情報システム学
— プログラミング言語論 —

久野　靖*

1997.2.18

1　はじめに

□ 本講義の目的→計算機言語/プログラミング言語に関する見聞を広めて頂くこと。

- おまけ:　WWW 関係の話題。

□ なるべく興味を持って頂ける話題も盛り込むようにしました。

□ 一応、準備は多めにやりましたが、全部やらなくてもよい。

□ 質問はいつでもしてください (手をあげて!)。

2　計算機言語とは…

□ まず計算機言語とは何かというお話から。

- では計算機とは何をするもの?

□ プログラミング言語…計算機言語の一種。

□ 計算機言語ってどういう言語?　なぜ存在する?

2.1　計算機と人間の情報伝達

□ 計算機…情報を取り扱うための装置

□ 人間と計算機はどうやって情報をやりとりするか?

□ ハードウェア…入出力装置

- 入力→キーボード、マウス、マイク、…
- 出力→ディスプレイ、プリンタ、スピーカ、…

□ 具体的にどのような形で情報をやりとり?

2.2　人工言語

□ 人間どうしの情報交換→自然言語 (日本語、英語、…)

- 計算機とのやりとりには不向き
- あいまいさ、処理が複雑

□ 簡単な規則に基づく人工言語→計算機の処理に向く

- 計算機で取り扱うために作った人工言語→計算機言語

2.3　プログラミング言語

□ プログラムを書くための計算機言語

- プログラムとは?

□ プログラミング言語の用途?

- 計算機に読み込ませる
- 人間が読む (娯楽、仕事、…)
- 自分が考えるための手段→でも動かした方が嬉しい

2.4　その他の計算機言語

□ プログラミング言語でない計算機言語も多数ある

□ 楽譜記述言語

□ 図形記述言語

- プリンタ→ページ記述言語→ PostScript…プログラミング言語でもある。

□ 文書記述言語 (マークアップ言語)、データ記述言語

- SGML → HTML → XML

□ アプリケーションを動かし画面で見れば十分?

- 「処理」したいときは言語になっている方がよい。

*筑波大学大学院経営システム科学専攻

2.5 本節のまとめ

□ 計算機言語とは、計算機で扱うための人工言語

- でも人間が読み書きしてもよい。

□ 代表はプログラム言語だが、他の計算機言語もいろいろある。

3 計算機とプログラミングの黎明期

3.1 計算機以前

□ 「計算」は必要不可欠 (何のために?)

□ 筆算、そろばん、計算尺

□ 「計算職」→多数の人間が手分けして計算 (数表等)

3.2 機械式計算機

□ 歯車式計算機→たとえば Pascal(数学者) も作った

□ その後→手回し式計算機

- どういう原理か分かります?

□ 今でいえば「電卓」に相当。

- どういう計算をするかは人間が逐一指示。

3.3 Babbage の解析機関

□ Babbage --- 「計算の自動化」に取り組んだ先駆者

- Ada Augusta --- Babbage の弟子で「世界最初のプログラマ」

□ 階差機関 (1820's)

- 差分を計算する歯車計算機。ある程度作られ使われた。

□ 解析機関 (1830's)

- 「パンチカード」で計算順序を制御する巨大な歯車計算機。
- 設計のみで製作はされなかった (その作った人はいる)。

3.4 電子計算機

□ 第 2 次世界大戦の頃…

- 計算機は戦略的に重要なものとして力を入れて開発された
- ものによっては「極秘」扱い

□ 電子計算機の前にリレー計算機というのがあった

- リレーでは速度や信頼性に弱点がある

□ 電子計算機

- 機械的な動作箇所がない→高速、信頼性
- 素子→真空管、トランジスタ、…
- 日本独自の素子「パラメトロン」というものもある。

3.5 世界最初の電子計算機は?

□ ENIAC(1946, Eckert, Mauchly) というのが通説だったが…

□ アメリカの特許裁判によって ENIAC の特許は却下され、ABC マシン (Atanasoff) が先とされる。

- ただし ABC マシンは動作していない。

□ ENIAC は実用に使われたという点では重要

- ただし、プログラムは「パッチボード」による

□ その後、素子は急速に進歩

- 真空管→トランジスタ→ LSI → VLSI → 1 チップ CPU

3.6 プログラム内蔵方式

□ 計算機でプログラムはどこに格納されているか?

□ 計算機の中身=(CPU、主記憶)

- CPU --- 計算等の処理を行う
- 主記憶 --- データ、プログラムを保持←プログラム内蔵方式 (Von Neuman)

□ プログラム内蔵方式は何が嬉しいか?

- プログラムを自由に取り換えられる
- プログラムを加工するプログラムが作れる

3.7 機械語によるプログラミング

□ 計算機の主記憶上のプログラム=機械語

- 単なるビットの列 (0 と 1 の羅列)
- 作るのも読むのもすごく大変

□ 初期の計算機ではすべて機械語によるプログラミングだった

□ 計算機に押しボタンやランプがついていてそれを操作

- 主記憶にプログラムを書き込む
- 主記憶にデータを書き込んだり、データの状況を調べる
- 実行の様子を調べる

3.8 アセンブリ言語

□ 機械語では「1 ビットでも間違うと破滅」「読みづらい」

□ →命令や番地などに名前をつけて書き表すように (アセンブリ言語)

□ →これを機械語に変換するプログラムを「アセンブラ」という

```
.globl _main
        .type    _main,@function
_main:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        call ___main
        movl $0,-4(%ebp)
L2:
        cmpl $9999,-4(%ebp)
        jle L4
        jmp L3
        .align 2,0x90
L4:
        incl -4(%ebp)
        jmp L2
        .align 2,0x90
L3:
L1:
        leave
        ret
```

3.9 高水準言語

□ アセンブリ言語でもやはりプログラミングは繁雑

- CPU の種類が違うと命令が違うのでプログラムが融通できない
- 機種に依存したプログラミング言語→「低水準言語」

□ もっと高いレベルの (人間に読み書きしやすい) 言語

- 機種に依存しない言語→高水準言語
- 高水準言語を機械語に翻訳するプログラム→コンパイラ

```
main() {
  int i = 0;
  while(i < 10000) {
    i = i + 1;
  }
}
```

3.10 ハードウェアの進歩

□ CPU 速度の進歩

- 加算の頻度…毎秒 1 回 → 100 回 → 1000 回 → 100,000,000 回 →…

□ 主記憶容量の増大

- 語数…100 語 (36bit/語) → 1000 語 → 100K 語 → 1MB(8bit/byte) → 1GB →…

□ 機械語ではとてもプログラミングできない

□ 高水準言語とコンパイラを使うのに十分な能力

- もちろん、コンパイラを作成する技術の進歩もある

3.11 本節のまとめ

□ 計算機の進歩

- 機械→リレー→電子計算機
- 真空管→トランジスタ→ LSI → VLSI

□ プログラミングの進歩

- 機械語→アセンブリ言語→高水準言語

4 初期の高水準言語

4.1 Fortran

□ 世界最初の高水準言語 (Backus らによる)

