クライアント/サーバー演習

東京電機大学工学部情報通信工学科
坂本直志

平成 21 年 2 月 20 日

目 次

1 予習

1. ノートパソコンで Java の開発キット (JDK) を入手し、利用できるようにしておいて下さい (付録 A(13 ページ) 参照)。
2. さらに Apache をインストールしておいて下さい (付録 B(13 ページ) 参照)。
3. 実験で使うプログラムの内容を読んで動作原理を理解しておいて下さい。

2 目的

インターネットにおけるクライアント/サーバモデルのネットワークシステムの実験を通して、ネットワークの仕組み、IP, TCP, UDP などに関して学ぶ。

3 原理

3.1 インターネットプロトコル (IP)

インターネットで使うプロトコルは *IP* (Internet Protocol [11]) と呼ばれています。このプロトコルが優れている点は、次のように制約条件が緩いからです。

1. どのような媒体で使用するか定められていない。伝書鳩 (RFC2549 [15]) から光ファイバ、無線など何でも使用できる。
2. パケット通信、つまり特定の長さのデータを特定の受信者に送れれば良い。
3. 相手にパケットが高い確率で届けば良い。100% の信頼性は保証しなくて良い。

IP ではアドレスをネットワークとホストを表す番号の組で表します。これを *IP* **アドレス**と言います。IP(version 4) の IP アドレスは 32 bit です。IP アドレスを表すのに 8 bit ずつ区切り、 4 組の 0 から 255 までの数の組をピリオドで区切ります。例えば、情報通信工学科のホストの IP アドレスには 133.20.160.1 というものがあります。IP アドレスは最上位のアドレスの値により 5 つのクラスに別れています。0 から 127 までは **クラス** *A* と呼ばれ、 0 から 127 がネットワークアドレスを表し、下 3 桁の 0.0.0 から 255.255.255 まではホストアドレスを表します。最上位のアドレスが 128 から 191 までは**クラス** *B* と呼ばれ、上位 2 桁の 128.0 から 191.255 までがネットワークアドレスで、下位 2 桁の 0.0 から 255.255 までがホストアドレスを表します。最上位のアドレスが 192 から 223 までは**クラス** *C* と呼ばれ、上位 3 桁の 192.0.0 から 223.255.255 までがネットワークアドレスで、下位 0 から 255 までがホストアドレスです。最上位のアドレスが 224 から 239 までは **クラス** *D* と呼ばれますが、これは特定のホストを表しません。クラス D のアドレスはマルチキャストと呼ばれる技術で各アドレスはチャネルという概念に対応します。最後に、最上位のアドレスが 240 から 255 までは**クラス** *E* と呼ばれてますが、この領域は予約領域で現在は利用されてません。

IP アドレスをネットワークアドレスとホストアドレスに分け、ホストアドレスを二進数で表した際、すべての桁が 0 の IP アドレスは特定のホストを指さず、**ネットワークアドレス**を表します。一方、ホストアドレスを二進数で表す時すべての桁が 1 の IP アドレスは**ブロードキャストアドレス**と呼ばれ、そのネットワーク全体のホストを示します。また、*0.0.0.0* はインターネット全体を表し、*127.0.0.1* は自分のホストを表します。例えば、東京電機大学工学部のネットワークアドレスはクラス B の 133.20.0.0 が割り当てられています。東京電機大学工学部すべてのホストは 133.20.255.255 で表せます。情報通信工学科のサーバは 133.20.160.1 という IP アドレスですが、これはネットワークアドレスが 133.20.0.0 で、ホストアドレスが 160.1 を表します。なお、 *255.255.255.255* は同一ネットワーク上のすべてのホストという意味を持ちます。

