Ada

- Objektově orientovaný jazyk
- Silná verifikace typů – preventivní přístup k omezení chyb udělaných programátorem
- Jazyk pro vojenské aplikace a kritické systémy
- Podporuje vysokoúrovňový synchronizační prostředek rendez-vous
  - bez inteligentního překladače může být generován zbytečně složitý kód
- K běhu programu se nepoužívá interpretr
- Překladače musejí projít testem na soulad se standardem jazyka
- Nepodporuje fibres
  - má v návrhu svůj vlastní model pro paralelní výpočty – přímo podporuje syntaxi pro zápis paralelně proveditelných akcí
  - bezpečnost
- Syntaxe je silně ovlivněna Pascalem
  - čitelnost kódu
- Původem z Pentagonu
- http://www.adaic.org/

```
with Ada.Text_IO;

procedure Hello is
begin
  Ada.Text_IO.Put_Line("Hello, world!");
end Hello;
```

Case Study – F22 Raptor
(http://www.faqs.org/docs/air/avf22.html)

Almost all electronics gear on board is integrated by two "Common Integrated Processors (CIP)", built by Hughes. The CIPs are breadbox-sized, and accommodate 66 plug-in modules. Both of the CIPs have 19 slots still open, and there is space in the aircraft for a third CIP, allowing still more room for future expansion.

The CIPs handle almost all the F-22's electronics functions. The CIPs provide a degree of self-test and reconfigurability that can keep the F-22 flying even with battle damage. Each CIP operates at 10.5 billion instructions per second and has 300 megabytes of memory. The Raptor is run by 2.5 million lines of software, with about 90% of it written in the Department of Defense's Ada language. Ada was not used for all the software because some functions required optimizations, and so waivers were granted.

Základní konstrukce Ady

- task

```
task [type] jméno is
   deklarace jmen komunikačních vstupů
end jméno;

task body jméno is
   lokální deklarace a příkazy
end jméno;
```

- terminate
  - ukončení tasku
- abort
  - násilní ukončení tasku

- rendez-vous
  - Prostředek pro synchronizaci úkolů (tasks)
  - Dva úkoly spolu komunikují pomocí rendez-vous
    - Meeting point, entry calls
  - Task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat

```
task Simple_Task is
  entry Start(Num : in Integer);
  entry Report(Num : out Integer);
end Simple_Task;

task body Simple_Task is
  Local_Num : Integer;
begin
  //čeká na vložení čísla – entry call
  accept Start(Num : in Integer) do
    Local_Num := Num;
  end Start;

  //normálně pokračuje v běhu
  Local_Num := Local_Num * 2;

  //čeká na vyzvednutí spočítané hodnoty
  accept Report(Num : out Integer) do
    Num := Local_Num;
  end Report;
end Simple_Task;
```

http://www.adaic.org/whyada/intro5.html

- task si může vynutit, viz uvedený příklad, v jakém pořadí se provedou jaké akce – entry calls
  - když by se neprovedlo start, výsledkem by byl deadlock tasku

- accept
  - klient zavolá server
  - server si převezme parametry
  - server provede výpočet, klient spí
  - server předá výsledky

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu
- pokud je třeba více instancí, je nutné přidat **type**

```
task type Simple_Task is
```

- task je vytvořen v okamžiku, kdy je nadeklarovaná instance
  - type Task_Pool is array(Positive range 1..10) of Simple_Task; - žádný task nebyl vytvořen

  - My_Pool : Task_Pool; - bylo vytvořeno 10 tasků

  - task type Radar_Track;
    type Radar_Track_Pointer is access Radar_Track;
    Current_Track : Radar_Track_Pointer;
    žádný task nebyl vytvořen

  - Current_Track := new Radar_Track;
    byl dynamicky vytvořen jeden task

- Select
  - Může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
  Local_Item := Item;
end Init;

loop
  select
    accept Stop;
      exit;
  or

    when podmínka = >//může i nemusí být
    accept Put(Item : in Integer) do
      Local_Item := Item;
    end Put;
    Local_Item := Local_Item * 2;

 else
   Put_Line("No entry call
             at this time");
 end select;

 delay 0.01;
end loop;
```

Based on http://www.adaic.org/whyada/intro5.html

- Protected Objects, Protected Types
  - Tasky mohou sdílet objekty
  - Objekt je instance typu – klíčové slovo **type**
  - Klíčové slovo protected zajistí exkluzivní přístup k chráněnému objektu
  - Jsou tři operace nad chráněnými objekty:
    - Procedury – mění stav objektu, aniž by pro to musela být splněna podmínka; musí mít exkluzivní přístup k objektu – starost překladače
    - Entry calls – stejné jako procedury, ale pro vykonání entry call je třeba navíc splnit podmínku
    - Funkce – pouze vrací stav a nic nemění a proto nemusí mít exkluzivní přístup k objektu

