

Introduction to Ada - slides suitable for a 40-minute presentation
Copyright (C) 2004 Ludovic Brenta <ludovic.brenta@insalien.org>

This presentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This presentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

J'ai conçu ces transparents pour une présentation de 40 minutes que j'ai donnée lors des Rencontres Mondiales du Logiciel Libre, à Bordeaux, en 2004.

Ces transparents ne sont pas un support de cours pour enseigner le langage, ce qui est impossible à faire en seulement 40 minutes. Le but est de proposer un survol rapide des principales fonctionnalités d'Ada et de fournir des pointeurs vers de plus amples informations.

Prologue: Lady Ada Lovelace

★ Ada Augusta Byron, comtesse de Lovelace (1815-1852)

- Fille du poète Lord Byron
- Mathématicienne
- Assistante de l'inventeur Charles Babbage
- Première informaticienne de l'histoire
- Invente le premier "langage de programmation" pour la machine de Babbage



Historique

★ Lisp (1958) a été créé par et pour des chercheurs du MIT

★ C (1973) a été créé par et pour des “kernel hackers”

★ Ada a été créé par et pour des ingénieurs et des industriels

→ 1980 : première norme militaire MIL-STD-1815

- Tous les compilateurs doivent être validés (certifiés conformes)
- Pas de dialectes

→ 1983 : première norme civile ANSI (Ada 83)

→ 1987 : la norme devient ISO 8652

→ 1995 : premier langage orienté objet normalisé ISO (Ada 95)

→ 2005 : future norme Ada 2005

★ Trafic en forte hausse sur comp.lang.ada depuis 6 ans

Qui utilise Ada en 2004?

★ Aéronautique et spatial

- Eurofighter: 2 millions de lignes de code en Ada

- Boeing (en cours: 7E7)

- Airbus (en cours: A380, A400M)

- Ariane

- Satellites

- Eurocontrol (contrôle aérien)

- Leurs fournisseurs: Barco Avionique, Thalès, BAe Systems, Smiths Aerospace, Raytheon, etc.



Qui utilise Ada en 2004?

★ Industrie ferroviaire

- TGV

- Métros de New York, Paris (ligne 14), Delhi, Calcutta, etc.

★ Industrie nucléaire

- EDF : système d'arrêt d'urgence de réacteur

★ Autres

- Banque et finance : BNP (France), Paranon (Suisse), PostFinance

- Santé : JEOL (USA), ReadySoft (France)

- Automobile : BMW (Allemagne)

- Communications : Canal+ (France)

★ Et de nombreux logiciels libres!

Pourquoi Ada?