- もとは「数式の変換」(FORmula TRANslation)

```
X = 2*A + b
↓
mov #2,%eax
imull -4(%ebp),%eax
movl -8(%ebp),%edx
addl %eax,%edx
movl %edx,-12(%ebp
```

□ 数値計算のための言語として急速に普及

- 「計算機の専門家でなくてもプログラムが書ける」
- 当時は計算機を使うとはプログラムを書くこと

- 計算機の主要な用途は計算をすること

```
      real a(100,100), b(100,100), c(100,100)
      integer i, j, k, n
      read(5, *) n
      write(6, 100) n
100   format('matrix size = ', i5)
      do 10 i = 1, n
10      read(5, *) (a(i,j), j = 1,n)
      do 11 i = 1, n
11      read(5, *) (b(i,j), j = 1,n)
      do 12 i = 1, n
        do 12 j = 1, n
          c(i,j) = 0.0
          do 12 k = 1, n
12          c(i,j) = c(i,j) + a(i,k)*b(k,j)
      do i = 1, n
        write(6, 101) (c(i,j), j = 1, n)
101     format(10f12.5)
      end do
      stop
      end
```

4.2 Algol60

□ Fotran を作った Backus らが、もっと整理された科学技術計算用言語として設計。

- ブロック構造を導入。
- 再帰手続きを導入。
- BNF(Backus Naur Form) を使って言語を定義

□ BNF の例:

```
<program> ::= <statement> | <program> <statement>
<statement> ::= read <variable> | print <variable>
              | <variable> = <expression>
<expression> ::= <term> | <term> + <expression>
              | <term> - <expression>
<term> ::= <variable> | <literal> | ( <expresion> )
```

4.3 COBOL

□ 最初の計算機→科学技術計算用だったが、まもなく事務計算用の計算機も売られるように。

- Fortran、Algol60 のような科学技術用言語→科学技術用マシン
- 事務計算用マシンでは? → アセンブラ →やはり高水準言語がほしい
- CODASYL 委員会が案をとりまとめてできたのが COBOL (common business oriented language)。

□ COBOL は事務計算の世界ではまだまだ多く使われている。

□ COBOL の特徴:

- 英語ふうの構文。「move zero to x.」「add x to y giving z.」
- フォーマット機能、入出力機能などが充実。「$10,000,000」
- 十進演算。
- やたら書き方が長々しい。

```
identifcation division.
program-id. a-sample.
environment division.
configuration section.
source-computer.  my-pc.
object-computer.  my-pc.
input-output section.
file-control.
    select datain assign to file1.
    select dataout assign to file2.
data division.
file secion.
fd infile label record omitted.
01 person.
  10 name    pic x(12).
  10 age     pic 999.
  10 weight  pic 999.
  10 money   pic $$,$$$,$$9.
...
```

4.4 Algol68

□ Algol60 の後継言語を制定しようとした。

- 多くの計算機科学者が提案を出して競った。
- その中でヨーロッパのグループが提案したものが Algol68 に。
- 結果的にはあまり普及しなかった。

□ Algol68 の特徴:

- 文脈依存文法による定義。
- 型の直交的な取り扱い。ポインタ、レコードなど。

4.5 PL/I

□ 科学技術計算用言語 (Fortran) と事務言語 (COBOL) が乖離しているのはよくないと考えた IBM が提唱、普及させようとした言語。

□ Fotran の機能と COBOL の機能を合わせたような言語。

- やたらと機能を取り入れればいいというものではないとの批判。
- 構文などは Fortran、COBOL の固定フォーマットより新しい。

```
test: procedure options(main);
  dcl (x, y) float decimal;
  get list (x, y);
  put skip list (x+y);
end test;
```

□ 一時期それなりに使われたがすたれた。

4.6 この節のまとめ

□ 最初は科学技術計算言語と事務計算言語が分離。

□ それぞれにおいて改良の試み。

□ 両者を等号→汎用言語の試み。

□ しかし結局、もっとも最初の Fortran と COBOL が残っている。

- ただし、両言語とも時代に合わせた改良は行われている。
- 両言語とも、新しい言語の出現によりマイナーになっている。

5 プログラミングパラダイム

□ パラダイム==「考え方の枠組/流儀」のようなもの。

- プログラムは「人間が考えるもの」だから「流儀」は当然ある。
- しかし最初のころは「機械語」だから人間が機械に合わせていた。
- その後、計算機の能力にゆとりが生まれると、人間の流儀に合わせて言語をデザインできるようになってきた。
- 現在では非常に多くの「プログラミングパラダイム」が。

5.1 手続き型言語 (命令型言語)

□ 最も古くからある (Fotran から)。

□ 変数、代入、1 文ずつの順次実行。

□ すべて、計算機ハードウェアに対応。

- 変数←主記憶上の番地
- 代入←主記憶へのデータの格納
- 順次実行←機械命令の順次実行
- if、ループ←分岐命令

5.2 記号処理言語

□ 記号 (symbol) →他と相互に区別できる「何か」。

□ リスト→記号やリストの並んだもの。

□ 記号処理言語→記号やリストをおもに扱う言語。Lisp が代表。

```
>(defun app (l m)
   (if (null l)
       m
       (cons (car l ) (app (cdr l) m))))
>(app '(a b c) '(d e))
(A B C D E)
>(defun rev (l)
   (if (null l)
       '()
       (app (rev (cdr l)) (list (car l)))))
>(rev '(a b c d e))
(E D C B A)
```

5.3 文字列処理言語

□ 文字列を取り扱うための言語

- SNOBOL、Icon などが代表的。
- 可変長文字列を動的に扱え、パターンマッチなどの強力な文字列操作を提供。
- ただし、手続き型言語で文字列を柔軟に扱うことが難しかった時代背景が。
- たとえば Fortran では文字列操作はやりにくいし、C などでも不便。
- 現在では Perl や tcl や Python などのスクリプト言語で扱うように…

5.4 関数型言語

□ 変数への代入機能を持たず、関数の呼び出しだけで計算を記述。

- ML、Haskell などが代表的。
- 動作が副作用を持たない→慣れている人にとっては扱いやすい (慣れていない人にとっては大変困る)
- 並列計算への適用性もある。
- Lisp も副作用を使わなければ関数型ぽいプログラム (先の例)。

5.5 論理型言語

□ 変数の単一化と節 (clause) の還元に基づいて計算を記述。

- Prolog が代表的。いろいろなバリエーションがある。

```
child(mary, john).
child(mary, susan).
child(susan, bob).
child(bob, dick).
child(bob, jean).
brothers(X,Y) :- child(Z,X), child(Z,Y).
grandson(X,Y) :- child(X,Z), child(Z,Y).
| ?- grandson(X,Y).
```

```
X = mary
Y = bob;
X = susan
Y = dick;
X = susan
Y = jean
no
| ?-
```