```
protected type Counting_Semaphore is
   entry Acquire;
   procedure Release;
   function Count return Natural;

private
   Holding_Count : Natural := 0;
end Counting_Semaphore;


protected body Counting_Semaphore is

   entry Acquire when Holding_Count < 5 is
   begin
      Holding_Count := Holding_Count + 1;
   end Acquire;

   procedure Release is
   begin
      if Holding_Count > 0 then
         Holding_Count := Holding_Count - 1;
      end if;
   end Release;

   function Count return Natural is
   begin
      return Holding_Count;
   end Count;

end Counting_Semaphore;
```

http://www.adaic.org/whyada/intro5.html

- Pokud nelze splnit podmínku pro entry call, lze místo toho provést nějakou jinou akci, např. vyvolat výjimku, než aby se volající task uspal

```
select
   Semaphore.Acquire;
else
   raise Resources_Blocked;
end select;
```

Rendez-vous/Entry Call v Javě

- Java nemá chráněné objekty ve smyslu Ady, ale pouze jeden globální zámek na objekt/statickou třídu
  - Chráněný objekt je mnohem více než javovský monitor
- Je třeba vytvořit EntryCall objekt, který bude spravovat frontu klientů – bude ji však používat i server
  - Je nutné použít wait a notify
  - Tím se zaručí nedeterministické chování, protože nelze se 100% spolehlivostí dopředu určit, kdy se vzbudí které vlákno – pořadí není garantované
  - Bylo by nutné kompletně přepsat mechanismus uspání a vzbuzení vlákna
    - Další problémy na obzoru, které souvisejí se samotnou implementací konkrétního JVM a k nimž nemusí existovat řešení, protože do plánovače JVM a správy paměti (chráněný objekt) se interpretovaný kód nedostane
      - Odvážní se mohou i tak pokusit – co kdyby;-)
- Rendez-vous je zajímavé hlavně ve spolupráci se select
  - Java nemá ekvivalent pro select Ady

Realizace Rendez-vous/Entry-call apod. obecně

- Každý vykonávaný programový kód v jakékoliv podobě se nakonec projeví jako sekvence strojových instrukcí, se kterou pracuje procesor
  - Je tak jen otázkou, co vše je třeba naprogramovat a zda to má stále smysl
    - Například v Javě by to mohlo znamenat napsání vlastního interpretru, Java-in-Java, který by podporoval dané vymoženosti.

    - V Javě by bylo možné vytvořit na straně serveru frontu volání, žádostí o entry call, kdy by každá z nich měla vlastní zámek, na kterém by se čekalo.

      Aby se zabránilo korupci dat, je ovšem nutné použít alespoň dva zámky (na straně klienta k jeho uspání a na straně serveru k manipulaci s frontou):
      - Vzniká možnost race condition díky pořadí uzamykání zámků– klient, server vs. server, klient => možnost deadlocku, protože vlákna jsou přepínána nedeterministicky
      - Lze zaručit voláním wait uprostřed kritické sekce na straně klienta => deadlock
      - Řešením by byla obdoba z WinAPI pro
        - PostMessage
        - WaitForMultipleObjects
        - CreateEvent
        - Java nemá vše, stejně jako pointry

Klient

```
entryCallParams = prepareEntryCall();
Server.enqueueEntryCall(entryCallParams);
 //synchronized call

//zde může dojít k přerušení plánovačem

/*se vzrůstajícím množstvím vláken
a zmenšující se dobou výpočtu entry-call
roste pravděpodobnost, že server zavolá
entryCallParams.Lock.notify(); dříve než
klient zavolá
entryCallParams.Lock.wait();*/

//klient počká, než to server vyřídí
entryCallParams.Lock.wait();
   //pokud už ovšem server stihnul
   //notify(), nastane deadlock vlákna
```

Server

```
//získáme další zpracovatelný entry-call
entryCallParams = popEntryCall();

//zpracujeme ho
processEntryCall(entryCallParams);

//a vzbudíme klienta
entryCallParams.Lock.notify();
```

- Co už je bezchybný kód a co jsou jen podmínky, za nichž se konkrétní plánovač JVM zachová stejně?

Nicméně konkrétně v Javě

- nebudeme-li vyžadovat chráněné objekty
- nebudeme-li vyžadovat akceptování entry-calls v jasně daném pořadí, jak lze nadefinovat v Adě
    - např. viz SimpleTask

- omezíme-li možnosti select a loop na jednu smyčku, ve které se budou jen akceptovat entry-calls podle pořadí, které určí plánovač
    - tj. žádné when, or a else

- pak by to šlo, ale
    - je třeba vymyslet vhodné synchronizační schéma
        - jeden zámek nestačí, mají-li volání parametry
            - fronta parametrů a klientů se může rozsynchronizovat – klienti (on leave) mohou být plánovačem vzbuzeni v jiném pořadí, než v jakém byly zpracovány parametry
                - => tj. žádné parametry, ale má to pak stále smysl a použití?

        - čistě synchronized řešení má předchozí problémy

    - od verze 1.5 se nabízí objekt Condition, nemá ale persistentní stav signaled => zase nic?
        - Byla by třeba obdoba SpinLock => zbytečně konzumuje strojový čas, problémy s rychlostí odezvy
        - Je to stále ještě žádoucí řešení?
            - Není přece jen lepší jiný návrh programu?