★ Dans l'industrie

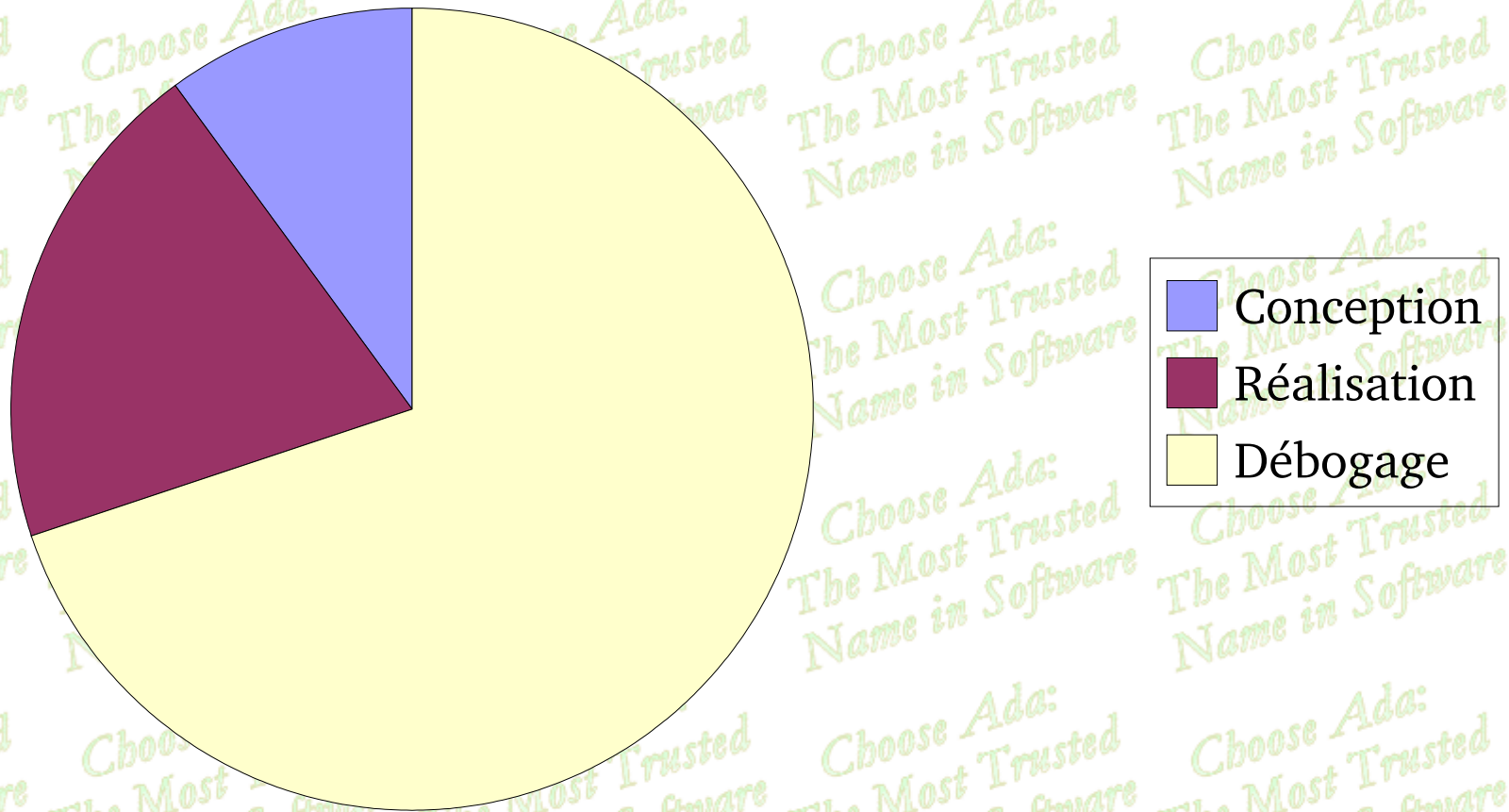
- Quand la vie humaine est en jeu
- Quand le logiciel *doit* fonctionner (aucune excuse acceptée)
- Quand on veut savoir *pourquoi* le logiciel fonctionne (certification)

★ Les logiciels libres

- Pour développer deux fois plus vite qu'en C
- Pour avoir 90% de bogues en moins par rapport au C
- Pour être proche du domaine d'application et éloigné de la machine (programmation de haut niveau)
- Pour que le logiciel soit portable
- Pour que le logiciel soit maintenable

Principes philosophiques (1)