5.6 システム記述言語

□ OS(オペレーティングシステム) などの記述に使える言語

- BLISS、BCPL、C などが代表的。
- Fortran や COBOL では OS は書けないから。
- 何が違う? → アドレス操作、任意番地のアクセスなど

□ C 言語はもともと UNIX を記述するために作られたシステム記述言語。

- しかし現在では汎用の言語として広く使われている。
- しかしポインタ演算やポインタ操作のあたりはいかにもシステム記述言語らしい。

```
main() {
  int i;
  unsigned int *p = (unsigned int*)main;
  for(i = 0; i < 10; ++i)
    printf("%08x %08x\n", p+i, p[i]);
}
```

5.7 対話型言語

□ もともと、Fortran 等は「ずーっとコンパイルを待って、終ったらようやく動かせる」言語。

- これに対して、教育用などでは「ちょっと打ち込んですぐ動かし、駄目なら直してすぐ動かし」と反復したい。
- このため「教育用高速コンパイラ」も作られたが、1 行単位で動かせるわけではないい。
- 「1 行単位ですぐ動かせる」言語→「対話型言語」

□ 対話型言語の元祖→ BASIC。ダートマス大学で教育用に開発。

- Lisp、ML、Prolog などもそういう使い方ができる。
- 実行をちょっと止めて変数の様子を調べられるという点は便利。
- しかし近年は普通の言語でも、コンパイルが高速になり (単にマシンが速くなったから)、またデバッガも発達しているので差が縮まっている。

5.8 教育用言語

□ もともとは「コンパイルが速い」とか「教育しやすいように複雑な機能は省く」という考え方から。

□ Pascal は「教育用」として始まった。

- 「型」「データ構造」などの概念を整理し、ごちゃごちゃしたプログラムが書けないような制約を設け、なるべく「整合性のある」言語にしようと努力した。
- かえって不自由で美しくないという論もある。
- 現在では「役に立つ言語」で教育するのがいいという考えが主流。

5.9 スクリプト言語

□ 「きちんと大きなプログラムを書く」より「ちょっと書けばすぐ動きいろいろなことが簡単にできる」言語→「スクリプト言語」

- 文字列処理なども得意に
- コマンド言語のスクリプト言語化→シェルスクリプト
- フィルタ向けのスクリプト言語→ Awk
- 汎用のスクリプト言語→ Perl、tcl
- オブジェクト指向スクリプト言語→ Python、Ruby
- さまざまなアプリケーションに組み込める→ VBScript
- ブラウザ内部に組み込める→ JavaScript

5.10 この節のまとめ

□ 言語にはいろいろなパラダイムがある。

□ もっとも主流なのは手続き型言語で、その流派はいろいろある。

- システム記述言語、対話型言語なども多くが手続き型

□ 論理型、関数型、記号処理などはまた独自の文明になっている。

6 手続き型言語の発展

□ 手続き型言語がプログラミング言語ではやはり主流。

□ 手続き型言語も年とともに改良されている。

□ その経過を順を追って見ていこう。

6.1 GOTO 文

□ 初期の言語では制御構造 (ループや枝分かれ) を作るには GOTO(ないし類似の機能) を使うしかなかった。GOTO は機械語の分岐命令と同じなので最初からあったのはうなずける。

```
      integer max, k
      max = 0
103   read(5, *, end=100) k
      if(k-max) 101,101,102
102   max = k
101   goto 103
100   write(6, *) max
      stop
      end
```

□ しかし GOTO だとぐちゃぐちゃに飛び回るプログラム (スパゲティプログラム) を作ってしまいやすいので、現在では GOTO は「なくすべきもの」ということになっている。

6.2 サブルーチン

□ サブルーチンとは「呼ぶと、あるひとまとまりの動作を行い、終ったら呼んだところへ戻って来る」ような機構。

- これを使うことでプログラムを適度な「かたまり」に分けることができる。
- 言い替えれば、サブルーチンを作ることは「新しい (自分にとって都合のよい) 命令を増やすこと」に等しい。
- いちど作ってしまったサブルーチンは、その中がどうなっていたか覚えていなくても使うことができる (一度に考えることの範囲が少なくて済む)。
- 呼び出しごとにちょっとずつ違うところは「引数」(パラメタ) として渡すことで変更できる。

6.3 コルーチン

□ 高水準言語にはあまり見られないが、サブルーチンの親戚で「コルーチン」というものもある。

- サブルーチンでは「呼ぶ側」と「呼ばれる側」の区別ははっきりしている。
- しかしコルーチンでは「互いに相手がサブルーチンだと思って呼び合う」形になる。

```
coroutine A(y)      coroutine B(x)
do while ...        do while ...
  resume B(...)       resume A(...)
  ...                 resume A(...)
  resume B(...)       ...
  ...                 resume A(...)
end do              end do
```

6.4 構造化プログラミング

□ 1960 年代に、GOTO によるスパゲティプログラミングをやめるために、「整った構造だけを使ってプログラムを作ろう」という運動が起こった。これが「構造化プログラミング」。

- 具体的には、整った構造とは C でいうと「if-else」「while」「do-while」の 3 種類。要するに C で GOTO を使わずにプログラムする。
- ただし、GOTO を使わなければいい、という表面的なものでなく、きちんと考えて整理された構造のプログラムを作ろう、という運動。

□ これと関連して提唱されたのが「トップダウンプログラミング」。

- 具体的には、まずプログラム全体の動作をおおまかに記述し、それを順次詳細化していくことで最終的なプログラムを作成する、というやりかた。
- 実際には、ある程度大きなプログラムになってくるとトップダウンだけでは無理 (最初に決めたことが見込み違いで行き詰まる)。ボトムアップ (必要な部品を用意して組み立てて行く) も併用する必要。

6.5 強い型

□ 変数や式に「型」を対応させ、型が合わないとコンパイルできないような言語を「強い型の言語」という。

- PL/I などは型はあるが「できるだけ自動変換してくれる」方針なのであまり「強い型」らしくない。
- 型検査を厳密に行うようになった言語の代表→ Pascal。

□ 併せて、「配列」「レコード」などのデータ構造を「型」として統一的に扱うようになったのも Pascal のころから。Pascal の「型」とは…

- 基本型: 整数、実数、文字、範囲型、列挙型「type color = (red, yellow, blue)」
- レコード型: 「record x:real; y:1..100 end」
- 配列型: 「array[1..100] of color」
- 集合型: 「set of 1..100」
- ポインタ型:「↑ color」

□ 以後の言語ではデータ構造を型として統一して扱うことは常識になっている。しかし範囲型や列挙型はあまり採用されていない。

6.6 例外

□ エラーの取り扱い→「if(エラーがある) { 処理… }」→通常の流れとエラー処理の流れが混ざる→コードがごちゃごちゃに

□ 昔は「GOTO で飛び出して 1 箇所で処理する」という手があった

□ その代替として、「例外機構」を持つ言語が増えている (例: C++、Java)

```
try {
  ...
  ... エラーが発生し得る処理
  ...
} catch(NumberFormatException e) {
  ... そのエラーの処理 ...
}
... エラーがあってもなくてもここへ続く
```

