abstratos** ou **programação orientada a objetos** fornece uma das principais ferramentas que os programadores podem usar para **gerenciar sistemas maiores** de software
- Modelos de **tratamento de exceção** para processos sequênciais em C, Ada e Java