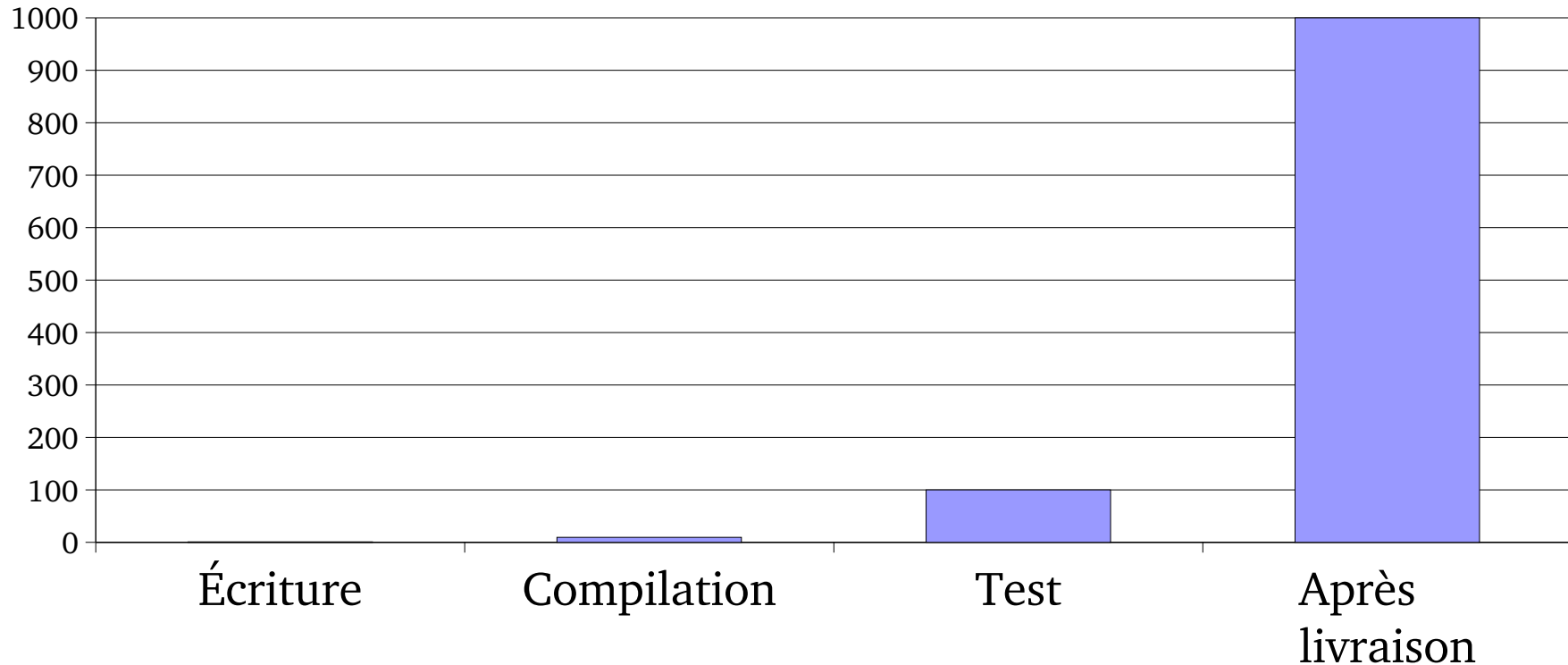
Coût de développement d'un logiciel



★ D'où l'idée géniale: le problème, ce sont les bogues

Principes philosophiques (2)

Effort requis pour corriger un bogue



★ D'où l'idée géniale: trouver les bogues le plus tôt possible!

Évident? Pas du tout!

Comme ça paraît, abonnez-vous à la [liste de diffusion](#) du monde Debian.

Annonces de sécurité

- [19 juin 2004] [DSA-524 rlpr](#) - Plusieurs failles
- [19 juin 2004] [DSA-523 www-sql](#) - Dépassement de tampon
- [19 juin 2004] [DSA-522 super](#) - Format de chaînes de caractères
- [18 juin 2004] [DSA-521 sup](#) - Format de chaînes de caractères
- [16 juin 2004] [DSA-520 krb5](#) - Dépassement de tampon
- [15 juin 2004] [DSA-519 cvs](#) - Plusieurs failles de sécurité
- [14 juin 2004] [DSA-518 kdelibs](#) - Entrée non vérifiée
- [10 juin 2004] [DSA-517 cvs](#) - Dépassement de tampon
- [7 juin 2004] [DSA-516 postgresql](#) - Dépassement de tampon
- [5 juin 2004] [DSA-515 lha](#) - Plusieurs failles de sécurité

Pour les annonces de sécurité, consultez la [page sécurité](#). Si vous souhaitez vous abonner à la [liste de diffusion debian-security-announce](#), rendez-vous à la [liste de diffusion](#).