□ 例外機構を使うと「正常値とエラー値を区別する」という問題もなくてすむ。たとえば「-1 はエラーの目印」という方法は、その関数が普通の値として-1 を返し得る場合には使えない。

6.7 モジュール化

□ プログラムが大きくなってくると、関数 (サブルーチン) では「かたまり」が小さすぎる→関数が多すぎてそれらの関係が把握できない。

□ また、広域変数 (関数の外側にある変数) も同様。

□ そこで、関連のある複数の関数/広域変数を「モジュール」としてまとめる。

□ モジュール機能を持つ言語→ Modula、Modula-2、Euclid、Ada など。

```
module STACK exports PUSH, POP, TOP;
  var A:array[1..MAXSTACK] of INT;
      CNT:INT := 0;
  procedure PUSH(i:INT);
    CNT := CNT + 1; A[CNT] := i
  end;
  ...
end STACK;
```

□ モジュールの外部からはモジュールの実現方法に依存するデータにはアクセスできない→安全であり、実現方法の変更が容易。

6.8 抽象データ型

□ 例えば上のモジュールではスタックは 1 個しかできない。多数のスタックが使いたい時は? 「スタック型」の変数が沢山定義できると嬉しい。

□ そもそも型とは?

- 「値の集合」説: たとえば整数なら「0、1、-1、2、-2、…」だが、もっと抽象度の高いものになると「値」が何を意味するか難しい。たとえば「人間型」の「値」とは?
- 「操作の集合」説: たとえばスタックは「中身は知らないがとにかく PUSH、POP ができて、最後に PUSH したものほど先に POP されてくる」→「抽象データ型」

□ 抽象データ型をサポートする言語→ CLU、Alphard

```
stack = cluster is create, push, pop, top
  rep = record[a:array[int], cnt:int]
  create = proc() returns(cvt)
    return(rep$[a:array[int]$[1..100:0], cnt:0])
  end create
  push = proc(r:cvt, i:int)
    r.cnt := r.cnt + 1; r.a[r.cnt] := i
  end push
  ...
end stack

a1:stack := stack$create()
stack$push(a1, ...)
```

6.9 型パラメタ

□ 前記の stack 型は整数専用だったが、何型のスタックでもコードはほとんど同じはず→「型」を「パラメタ」として指定できるとよい。

```
stack = cluster[T:type is create, push, ..
  rep = record[a:array[T], cnt:int]
  ...
  push = proc(r:ct, data:T)
    r.cnt := r.cnt + 1; r.a[r.cnt] := data
  end push
  ...
```

□ さらに、言語によってはパラメタ型が「このような操作を持つ」という条件を指定できる。

```
sortedstack = cluster[T:type] is ...
    where T has lt:proc(x,y:T) returns(bool)
  ...
  push = proc(r:cvt, data:T)
    ...
    while T$lt(data, r.a[i]) do...
```

□ オブジェクト指向言語だと、直接操作を指定する代りに「パラメタ型はこのクラスのサブクラス以下」という指定方法にできる。

6.10 オブジェクト指向

□ 「オブジェクト指向」とは…計算機で扱う対象をさまざまな「もの」(object) として扱う考えかた。

- もともとは「オブジェクト指向プログラミング言語」

- 現在では「ソフトウェア工学」「データベース」「システム構築ツール」などさまざまな分野で「オブジェクト指向」が適用されている。

□ たとえば、「温室の温度調節システム」を考える。

- 旧来の考え方→「センサーを見て、気温が下がってきたらヒーターを通電するが、温度が上がりすぎたらヒーターを切る」「気温が上がってきたら窓を開くが、下がってきたら閉める」など機能中心に考える→制御する要素や条件が複雑になるとごちゃごちゃになりやすい。
- オブジェクト指向→「気温センサ」「ヒーター」「窓開閉装置」などの「もの」を考える→「気温センサ」は温度が低いと「ヒーター」、高いと「窓」に注意を喚起→「ヒーター」は注意を喚起されると、定期的に「センサ」に温度を尋ね、一定以下だと通電、十分暖かいならヒーターを止めて仕事を終る→人間にとって考えやすく、適度な大きさに分けて考えられる。

6.11 オブジェクト指向言語

□ 「オブジェクト指向」を最初に始めた言語→ Simula('60 年代後半)

□ Smalltalk-80('80 年) →グラフィクスを駆使した革新的なシステム→世の中に「オブジェクト指向言語」を認知させた

□ Smalltalk-80 は閉じた環境で速度の点からも「お仕事」には使いにくかった→さまざまな既存の言語 (C、Lisp、…) にオブジェクト指向機能を追加する動き

□ その中で勝ち残った言語が使われる状況に。

- Lisp 属→ CLOS(CommonLisp Object System)
- C 属→ C++
- Smalltalk-80 ←計算機の能力向上→それなりに利用
- Java ← WWW ブームの落し子だが汎用のオブジェクト指向言語

6.12 並列性への要求

□ 計算能力に対する要求は際限がない→しかし、1 つの CPU の速度向上には限界が見えている ←光の速度等

□ 行列計算のような「多数のデータに同じ演算をする」場合には、その大量の計算を「流れ作業」で行う特殊な計算機 (ベクトルプロセサ) を使える (スーパーコンピュータ)。極めて特殊用途のマシン。

□ CPU の VLSI 化→ 1 個の CPU の値段は急速に下がっている→多数の CPU を搭載して速度を稼ぎたい→並列プログラミング

□ 原始的なやり方としては、多数の CPU でそれぞれプログラムを動かし、必要なら通信機能を使って協調動作→繁雑で大変

6.13 マルチスレッド

□ スレッド=「実行の流れ」。

- これまで→サブルーチンを呼ぶと実行はサブルーチンから戻るまで先へは行けない→すべて「一筆描き」。
- そこで、「実行の流れ」を新たに作り出してその新しい流れがサブルーチンを実行→サブルーチンの実行と本体の実行が「並列に」行える。N 個の CPU で N 個のスレッドをうまく動かすと計算が N 倍速くなる (実際にはそう簡単には行かない)。

```
t = thr_create(subr, ...);
... ←この部分の仕事が subr と並列に
thr_join(t);
```

□ 最近ではこのようなスレッド機能を使うための標準ライブラリを提供したり (C、C++) 言語の標準仕様としてスレッド機能を持たせる (Java) ものがある。

6.14 並列言語

□ しかし、「普通の (直列) 言語」+「スレッド」よりは「最初から並列実行」な言語の方が本来よいはず。

□ 並列言語に関する研究は昔からある。実験的言語なら多く作られている。

- Concurrent Pascal, PLisp, Concurrent Prolog, KL/1, ...
- 並列オブジェクト指向言語 (ABCL/1, Plasma, ...)