ネットワークアドレスは一般には一つの組織に対して一つ与えられます。したがって、 IP アドレスを原則通りに使用すると、一つの組織に一つしかネットワークを持つことができなくなります。そこで、組織内では IP アドレスを原則通りではなく、ネットワークを複数持てるように IP アドレスの解釈を変えて運用します。組織内で区分したネットワークを**サブネット**と言います。そして、ネットワークアドレスの解釈の変更を表すために、**サブネットマスク**と言う値を決めます。これは、IP アドレスを二進数で解釈した時、ネットワークアドレスとして解釈する桁の部分を 1、ホストアドレスの部分を 0 で示した値で表します。通常サブネットマスクの 1 の部分は連続するように決めるので、単純に 1 のビット数で表すこともあります。例えば、東京電機大学工学部の 11 号館系のネットワークではサブネットマスクは二進数で 11111111.11111111.11111111.00000000 であると定めています。これは十進数では 255.255.255.0 と表せます。このようにすると IP アドレスの上位 2 桁は 133.20 で固定されていますが、3 桁目の値はサブネットの番号を表せるようになります。したがって、例えば 133.20.160.1 というアドレスは 133.20.160.0 というネットワークアドレスと 1 というホストアドレスを持ちます。ところで、サブネットマスクを指定して運用している状態では、このネットワークアドレスとホストアドレスの区切りは自明ではありません。そこで、サブネットマスクと併記して 133.20.160.0/255.255.255.0 などと IP アドレスを表すこともあります。また、サブネットアドレスのビット数だけを併記して 133.20.160.0/24 などと表すこともあります。この場合、 133.20.160.0 サブネットのブロードキャストアドレスは 133.20.160.255 になります。なお、原則的なネットワークアドレスを示すサブネットマスク (クラス A: 8 bit, クラス B: 16 bit, クラス C: 24 bit) を**ナチュラルマスク**と呼ぶことがあります。

IP アドレスは国際機関 *ICANN*[7] で管理され、北米、ヨーロッパ、アジア、南米、アフリカを統括する各ネットワーク組織に分担され、さらに各国毎の組織に分担されます。例えば、日本の IP アドレスの管理は *JPNIC*[8] という組織で行っています。したがって、133.20.0.0 というネットワークアドレスもこの JPNIC から東京電機大学工学部へ割り当てられています。ところで、実験などを行うのに、その都度アドレスを管理組織に割り当ててもらうのは手間です。そこで、インターネットに直接接続して通信しない、個人的な目的で自由に使用するためのアドレスが決められています。これを**プライベートアドレス** [14] と言います。プライベートアドレスはクラス A, B, C にそれぞれ割り当てられています。**クラス** *A* は 10.0.0.0 一個、**クラス** *B* は 172.16.0.0 から 172.31.0.0 の 16 個、**クラス** *C* は 192.168.0.0 から 192.168.255.0 までの 256 個です。通常、家庭内 LAN などを使用する場合は、このプライベートアドレスを使用し、インターネット接続点でこれを通常の IP アドレス (**グローバルアドレス**) に書き換える技術 (*NAT*) が使用されています。

3.2 TCP, UDP

IP はインターネットの端から端までパケットを送ることができますが、信頼性はなく、また大きなデータをそのまま送ることもできません。*TCP*[12] は IP を利用して大きなデータを確実に送るためのプロトコルです。TCP は通信元と通信先を繋ぎ、送りたいデータを適当な長さに分割して IP を利用して送り、受信側に送信したデータが届いたことを確認して、もし送り損なっていたら再送するような手続きをとります。なお、TCP は同時に複数利用できるよう、**ポート番号**という番号が付けられ、そのポート番号毎に通信を管理します。ポート番号 1023 以下はシステムとして通信を行う内容が規定されていて、25 番は電子メール、 80 番は WWW となっています。このように特定のサービスにポート番号が対応しているため、**サービス番号**とも言います。また、 1024 から 49151 までは *IANA* に使い方が登録されています。例えば 8080 は WWW の予備のポート番号に割り当てられてます。49152 から 65535 までは各利用者が個人的に自由に使ったり、動的なポート番号として割り当てられています。