Les buts du langage

★ Réduire les coûts de développement et de maintenance

- ➔ Empêcher d'écrire des bogues (mieux vaut prévenir que guérir)
- ➔ Détecter les autres bogues le plus tôt possible
- ➔ Favoriser la réutilisation du code et le travail en équipe
 - Paquets avec interface et implémentation séparés
 - Génériques (*templates*)
- ➔ Faciliter la maintenance
 - Langage très lisible et "auto-documenté"

★ S'adapter à toutes les situations

- ➔ *In the small*: embarqué, mémoire limitée, pas de système d'exploitation
- ➔ *In the large*: millions de lignes de code, réseau, fenêtres, etc.

Le système de types

★ Ada est très puissamment et *statiquement* typé

➔ Détecte la plupart des erreurs lors de la compilation, *avant de tester le programme!*

★ C'est le principal atout du langage

★ Le programmeur peut définir des types

➔ Pour refléter le domaine d'application

- Nombre_De_Torchons, Nombre_De_Serviettes, et pas simplement Integer

➔ Pour documenter les contraintes du problème

- Ne pas mélanger les torchons et les serviettes
- Le compilateur vérifie toutes ces contraintes

Les types scalaires

```
package Torchons_Et_Serviettes is
  type Nombre_De_Torchons is range 1 .. 20; -- entier
  type Nombre_De_Serviettes is range 1 .. 40; -- entier
  type Masse is digits 4 range 0.0 .. 4000.0; -- réel
  type Couleur is (Rouge, Vert, Bleu); -- énuméré
end Torchons_Et_Serviettes;
```

★ Les types scalaires ont des attributs :

→ T'First, T'Last : constantes

→ T'Range = T'First .. T'Last (intervalle des valeurs admises)

→ T'Pred(X), T'Succ(X) : fonctions qui retournent le précédent et le suivant

→ T'Image (X) : retourne X sous forme de chaîne de caractères

→ etc.

Les tableaux

```
package Tableaux is
  type Couleur is (Rouge, Vert, Bleu);
  type Valeur is range 0 .. 255;
  type Point is array (Couleur) of Valeur;
  -- L'index est un type énuméré

  type Largeur is range 0 .. 640;
  type Hauteur is range 0 .. 480;
  type Ecran is array (Largeur, Hauteur) of Point;
  -- Les index sont des intervalles
end Tableaux;

with Tableaux;
procedure Essai_Tableaux is
  Mon_Ecran : Tableaux.Ecran;
begin
  Mon_Ecran (4, 5) := (Tableaux.Rouge => 42,
                      Tableaux.Vert => 0,
                      Tableaux.Bleu => 35);
end Essai_Tableaux;
```

Les tableaux ont des attributs

★ Pour un tableau T:

→ T'Range est l'intervalle de l'index

- T1 : array (4 .. 8) of Integer;
- T1'Range = 4 .. 8;
- T2 : array (Couleur) of Integer;
- T2'Range = Bleu .. Rouge;

→ T'Range (N) est l'intervalle pour la Nième dimension

→ T'Length et T'Length (N) renvoient le nombre d'éléments

→ etc.