□ しかし「実用的な」「並列言語」の「普及」はまだこれから。

6.15 この節のまとめ

□ 手続き型プログラミング言語はさまざまな形を取って進化している。

□ それぞれ利点も欠点もある。人間が使う以上、どれか 1 つあれば他はいらない、というわけにはいかない。

□ しかし 1 つの流派の中でも着実に進化は起こっている。

□ まだまだ発展途上。

7 World Wide Web

□ 清水先生のリクエストにより、WWW と HTML 関係の内容も用意しました (時間があればということで)。

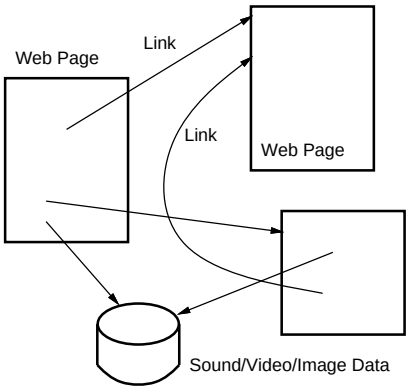
□ WWW(World Wide Web) は、インターネット上の情報システムとして急速に普及し、「インターネットブーム」の原動力となった。

- WWW 以前のインターネット→情報は各所にあるが、そのありかや参照方法を知らないとアクセスできなかった。
- WWW 以後→誰でも「point and click」で情報を入手できる。
- 情報システムのアーキテクチャとしても興味深い。

7.1 WWW の由来

□ CERN(欧州粒子物理研究所) の Tim Berners-Lee が考案 → その後、氏は WWW Consosium (W3C) を設立 → 現在は W3C は MIT/INRIA/Keio で維持。

□ ハイパーテキスト → 文書の中に、他の文書へのリンクが埋め込まれている → リンクを選択すると、そのリンクにつながっている文書にジャンプする



□ WWW は、ネットワーク中に「くもの巣 (Web)」のようにまたがったハイパーテキスト

□ 個々のページはネットワークの各所にある WWW サーバによって提供されている。

7.2 WWW と URL

□ ネットワーク上の資源はすべて URL(Uniform Resource Locators) によって指し示すことができる

```
プロトコル 所在
    ↓      ↓
news:<5315$fw27@gssm.otsuka.tsukuba.ac.jp>
mailto:kuno@gssm.otsuka.tsukuba.ac.jp
http://www.w3.org:80/hypertext/index.html
            ↑       ↑     ↑
         サーバ   ポート ディレクトリ+ファイル
```

□ この「どれでもあり」な URL が WWW のキーアイデアの 1 つ

7.3 WWW とブラウザ

□ WWW のページを見るためのツール → ブラウザ

- ブラウザの機能は原理的にはきわめて単純。
- ブラウザは URL を提示されると、その URL にあるサーバに接続してページを取り寄せて来て表示。
- ページ中から参照している画像等も同様に同じサーバから取り寄せて表示。
- リンクが選択されると、そのリンクの URL から新しいページを取り寄せ表示。

□ サーバは「要求されたコンテンツをそのまま返す」だけ。

7.4 ブラウザの歴史

□ 最初のブラウザ → テキストのみ表示 → あまりぱっとしなかった (その他の情報システムと同等)

□ NCSA Mosaic → 最初の「グラフィカル」ブラウザ → 突然、画像と文字が混在したカラフルなページが可能になった! → WWW の急速な利用拡大

□ 最初は NeXT や UNIX などの専門家むけシステムばかり → PC(Windows、Mac) でもブラウザが動くように → さらに急速な利用拡大

□ Mosaic を開発した学生がスピンオフして会社を → Netscape 社。Netscape Navigator はその高機能のため圧倒的に支持され → Mosaic を駆逐

□ Microsoft も Netscape の急成長を見て急拠参入 → MS Internet Explorer(MSIE)

□ ブラウザ戦争→現在も Netscape と MSIE の争いはあるが、どちらかというとポータルサイト戦争に…

7.5 WWW の発展

□ WWW の利用が急速に拡大した背景…

- 誰でも簡単にコンテンツを作って世界中に公開できる
- Web ページは HTML と呼ばれる言語で記述するがそう大変ではない
- 読み手から見れば…世界中の情報を居ながらにして入手。
- 使い方が極めて易しい。ポイント＆クリック。
- 容量に制限がない。情報が古くならない。タイムラグがない。(cf. CD-ROM)

7.6 WWW のマイナス面

□ もちろん、弱点も多くある

- 急速に「誰でもインターネット」→ネットワーク容量の不足。いくら待ってもページが取れてこない…
- 誰がどこで何をしているか制御不能 → ○○画像、不正確な情報、…
- インターネットは基本的に「公道」→ 盗聴、改ざん、なりすまし、泥棒、…
- 作るだけは作ったけれど放置されているページ…

7.7 本節のまとめ

□ WWW は個人での世界へ向けての情報発信を可能にした

□ 情報利用者にとってもこれまでにない新しい情報媒体

□ インターネットという共通のインフラのおかげ

□ WWW 自体もその後の新しい概念や技術のインフラ (イントラネットもその 1 つ)

□ では次に HTML によるページ記述について見てみましょう

8 WWW と HTML

8.1 WWW と HTML の関係は…

□ Web ページは基本的に HTML(HyperText Markup Language) で記述されている

□ HTML では、文章に「ここは表題」「ここは箇条書」といった印 (マークアップ) を付加することで構造を規定

- 簡単なページの例:

```
<HTML>
<HEAD><TITLE>Sample Page</TITLE></HEAD>
<BODY>
<H1>サンプルページだよーん</H1>
<UL>
<LI>なんか
<LI>かんか
<LI><A HREF="page14.html">戻る</A>
</UL>
</BODY></HTML>
```

8.2 リンクの記述

□ リンクも「ここがリンクで、選択されたらこの URL へジャンプ」と指定

□ リンクを選択すると、Web ブラウザは対応する URL の資源 (Web ページに限らない) を取り寄せて来る → それが HTML ファイルならそれを表示

□ Web ブラウザ (WWW を見るためのプログラム) は HTML ファイルを表示する際、画面やシステムの特性に合わせて整形する → どう見えるかはユーザの環境によって変化 (もちろんブラウザによっても変化)

□ 初心者が陥りやすい間違い: 「自分がこう見えているのだから他人もこう見えているに違いない」「Windows 98+MSIE5 を使ってない人は遅れている」

□ オープン、クロスプラットフォーム、というのが WWW の基本。

8.3 HTML と標準規格

□ HTML は SGML(Standard Generalized Markup Language) という国際規格に従って定義された言語

- SGML は「文書を格納/検索する共通基盤のための言語の枠組み」
- SGML に基づいて、言語定義 (Document Type Definition, DTD) →さまざまな言語が作り出せる