一方、*UDP*[10] は IP のパケットにそのままデータを組み込むものです。したがって IP パケットと同等の機能を持ち、そのため、信頼性が保証されてない通信を行います。UDP と TCP を対比させると、TCP では信頼性を保証するために再送などの処理を行います。それには、特定の相手と相互にパケットをやりとりしながら情報を通信し合う必要があります。そのため、常に一対一通信しかできず、また、やりとりによる手間が多いです。一方、 UDP は単純にパケットを相手先に向けて送るだけなので、受信側は情報を受ける以上のことはしません。したがって、信頼性は低いですが、情報伝達の負荷も低いです。さらに、TCP では 1 対 1 通信のみを想定していますが、 UDP ではブロードキャストアドレスを指定して送ることで、ネットワーク内のすべてのホストに対して情

報を同時に伝達することも可能になります。UDP はネットワーク管理やビデオのストリーミングなどに利用されています。

なお、 TCP を電話、 UDP を葉書に喩えて説明する本 [17] もあります。

3.3 IP アドレスとホスト名

我々がインターネットを利用する時、 IP アドレスを直接用いず、実際は www.c.dendai.ac.jp のようなホスト名で相手ホストを指定しています。しかし、これまで説明してきたように通信には IP アドレスを使用します。したがって、www.c.dendai.ac.jp というホスト名から 133.20.160.1 という IP アドレスを対応づける仕組みが必要になります。

これを行うのが**ネームサーバ**と呼ばれるサーバです。各コンピュータはネームサーバに対して www.c.dendai.ac.jp などのホスト名を使って IP アドレスを問い合わせます。もし、そのネームサーバが直接その IP アドレスを知っていればそのまま IP アドレスを送り返します。もし、 IP アドレスを知らない時は、そのネームサーバは他のネームサーバに問い合わせて調べます。まず、国際機関 ICANN[7] が統括している**ルートサーバ**と呼ばれる根幹となるネームサーバに問い合わせます。ルートサーバは jp や com などホスト名の一番下位の部分 (ドメイン) を管理しているネームサーバのアドレスを返します。例えば jp なら日本のネットワーク管理組織である *JPNIC*[8] のネームサーバのアドレスを返します。次に、得られたアドレスを元に、この jp を管理しているネームサーバに問い合わせます。このサーバでは ac や dendai のネームサーバのアドレスを知っているので、そこに登録されている東京電機大学のネームサーバのアドレスを教えます。すると、次に東京電機大学のネームサーバに問い合わせをします。すると、このサーバは c を解釈して情報通信工学科のネームサーバのアドレスを教えます。最後に情報通信工学科のネームサーバは登録されている www という名前を持つホストの IP アドレスを教えます。

3.4 BSD ソケット

カルフォルニア大学バークレイ校 (*UCB*) では *UNIX* の研究が行われていました。ここでインターネットに接続するソフトウェアが開発されました。ここで開発された手法は、基本的にはファイルの入出力と同様にインターネットでのやりとりも扱えるようにする、ファイルハンドルに対応する**ソケット**と呼ばれる仕組みを使ってインターネットを使用する仕組です。プログラムでインターネットを利用する際は、ファイルを利用する時にファイルハンドルを指定するのと同じように、このソケットに対してデータを読み込んだり書き込んだりします。

当時 UCB で開発された UNIX は *Berkley Software Ditribution*(*BSD*) と呼ばれ、著作権表示をするだけで自由にソースファイルを利用できるライセンスで配布されてました。そのため BSD の UNIX と同じ手法はパソコンの OS など様々なコンピュータシステムに移植されていきました。 Microsoft Windows にもこの BSD ソケットが winsock という形で移植されています。