Les articles

```
with Ada.Calendar;  
package Articles is  
  type Type_Sexe is (Masculin, Feminin);  
  type Personne (Sexe : Type_Sexe) is record  
    Nom, Prenom : String (1 .. 100);  
    case Sexe is  
      when Masculin => null;  
      when Feminin =>  
        Nom_De_Jeune_Fille : String (1 .. 100);  
    end case;  
  end record;  
end Articles;
```

★ L'article ci-dessus a un *discriminant* (c'est facultatif)

Les pointeurs (types accès)

- ★ Les types accès sont incompatibles entre eux
 - ➔ pas de mélange possible entre pointeurs différents (sûreté)
- ★ Le compilateur garantit l'absence de pointeurs vers le vide
- ★ Les pointeurs sont utilisés seulement quand nécessaire
 - ➔ Structures de données dynamiques
 - ➔ Programmation objet et sélection dynamique des méthodes

```
with Articles; use Articles;
procedure Pointeurs is
  type Pointeur_Vers_Element;
  type Element_De_Liste is record
    P : Personne;
    Suivante : Pointeur_Vers_Element;
  end record;
  type Pointeur_Vers_Element is access Element_De_Liste;
  Tete_De_Liste : Pointeur_Vers_Element := new Element_De_Liste;
begin
  Tete_De_Liste.Suivante := new Element_De_Liste;
end Pointeurs;
```

La programmation objet

```
with Ada.Calendar;  
package Objets is  
  type Type_Sexe is (Masculin, Feminin);  
  
  type Personne (Sexe : Type_Sexe) is tagged record  
    Nom, Prenom : String (1 .. 100);  
    case Sexe is  
      when Masculin => null;  
      when Feminin =>  
        Nom_De_Jeune_Fille : String (1 .. 100);  
    end case;  
  end record;  
  
  type Diplome is record  
    Obtention : Ada.Calendar.Time;  
    Etablissement : String (1.. 100);  
    Titre : String (1 .. 200);  
  end record;  
  
  type Ensemble_De_Diplomes is array (Positive range <>) of Diplome;  
  
  type Personne_Diplomee (Sexe : Type_Sexe; Nombre_De_Diplomes : Positive) is  
    new Personne (Sexe) with record  
      Diplomes : Ensemble_De_Diplomes (1 .. Nombre_De_Diplomes);  
    end record;  
end Objets;
```

Conclusion sur les types

- ★ Ada permet de construire des types très complexes
- ★ Les types décrivent “l'univers” : c'est la *modélisation*
 - ➔ Haut niveau d'abstraction
 - ➔ Contraintes explicites
 - ➔ Auto-documentation
- ★ Les types et les objets ont des attributs
 - ➔ Les dimensions des tableaux sont toujours connues
 - ➔ Utile pour les boucles et tests (if, case)
- ★ Les pointeurs sont beaucoup plus sûrs qu'en C ou C++
- ★ Le compilateur vérifie toutes les contraintes
 - ➔ Statiquement quand il le peut
 - ➔ A l'exécution quand il le doit (avec des exceptions)

Sous-programmes

- ★ Procédures et fonctions comme en Pascal, sauf que:
 - ★ Paramètres “in”, “out” et “in out”
 - Le programmeur dit *ce qu'il veut faire*, pas *comment le faire*
 - Le compilateur décide de passer par valeur ou par référence
 - On peut passer de gros objets sans pointeurs explicites (tableaux, articles, etc.)
 - ★ Opérateurs surchargés
 - ★ Paramètres par défaut comme en C++
 - ★ Fonctions
 - Surchargées même par leur valeur de retour
 - Interdit d'ignorer la valeur de retour (pas d'ambiguïté, sûreté)
 - Tous les paramètres sont “in”

Sous-programmes : exemple

```
package Subprograms is
  function A return Boolean;
  function A return Integer;
  procedure A;

  type Type_Prive is private; -- un type de données abstrait;
  Null_Object : constant Type_Prive;
  function "+" (Left, Right : in Type_Prive) return Type_Prive; -- opérateur

  procedure A (T1 : in Type_Prive := Null_Object; -- paramètre par défaut
              T2 : in out Type_Prive);
  procedure A (T : out Type_Prive); -- constructeur
private
  type Type_Prive is record
    Valeur : Integer;
  end record;
  Null_Object : constant Type_Prive := (Valeur => 0);
end Subprograms;
```