□ HTML の歴史

- HTML は簡潔で軽く、を目標にデザインされた
- HTML 1.0 → HTML 2.0 → (HTML 3.0) → HTML 3.2 → HTML 4.0
- HTML 2.0 はインターネット標準 (RFC1866)。HTML 3.2 と 4.0 は W3C 推奨規格。
- HTML 2.0 のころから、WWW が急速に「金のなる木」になったため…→ 各ブラウザメーカは競って「独自の新機能」を自社ブラウザに組み込み、それを標準にさせようとした → 標準になった方が勝ち → すなおに標準を作るのは結構難しくなってしまった

□ 今後は:

- 現在の方向: 「内容は定義するが、スタイルは定義しない」「スタイルはスタイルシート機能へ移行」
- では HTML 2.0 から見てみましょう

9 HTML 2.0

9.1 ドキュメントの基本構造

□ HTML 文書の基本構造は次のようになっている。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.0//EN">
<HTML>
<HEAD>
ヘッダ情報
</HEAD>
```

```
<BODY>
本体
</BODY>
</HTML>
```

□ DOCTYPE宣言は「どの版のHTMLか」を示すのに必要(省略するとHTML 2.0)。

□ ヘッダは「何の/どんなページか」を表す情報。典型的には

```
<TITLE>タイトル…</TITLE>
```

だけ入れておく。他にも入れられるものはいろいろあるが…

□ 本体はページの中身。

9.2 本体の要素

□ 本体の大域的な要素は…

- 段落(<P>… </P>)
- 箇条書き(<UL>… </UL>、<OL>… </OL>、<DL>… </DL>)
- 整形すみテキスト(<PRE>… </PRE>
- その他 … 引用<BLOCKQUOTE>、横線<HR>、改行
…

```
<P>HTMLでは段落の中は特に指定しない限り詰め合わされる。
特に改行をしたい場合は改行タグ<BR>を使う。もちろん、
</P>
<P>別の段落にすればそこでは行は変わる。</P>
<HR>
<P>横線を区切りに入れて見てもよい。</P>
<PRE>
    プログラムみたいに
    元のままの形にしたい場合は
    整形すみテキストを使う
</PRE>
<UL>
<LI>箇条書は
<LI>このように
<LI>項目を1つずつ列挙するのによい。
</UL>
```

9.3 段落内部などの要素

□ 「地の文」の部分には…

- 強調<EM>、強い強調<STRONG>、コード<CODE>など → 論理スタイル
- ボールド<B>、イタリック<I>、タイプライタフォント<TT>など → 物理スタイル
- イメージ<IMG SRC="イメージファイル">
- リンク<A HREF="url… ">リンクテキスト</A>

□ 使用例:

```
<P>文字を<EM>Enhance</EM>したり、
<STRONG>Strongly Enhance</STRONG>したり、
<CODE>i = 1;</CODE>のようにコード例をそれらしくしたりできる。<B>Bold</B>、<I>Italic</I>、Typefaceなどは物理的なスタイルなのであまり推奨されない。<IMG SRC="kuno.gif">のようにイメージを入れられる。<A HREF="kuno.gif">リンク</A>として入れてもよい。</P>
```

□ この他にフォーム機能があるが後で。

10 HTML 3.2

□ HTML 2.0では表現できないことが色々あるという不満 → 各ブラウザメーカの独自拡張 → W3CはHTML 3.0ドラフトを提唱 → 実装が難しい面がいろいろあった → 支持されず → 各ブラウザの拡張を後追い的に整理 → HTML 3.2

- テーブル機能が追加された → 非常に待ち望まれた方向
- 各種のスタイル制御機能が追加された → これらは本来のHTMLの思想からはズレている → 現在はこれらの機能を取り除く方向

10.1 テーブル機能

□ テーブル機能 … 「ここは表」「行の始め」「セル」を個別に指定していく。

```
<TABLE BORDER>
<TR><TH>Unix<TD>プロ向けだったが現在はDOS/V機で動く
<TR><TH>Macintosh<TD>GUIの先駆者だったが商売が下手で…
<TR><TH>Windows<TD>圧倒的多数派になったが嫌う人も…
</TABLE>
```

Unix	プロ向けだったが現在はDOS/V機で動く
Macintosh	GUIの先駆者だったが商売が下手で…
Windows	圧倒的多数派になったが嫌う人も…

10.2 スタイルの制御

□ とりあえず必要そうな各種のスタイル指定

- 中央揃え、右揃え(<DIV ALIGN=… >)、画像流し込み(<IMGALIGN=… >)
- フォントの大きさや色制御 … <FONT SIZE= COLOR=>
- 上つき、下つき … <SUP>、<SUB>
- 背景色、背景イメージ … <BODY BGCOLOR=… > <BODY BACKGROUND=… >

```
<DIV ALIGN=CENTER>
<P>このように中央に揃えることもできます。</P>
</DIV>
<P><IMG ALIGN=LEFT SRC="kuno.gif">
また、イメージに揃えを指定するとテキストをその
反対側に流し込むことができるようになります。</P>
<P>フォントは<FONT SIZE="+2">大きく</FONT>も、
<FONT SIZE="-2">小さく</FONT>も、また<SUP>上つき
</SUP>にも<SUB>下つき</SUB>にもできますし、
<FONT COLOR=RED>色つき</FONT>にもできます。</P>
```

10.3 背景などの指定

□ <BODY>の属性で背景色、または背景画像を指定できる。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML><HEAD><TITLE>bgcolor… </TITLE></HEAD>
<BODY BGCOLOR=BLUE>
<H1>青い背景</H1>
</BODY></HTML>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML><HEAD><TITLE>bgcolor… </TITLE></HEAD>
<BODY BACKGROUND="bgsample.gif">
<H1>模様のある背景</H1>
</BODY></HTML>
```

11 HTML 4.0

□ HTML 3.2 は「つなぎ」かつ「ブラウザメーカとの妥協」→ W3C 本来の思想を反映した新しい版としての HTML 4.0。

□ HTML 4.0 ドラフト： 1997.7 公開。1997.9 改訂版。そろそろ W3C 推奨規格になると思われる。

□ HTML 4.0 でのおもな変化

- フレーム機能の導入 --- これはブラウザの後追い
- 文字セットを UNICODE に → ようやく日本語等が「正式に」認められる
- <OBJECT>タグ → イメージ、アプレット、埋め込みオブジェクトなどが統一的に扱えるようになる
- スタイルシート機能の導入
- 入れ替わりに、HTML 3.2 までのスタイル関係機能の多くが「非推奨」に