最初のソケットは UNIX の開発言語である C 言語に組み込まれました。C 言語ではファイルを使うのに次のようにファイルハンドルを用います。fopen 関数によりファイル名を指定してファイルハンドルを得ます。そして read 関数によりファイルハンドルを指定してデータを得、 write 関数によりファイルハンドルとデータを指定してデータを書き込みます。そして fclose 関数でファイルハンドルの使用を終了します。一方 C 言語でインターネットを利用する際、まず *connect* 関数で相手のホスト名とポート番号を指定してソケットを得ます。そして *read* 関数ではソケットを指定してデータを得、 *write* 関数でソケットとデータを指定してデータを送信します。そして *close* 関数でソケットの使用を終了して接続を切ります。このように fopen が connect に、 fclose が close に替わるだけで、ほぼファイルと同じ操作でインターネットが使用できます。

しかし、connect される側はもう少し複雑です。それは不特定の相手から connect されるのを待つ必要があるからです。相手に呼び出してもらうためにあらかじめポート番号を一つ固定する必要があります。また、connect された時、それに対応するのと同時に、平行して別の接続を待つ必要があります。そのため、待つためのソケットは一定のポート番号のままにする一方、受け付けた接続に対しては別に割り当てられたポート番号を使用して処理します。つまり、connect される側 (サーバ) は次のような処理になります。

1. 接続を待つポートを固定して、接続を待ちます (*listen*)。
2. そのポートに接続が来たら、サーバプログラムはそのポートを次の接続を待つために明け渡して、別の通信用のポートを用意して、接続者と共に移ります (*accept*)。その時、 accept の処理で得られる実際にデータをやりとりする通信用のソケットを得ます。この accept で得られたソケットに対して read, write を行います。なお、接続者側の移行処理は connect に含まれ自動的に行われます。
3. 得られた通信用のソケットに対して read, write によりメッセージを送受します。

3.5 Java とソケット

本実験では *Java* 言語を使用します。Java 言語そのものについては、簡単な Java 言語の紹介を付録 C(15 ページ) に用意しましたので参照して下さい。ここでは Java を利用したネットワークの利用法を紹介します。

3.5.1 TCP のクライアント

Java で TCP のクライアントを作るには java.net.InetAddress と java.net.Socket というクラスを使います。

java.net.InetAddress は public コンストラクタのないクラスですが、InetAddress.getByName() というファクトリ (クラスメソッド) を使うとホスト名を示す文字列から InetAddress のインスタンスを生成できます。

java.net.Socket ではコンストラクタに InetAddress のインスタンスとポート番号を指定することで、Socket インスタンスを生成できます。これは C 言語での connect に対応しています。但し、この後、実際に情報をやりとりするのに、C 言語と異なり、Socket と情報のやりとりを行いません。Socket のインスタンスに getInputStream() メソッド、 getOutputStream() メソッドを送ることにより、それぞれ java.io.InputStream, java.io.OutputStream のインスタンスを得ることができます。この得られたインスタンスと情報のやりとりをします。

通信が終ったら Socket のインスタンスに close() メソッドを送ります。

3.5.2 TCP のサーバ

Java で TCP サーバを作るには java.net.ServerSocket と java.net.Socket を使います。ServerSocket のコンストラクタにポート番号を指定してインスタンスを生成することで listen 状態になります。作られたインスタンスに対して、accept() メソッドを送ると Socket インスタンスが得られます。この Socket インスタンスは接続してきた相手とつながってますので、この Socket インスタンスに getInputStream() や getOutputStream() メソッドを送って Strem オブジェクトを得て通信をします。

3.5.3 HTTP クライアント

TCP クライアントの特殊なものとして、 HTTP のクライアントを作る場合、java.net.Socket を使うよりもっと簡単に作ることができる専用のクラスライブラリがあります。java.net.URL はコンストラクタに目的のリソースの URL を文字列で指定することでインスタンスを生成します。このインスタンスに openStream() メソッドを送ると java.io.InputStream のインスタンスが得られます。

3.5.4 UDP のクライアント

Java で UDP を使用するクライアントを作るには java.net.InetAddress, java.net.DatagramPacket, java.net.DatagramSocket を使います。送るデータを byte 型の配列として用意し、送り相手のアドレスを java.net.InetAddress のインスタンスで用意します。java.net.DatagramPacket のコンストラクタに、byte 型の配