Sous-programmes en POO

- ★ Une méthode est un sous-programme qui :
 - accepte un ou plusieurs paramètres du type étiqueté (*tagged type*) T ou un type accès vers T
 - est déclarée dans le même paquet que le type
 - procédure Foo (Objet : in T); -- méthode du type T
 - Pas de syntaxe particulière pour les méthodes en POO, “this” explicite

- ★ La sélection des méthodes peut être :
 - statique : décidée au moment de la compilation
 - dynamique : décidée à l'exécution (équivalent des méthodes virtuelles du C++), avec les types accès

- ★ Pour chaque type étiqueté T, il existe un type T'Class qui recouvre T et ses descendants

- procédure Foo (Objet : in T'Class); -- pas une méthode
- permet de forcer la sélection statique

Appel de sous-programmes

```
with Subprograms;  
procedure Use_Subprograms is  
  Object1, Object2 : Subprograms.Type_Prive;  
  B : Boolean := Subprograms.A;  
  I : Integer := Subprograms.A;  
use Subprograms;  
begin  
  A (Object1); -- constructeur  
  Object2 := Object1 + Null_Object; -- opérateur "+"  
  A (T1 => Object1, T2 => Object2);  
end Use_Subprograms;
```

Structures de contrôle

-- Boucles

```
procedure Control_Structures is
  type Couleur is (Rouge, Vert, Bleu);
  I : Natural := 0;

  function Foo (I : in Natural)
  is separate;

  Coul : Couleur := Foo (I);
begin
  for C in Couleur loop
    I := I + 1;
  end loop;

  while I > 1 loop
    I := I - 1;
  end loop;

  Funky_Loop : loop
    I := I + 1;
    exit Funky_Loop when I = 1000;
    I := I + 2;
  end loop Funky_Loop;
end Control_Structures;
```

-- Branchements

```
separate (Control_Structures)
function Foo (I : in Natural) return Couleur is
  Result : Couleur;
begin
  if I in 1 .. 10 then
    Result := Rouge;
  elsif I in 11 .. 20 then
    Result := Vert;
  elsif I in 21 .. 30 then
    Result := Bleu;
  end if;

  case I is
    when 1 .. 10 => Result := Rouge;
    when 11 .. 20 => Result := Vert;
    when 21 .. 30 => Result := Bleu;
    when others => Result := Rouge;
    -- tous les cas doivent être traités
  end case;

  return Result;
end Foo;
```

Exceptions

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Essai_Exceptions (I : in Natural) is
  Erreur : exception;
  type Couleur is (Rouge, Vert, Bleu);
  Result : Couleur;
begin
  case I is
    when 1 .. 10 => Result := Rouge;
    when 11 .. 20 => Result := Vert;
    when 21 .. 30 => Result := Bleu;
    when others => raise Erreur;
  end case;
  exception
  when Erreur =>
    Put_Line ("Erreur : I ne représente pas une couleur");
end Essai_Exceptions;
```

Génériques (1)

★ Plus puissants que les *templates* du C++

★ Un générique peut être une procédure, une fonction ou un paquet

★ Les paramètres génériques peuvent être :

→ des types

→ des variables

→ des procédures

→ des fonctions

→ des paquets

★ Le générique peut imposer des restrictions sur les paramètres acceptés

Génériques (2)

generic

```
type Item_Type is private; -- Item_Type peut être n'importe quel type non limité
package JE.Stacks is
type Stack_Type is limited private;
procedure Push (Stack : in out Stack_Type;
Item : in Item_Type);
procedure Pop (Stack : in out Stack_Type;
Item : out Item_Type);
function Top (Stack : Stack_Type) return Item_Type;
function Size (Stack : Stack_Type) return Natural;
function Empty (Stack : Stack_Type) return Boolean;