11.1 フレーム機能

□ フレーム機能とは → 1 つの画面の中を複数の「枠」に分割してそれぞれを独立に (しかし関連させて) 制御する機能

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<HTML><HEAD><TITLE>Frame Demo</TITLE></HEAD>
<FRAMESET COLS="200,*">
<FRAME NAME="Left" SRC="r1-frame1.html">
<FRAME NAME="Right" SRC="r1-frame2.html">
</FRAMESET>
</HTML>
```

- これで画面を 2 つのフレーム Left、Right に分割。Left は 200 ピクセル、Right は残り。
- たとえば、Left に目次を表示させ、Right に対応するページを表示させる。その場合、Left では「<A HREF=… TARGET="Right">」と指定する (指定した名前のフレームにリンク先がロードされる)。

11.2 UNICODE 文字セット

□ HTML の文字セット…

- これまでは、あまりきちんと決められていなかった。たとえば文字セットは「普通は」(WWW が欧米製のため)ISO-Latain1 文字集合 (欧州語のアクセントくらいまで含めた文字) だけサポート。
- 日本語は… 文字集合は無視して JIS、EUC、SJIS をそのまま入れていた。「化ける」問題は「文字コード自動検出」で逃げる
- これからは、ヘッダ等で「これは ISO-2022-JP」などきちんと記述し、それに合わせた内容で送る
- UNICODE を使えば各国語の混在も可能 (日本でも UNICODE 嫌い! という人は多いが、既に国際規格だから…嫌いなことは提案して改良していく、という方向が…)
- 特別な文字も「名前や文字コードで」指定できる (これまでもできたがごく限られていた)。

11.3 OBJECT タグ

□ <OBJECT>タグとは…

- これまではイメージが表示できない場合は<IMG SRC="… " ALT="… "> で代わりに表示する文字列を指定していた。しかし ALT の文字列は長さが限られ、HTML のタグが入れられない。
- また、イメージの他にもアプレット、組み込みオブジェクトなど「埋め込む」ものは多数あって不統一だった。
- そこで、これらを統一的に扱う → <OBJECT>タグ

```
<OBJEFCT CLASSID="… ">  ← アプレットなど
  <OBJECT DATA="… ">  ← 代替
    すいません、イメージが表示できません ← 代替
  </OBJECT>
</OBJECT>
```

11.4 スタイルシート

□ スタイルシート: これまでのスタイル関係属性/タグの代替。

- 特定の表現に固定してしまうのは色々と不便 → スタイル関係の指定はすべて HTML から取り除こう、というのが W3C の立場
- 取り除いただけでは困る → 代わりに「スタイルシート言語」によってそれらを指定できる。
- スタイルシートをページと分離できるように → ページ作成者だけでなく、読み手も「こういう風に表示させて読みたい」と言える
- スタイルシート言語は複数ある → 特定のものに固定されないですむし、進歩が可能。現在は W3C が取りまとめた CSS2(Cascading Style Sheet Level 2) がおもに使われている。

□ スタイルシートの指定方法さまざま:

- タグの中で STYLE 属性を使って指定 → 個別の要素ごとに指定可

```
<P STYLE="color: blue">青い段落だぞー。</P>
```

- ヘッダの部分で<STYLE>… </STYLE>で指定 → 「すべての H1 は青」といった統一が可能。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<STYLE TYPE="text/css">
H1 { border-width: 3; border: solid;
     bolor-color: blue }
</STYLE>
<TITLE>Style Sheet Demo</TITLE>
</HEAD><BODY>
<H1>スタイルシートとは</H1>
<P>とは…</P>
<H1>その特徴</H1>
<P>とは…</P>
</BODY>
</HTML>
```

11.5 スタイルシートによる位置指定

□ 画面上で他の内容と独立に位置指定を行う機能。

- 各要素に position 属性を指定でき、これに absolute を指定した場合、画面上の絶対座標を (top と left の両属性で指定できる)。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<STYLE TYPE="text/css">
SPAN.kuno { position: absolute;
            top: 40px; left: 80px }
</STYLE>
<TITLE>Positioning Demo</TITLE>
</HEAD><BODY>
<H1><SPAN CLASS="kuno"><IMG SRC="kuno.gif"></SPAN>
スタイルシートによる位置指定</H1>
<P>どこでも好きなところに画像 (に限りませんが)
を動かせます。</P>
</BODY>
</HTML>
```

12 CGI

□ これまでのところ、WWW ページはいろいろな表現ができるが「固定されていて動かない」ものだった。

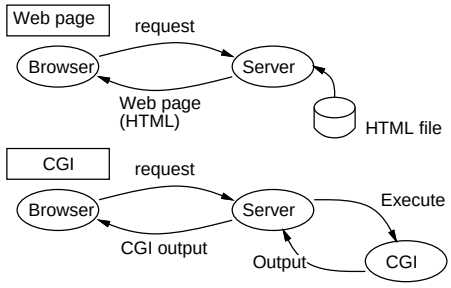
□ そうではなく、「対話的に動く」ようなページ (動的コンテンツ) を作りたい→さまざまな技術。

□ まずその最初のものが CGI

12.1 CGI とは?

□ CGI → Common Gateway Interface の略。そのココロは、Web サーバから下請けプログラムを呼び出す際の決まりごと。この呼び出されるプログラムのことを「CGI プログラム」と呼ぶ

- Web サーバとブラウザは HTTP(HyperText Transfer Protocol) で通信
- HTTP では、ブラウザとサーバがさまざまな情報を交換
- CGI では、プログラムを呼び出す際にブラウザからの情報を利用可能にする。逆にプログラムからデータに加えて各種の情報が返せる



12.2 CGI の原理

□ URL で指している先が CGI プログラムだと、そのプログラムが起動されてそれが返した結果がブラウザに返される。たとえば次のようにすれば好きなコマンドの出力を送り返せる

```
echo 'Content-type: text/plain'  ←MIME タイプを通知
echo ''                          ←ヘッダの終り
date
uptime
```

これを呼び出すには単にリンクでこれを格納したプログラムを指定すればよい:

```
<A HREF="html/s4-test0.cgi">テスト起動</A>
```

12.3 パラメタの渡し方

□ CGI へのパラメタ渡し → URL の一部が CGI に渡される。具体的には QUERY_STRING(「?」より先の部分) と PATH_INFO(CGI プログラム名と「?」にはさまれた部分がそう

```
echo 'Content-type: text/plain'  ←MIME タイプを通知
echo ''                          ←ヘッダの終り
echo $QUERY_STRING     ←以下テキストデータ
echo $QUERY_STRING
echo $PATH_INFO
echo $PATH_INFO
```

というようなシェルスクリプトを s4-test1.cgi というファイルに入れておいて次のようにすると、これらのパラメタつきでプログラムが呼び出せる:

```
<A HREF="html/s4-test1.cgi/ABC?DEF">テスト起動</A>
```

12.4 フォーム処理

□ CGI を普通のリンクから呼び出す代わりに FORM タグから呼び出すことができる → フォームのデータが URL 符合化と呼ばれる形に変換されて CGI に渡される → CGI ではフォームのデータを受け取って処理 → そこから先は他の CGI と同じ。たとえば 2 つの数を足す、というのをやってみる。

```
<FORM METHOD=POST ACTION="s4-test2.cgi">
<INPUT TYPE=TEXT NAME=A>た
す<INPUT TYPE=TEXT NAME=B>は?
<BR>
<INPUT TYPE=SUBMIT VALUE="計算!"></FORM>
```

これを処理する CGI は次のとおり。