列名、配列のサイズ、 InetAddress のインスタンス、ポート番号を指定して送信データに対応するインスタンスを生成します。

一方、java.net.DatagramSocket のデフォルトコンストラクタを利用してインスタンスを作っておきます。このインスタンスに対して、send() メソッドで先ほど作った DatagramPacket のインスタンスを指定すると、 UDP を利用してデータが送られます。

3.5.5 UDP のサーバ

UDP のサーバを作るには、 java.net.DatagramSocket のコンストラクタにポート番号を指定してインスタンスを作成します。すると、 listen 状態になります。一方、あらかじめ領域を確保した byte 型の配列を用意して、配列変数名と配列のサイズを指定して java.net.DatagramPacket のコンストラクタを呼び出して、受信用の DatagramPacket を用意します。そして、 DatagramSocket のインスタンスに 受信用の DatagramPacket のインスタンスを指定して receive() メソッドを送ると、用意した DatagaramPacket インスタンスに受信データが入ります。

4 実験で使うプログラムについて

4.1 TCP を使うクライアント

TCP を用いる最も単純なクライアントは、各 OS に付属している telnet です (但し、 Windows Vista はオプション)。これは、ホスト名とポート番号を指定すると相手ホストの指定ポートへ TCP 接続をします。情報通信基礎実験では telnet を使って 80 番のポートを指定して HTTP サーバに接続しました。ここでは、telnet でアクセスするのと同じように HTTP サーバに接続して WWW サーバの性能を評価するプログラムを作ります。

始めに Java のソケットを使ったプログラム BenchSample.java を付録 G(38 ページ) に紹介します。これは、WWW サーバの特定のコンテンツを何度も受信して、その経過時間を計ることで WWW サーバの性能を測定するプログラムです。「javac BenchSample.java」でコンパイルした後、「java BenchSample ホスト名 ポート番号 パス 繰返し回数」を実行すると、指定した回数だけ相手ホストの指定ポートに対して TCP 接続を行い、HTTP/1.0 に従ってパスで指定したファイルを取得します。そして、これにかかった所要時間を出力して終了します。

プログラム中で、クライアント側のソケットを作成するために、サーバのホスト名とポート番号を指定したコンストラクタを呼び出してます。そして、getInputStream() メソッドを送ると java.io.InputStream が得られ、getOutputStream() を送ると java.io.OutputStream が得られます。これらを介して HTTP/1.0 のプロトコルをやりとりします。

ところで、 Java で HTTP を利用する時、わざわざソケットを使うより、もっと便利な java.net.URL を使った方がプログラムを簡潔にできます。上の BenchSample.java と同等で java.net.URL を使ったプログラム Bench.java を付録 H(39 ページ) に示します。

4.2 TCP を使ったサーバの作成 (WWW サーバ)

HTTP/1.0 の一部を実装した WWW サーバ Http.java を紹介します。HTTP/1.0 の仕様の抜粋については付録 F(35 ページ) を御覧下さい。これを元に作成したサーバが付録 I(40 ページ) です。このプログラムはオブジェクトクラスをいくつか定義してますので、順に説明します。

4.2.1 Constants

サービスクラスとして定義されたこのクラスは、単純にプログラムの中の定数を集めているだけです。プログラムの先頭に置くことにより、設定変更が容易になります。

4.2.2 Http

Http クラスは本プログラムのメインルーチンで、クラスメソッド main() のみが実装されています。具体的にはサービスポートを作成したあと、accept したソケットを使って情報のやりとりを行います。得られたソケットに対して、入出力のストリームを作成した後、Request オブジェクトを作り、 Reply コンストラクタに渡してオブジェクトを作ります。そして、作成した Reply オブジェクトを送信します。