Stack_Overflow, Stack_Underflow : exception;
private
-- to be dealt with later
end JE.Stacks;
```

Multitâche (1)

- ★ Intégré dans le langage, pas dans une librairie
- ★ Mots-clés *task*, *protected*, *accept*, *entry*, *abort*, etc.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Essai_Taches (Nombre_De_Taches : in Natural) is
  task type Tache_De_Fond;

  task body Tache_De_Fond is
  begin
    delay 1.0; -- 1 seconde
    Put_Line ("Je suis dans une tâche de fond");
  end Tache_De_Fond;

  type Indice_De_Tache is range 1 .. 10;
  Les_Taches : array (Indice_De_Tache) of Tache_De_Fond;
begin
  Put_Line ("La tâche principale démarre");
  Put_Line ("La tâche principale attend la fin des tâches de fond");
end Essai_Taches;
```

Multitâche (2) : objets protégés

- ★ Les *fonctions* sont réentrantes : appels multiples autorisés
- ★ Les *procédures* et les *entrées* ne sont pas réentrantes : un seul appel à la fois
- ★ Les entrées ont des *gardes* qui permettent de synchroniser les appels

```
protected type Shared_Stack_Type is
  procedure Push (Item : in Integer);
  entry Pop (Item : out Integer);
  function Top return Integer;
  function Size return Natural;
  function Empty return Boolean;
private
  package Int_Stacks is new JE.Stacks (Integer);
  Stack : Int_Stacks.Stack_Type;
end Shared_Stack_Type;
```

Multitâche (3): gardes

```
protected body Shared_Stack_Type is
  procedure Push (Item : in Integer) is
  begin
    Int_Stacks.Push (Stack,Item);
  end Push;

  entry Pop (Item : out Integer)
  when not Int_Stacks.Empty (Stack) is
  begin
    Int_Stacks.Pop (Stack,Item);
  end Pop;

  function Top return Integer is
  begin
    return Int_Stacks.Top (Stack);
  end Top;

  function Size return Natural is
  begin
    return Int_Stacks.Size (Stack);
  end Size;

  function Empty return Boolean is
  begin
    return Int_Stacks.Empty (Stack);
  end Empty;
end Shared_Stack_Type;
```

Autres fonctionnalités

★ Clauses de représentation

- ➔ Permettent la programmation sûre de très bas niveau

★ Compilation séparée définie dans le langage

- ➔ Le compilateur garantit la cohérence de l'exécutable final

- ➔ Pas besoin de Makefile

★ Mécanisme de rendez-vous en multitâche

★ Interface normalisée vers C, COBOL, FORTRAN

- ➔ Permet d'utiliser toutes les bibliothèques C (comme GTK+)

★ Objets contrôlés

★ ASIS (Ada Semantic Interface Specification)

- ➔ un programme peut inspecter un programme écrit en Ada

- ➔ interface de programmation de haut niveau avec le compilateur

- ➔ norme ISO

Encore d'autres fonctionnalités

★ Annexes spécialisées de la norme Ada 95

- Toutes sont prises en charge par GNAT
- Annexe C : Programmation système (interruptions, code machine)
- Annexe D : Systèmes temps-réel
- Annexe E : Systèmes distribués (appels de sous-programmes distants)
- Annexe F : Systèmes d'information (nombres en notation décimale)
- Annexe G : Calculs numériques (nombres complexes, performances garanties)
- Annexe H : Sûreté et sécurité (aide à la certification)

Plus d'information?

★ Sites web :

➔ Ada Information Clearinghouse : <http://www.adaic.com>

➔ ACT Europe Libre Software : <http://libre.act-europe.fr>

➔ Ada France : <http://www.ada-france.org>

★ Groupes de discussion

➔ comp.lang.ada, fr.comp.lang.ada

★ Nombreux cours, tutoriels et livres en ligne

➔ En français et en anglais

★ Merci à John English pour les exemples de multitâche

➔ *Ada 95, The Craft of Object-Oriented Programming*

➔ <http://www.it.bton.ac.uk/staff/je/adacraft>