```
PATH=/usr/local/bin:$PATH ← cgiparse を使うための設定
eval `cgiparse -form`      ← cgiparse で引数を受け取る
echo 'Content-type: text/plain'
echo ''
R=`expr $FORM_A + $FORM_B`     ←足し算する
echo "$FORM_A + $FORM_B = $R" ←結果の表示
```

12.5 CGI の位置付け

□ なぜ今さら CGI か? → CGI はサーバ上で好きなプログラムを起動することができる → 既にあるプログラムを WWW に組み込める

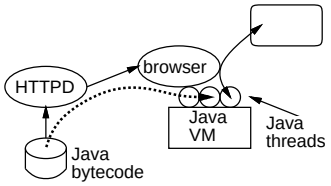
□ CGI の弱点 → 世界中のどこからでも呼び出され得る → セキュリティ対策をしっかりしないといけない

□ その他の CGI の可能性 → サーバ上のデータを複数人で共有するようなものなら何でも → たとえば我々はビジネスゲームのシステムを CGI で開発し授業に使っている

13 Java とアプレット

□ CGI では任意のプログラムが動かせるが、サーバ内で動くため、やりとりに時間が掛かる → アニメーションや対話的操作などには向いていない

□ そこへ出現したのが Sun の Java とアプレット。その基本的な考えかたは、ブラウザにインタプリタを組み込み、ダウンロードしてきたコードをブラウザの窓の「中で」実行させるというもの。



13.1 アプレットの例:

□ Java コード:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class MouseSensor3 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  int x0 = 100, y0 = 100, x1 = 150, y1 = 100;
  public void init() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x1 = e.getX(); y1 = e.getY(); repaint(10);
      }
    });
  }
  public void paint(Graphics g) {
    g.setFont(fn); g.setColor(Color.blue);
    g.drawString("Hello", x1, y1);
    g.drawLine(x0, y0, x1, y1);
  }
}
```

□ HTML タグ:

```
<APPLET WIDTH=300 HEIGHT=200 CODE="MouseSensor3.class">
</APPLET>
```

13.2 Java の歴史:

□ Sun Microsystems 社はワークステーションメーカーの老舗

□ 研究開発の一環として、家電組み込み情報システム用言語/システムというのをやっていた → Java 言語とインタプリタ

□ タイムワーナーの VOD 実験に応募して敗れる → プロジェクトは撤収 → チームの一人が HotJava ブラウザを開発 → 評判を呼び、Java ブームに

□ HotJava はすべて Java で書かれていて、Java インタプリタを内蔵しているので、画面の中で Java プログラムを動かすのは造作もなかった (しかしアイデアはなかなかだった)

13.3 Java の技術基盤

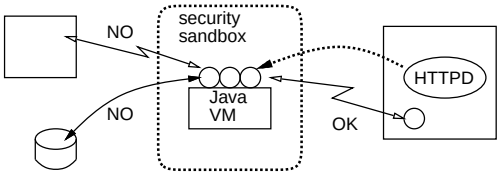
□ Java の技術的に興味ふかい点:

- プラットフォーム独立 → Java ソースコードは Java 仮想マシンコードへのコンパイルを行う → 仮想マシンがあればどのような計算機でも動く
- 実行性能の工夫 → その場コンパイラなどの高速化技術
- オブジェクト指向 → Java 言語はオブジェクト指向言語。C++を綺麗にした、という感じ
- 豊富な API --- Java ブームになったおかげで、さまざまな用途のためのライブラリ API が用意されるようになった。

13.4 Java とセキュリティ

□ インターネットがらみではセキュリティはとても重要

- セキュリティサンドボックス → アプレットは基本的に危険 (どこの何とも分からないコードがダウンロードされてきて動く) → アプレットにできる操作を制限した。ファイルの読み書き、プログラムの起動は一切禁止。ネットワーク接続はダウンロード元のサーバとのみ可能



- ディジタル署名、セキュリティポリシー → 署名や所在を通じて信頼できることが証明されたアプレットに対してはこの制限を緩められる。

13.5 Java の利点/欠点

□ Java でできること、できないこと…

- ○ 普通のプログラミング言語だから、普通にプログラムでできることは何でもできる (別にアプレットに限るわけではない)
- ○ アプレットは WWW のページに埋め込まれた知的なユーザインタフェースとして使うことができる
- ○ 豊富な API によってさまざまな仕事がこなせる
- ○ プラットフォーム独立。Write once, run everywhere
- △ 実行速度は JIT がないと遅い
- △ アプレットはセキュリティモデルによる制約がある

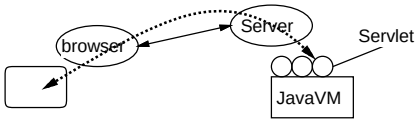
13.6 サーブレット: Java でサーバ側プログラミング

□ クライアント側プログラミングがあれば、CGI のようなものは不要か? →そんなことはない。共有される情報はサーバに置くしかない。

- しかし、CGI は起動されると独立したプログラムとして動き、すぐまま終わってしまうので、状態を保持しておくのが面倒
- 小刻みなやりとりにはオーバヘッドが大きい

□ このような問題に対して、Sun が提唱しているのが「サーブレット (Servlet)」…アプレットの反対 (?)

- サーバの中に、動的にロードできる。必要ならネットワーク経由でも
- サーブレットもセキュリティサンドボックスを使用
- 一度実行を始めると、サーバの中でずっと動いている
- クライアント (とくにアプレット) と簡単に通信できる
- これらの特性を活かして、複数のクライアント間で情報を共有するようなシステムが容易に作成できる



14 JavaScript

□ Java は簡単、とか言われてもやはりあくまでもプログラミング言語であり、プログラミングの素養のない人には書けない

□ そこで、もっと簡単に誰にでも書けるような、簡単な言語を→ スクリプト言語

□ Web サーバの中で動くスクリプト言語… Netscape Navigator の JavaScript (もとは LiveScript だったが、Java ブームのときに Sun から名前を買った)

- 構文的には Java に似ているが、クラスを作ったりしない。
- 実行は普通のインタプリタ方式
- できること…その場でページの内容を生成する、簡単なユーザインタフェースを作る、どこかへジャンプする、など

□ 簡単な計算をする例:

```
<SCRIPT TYPE="text/javascript">
function keisan() {
  fieldx = document.forms[0].elements[0];
  fieldy = document.forms[0].elements[1];
  fieldz = document.forms[0].elements[2];
  x = parseInt(fieldx.value);
  y = parseInt(fieldy.value);
  while(x != y)
    if(x > y) x -= y; else y -= x;
  fieldz.value = x;
}
</SCRIPT>
<FORM>
<P>X: <INPUT TYPE=TEXT VALUE=0 SIZE=8>
   Y: <INPUT TYPE=TEXT VALUE=0 SIZE=8>
 GCD: <INPUT TYPE=TEXT VALUE=0 SIZE=8><BR>
<INPUT TYPE=BUTTON VALUE="計算"
       ONCLICK="keisan()"></P></FORM>
```