4.2.3 Reply

Reply クラスでは、接続相手への情報送信を行います。定義されているメソッドはコンストラクタ Reply() と send() です。コンストラクタでは Request オブジェクトを受け取り、 ReplyData を作成します。この ReplyData は抽象クラスで、通常は Request で指定されたファイルへのハンドルを持つ FileData のオブジェクトになります。しかし、なんらかのエラーが発生した場合はエラーメッセージを持つ ErrorMessage オブジェクトに挿げ替えられます。

send() メソッドはコンストラクタで作成した ReplyData オブジェクトに対して出力ストリーム (PrintStream) に send するように指示します。

4.2.4 ReplyData

ReplyData はサブクラスに ErrorMessage と FileData を持つ抽象クラスです。

メンバ変数の errorCode と errorMessage は HTTP の Status 行に必要な値なので、メッセージの種類に関わらず必要になります。そのため、親クラスで持つことになります。また、 Status 行の出力、ヘッダ行の出力など、メッセージに関わらず送信時に行う処理も共通化させるため、このお役ラスに持たせます。但し、 ErrorMessage と FileData では各パラメータ (日付やサイズなど) の処理が異なってますので、これに関しては abstract 宣言をして、サブクラスに定義させます。

また、実際のデータ本体の送信も abstract 宣言をして、サブクラスに任せます。

4.2.5 ErrorMessage

ErrorMessage は与えられた Status コードからエラーメッセージのオブジェクトを作ります。Status コードとエラーメッセージの辞書 errorTable を HashMap により実現します。errorTable は ErrorMessage のファクトリ getInstance が最初に呼ばれた時に作られます。Status コードをキーとし、 Status コードとエラーメッセージから作られた ErrorMessage オブジェクトを値とした HashMap を作ることにより、検索されれば常に同一のオブジェクトを返すシングルトンを実現しています。

getLastModified, getContentLength, getContentType はそれぞれエラーメッセージに関する情報を返すように実装します。 日付は RFC822 に示された形式にフォーマットします。

send では親クラスである ReplyData の sendHeader を呼び出した後、 エラーメッセージを出力します。

4.2.6 FileData

FileData は与えられた File オブジェクトを送信するオブジェクトです。コンストラクタでは File オブジェクトから FileInputStream を作成します。この際、ファイルが存在しないと FileNotFoundException を発生します。正常動作なので、 Status コードは 200 Ok になります。

getLastModified では RFC822 で示された日付のフォーマットに基づいてファイルの作成日時を返します。getContentLength はファイルの長さを返します。 getContentType はファイルタイプを *MIME*(Multipurpose Internet Mail Extensions) 形式 [5] で返答します。ここでは、デフォルトの application/octet-stream と text/plain

のみの最低限の実装しか行ってませんので、 HTML の書かれたファイルや JPEG などの画像ファイルを送信するには、そのようなファイルタイプ (text/html, image/jpeg など) を認識するように書き換える必要があります。

send は親クラスの sendHeader を実行した後、コンストラクタで生成した FileInputStream から一文字ずつ読み込んだ文字を送信する。ファイルを読み終えたらクローズする。

4.2.7 Rfc822

Rfc822 クラスは RFC822 に示されている日付のフォーマットで世界標準時に日付をフォーマットする java.util.DateFormat オブジェクトを提供するクラスです。

4.2.8 Request

Request クラスでは接続相手からメッセージを受信します。定義されている public メソッドはコンストラクタと受信を処理する read です。コンストラクタは読み込み先である BufferedReader を受け取るだけです。

一方、read は HTTP/1.0 プロトコルに則り、このクラスの private メンバ readFirstLine, readHeader, readBody を順に呼び出し、リクエストされたファイル名を返します。

readFirstLine は HTTP リクエストメッセージの一行目の構文を調べ、正常なリクエストであれば、メッセージの中からファイル名を取り出し返します。なお、構文エラー、プロトコル違い、GET 以外のメソッド要求、その他入出力のエラーに関してはそれぞれ SyntaxException, ProtocolException, NotImplementedException, IOException の例外を発生します (IOException 以外は独自の例外)。

readHeader はヘッダ部分を解析し、 Content-Length フィールドの値を取り出し返します。ここで、構文に不都合がある場合は SyntaxException(数値のある位置に数値以外がある場合も含む) を、またその他入出力のエラーに関しては IOException を発生します。

readBody は行を読み捨て、文字数を返します。ここでも入出力エラーに関しては IOException を発生します。

このような 3 つの private 関数に関して、 read は次のような動作をします。

1. getFirstLine でファイル名を読み取る
2. getHeader で ContentLength を読み取る
3. getBody により読み込んだ文字数だけ、 ContentLength から減算していき、 0 になるまで繰り返す
4. ファイル名を返す

4.2.9 SyntaxException, ProtocolException, NotImplementedException

これらのクラスは java.lang.Exception のサブクラスで、新しい Exception を増やすために作られています。作られた新しい Exception は Request クラスの read() で発生し、Reply クラスのコンストラクタでエラーメッセージの生成や処理に利用されています。

4.3 UDP を使った通信

UDP を使った通信を行うプログラム Greeting.java を紹介します。付録 J(45 ページ) に示したプログラム Greeting は UDP を用いた通信を行うものです。「java Greeting アドレス」で実行し、なにか文字列を送ると指定したアドレスに対して文字列を送ります。一方、なにか文字列を受け取ったら受け取った文字列を表示し、送信先にあらかじめ設定しておいた文字列を送ります。なお、Windows なら Ctrl-Z+[Enter]、UNIX 系なら Ctrl-D で終了します。

この Greeting のプロトコルは次のような単純なものです。プログラム自体はサーバ機能とクライアント機能の両方を持ちます。サーバ機能は次のような動作をします。

1. なにか文字列を受けとったとき、送り主の IP アドレスと、受けとった文字列を表示します。
2. もし、受けとった文字列が '+' で始まる文字列の場合はなにもせずに 1 に戻ります。
3. そうでない場合、あらかじめ決めておいた文字列の先頭に'+' を付加して送信元に送ります。そして 1 に戻ります。

一方、クライアント側は単に文字列を送るだけです。

UDP の場合、ブロードキャストアドレスを指定すると、そのネットワークに属するすべてのホストに同じ情報を送ることができます。そこで、このプロトコルにおいてクライアントがブロードキャストアドレスに文字列を送ると、このネットワークに属するサーバ全部からメッセージを受けとることができます。

プログラムはサーバとクライアントを並列に動かすため、 java.lang.Thread の機能を使います。これは、public void run() メソッド を用意した java.lang.Thread のサブクラスを作成すると、そのサブクラスのインスタンスに対して、 start() メソッド を送ることにより run() メソッドが並行して実行されると言う機能です。そのため、Server クラスは Thread のサブクラスとして定義し、並列に実行したい部分を public void run() メソッド に記述しています。public void run メソッドでは規定のサイズのメッセージを java.net.DatagramPacket の receive メソッドにより受信します。メッセージを printPacket メソッドにより表示します。そして、受信したメッセージの先頭に + 記号が無ければ Sender クラスへ + 記号付きのメッセージを送らせます。なお、receive に使う DatagramSocket は 1 秒でタイムアウトし、InterruptedIOException を発生するため、 halt メッセージを受け付けると、一秒以内に while の条件 ok が false になっていることを確認し、 Thread を正常終了します。

Greeting の main メソッド では Server のインスタンスを作成して、start() メソッドを送ってサーバを起動します。その後、クライアントの処理を行います。クライアントは単純に標準入力を一行ずつ読み、読み込んだ内容を Sender クラスへ渡し、メッセージを送る処理を繰り返しています。入力が終了したら、 Server に halt メッセージを送り、終了します。

Sender クラスは は シングルトン を実装しており、 getInstance 静的メソッドで 共通のオブジェクトを受け取った後、 sendMessage でメッセージを送ります。sendMessage メソッドは 与えられた文字列を指定されたアドレスへ送ります。

5 実験

5.1 実験準備

1. Java の開発キット (JDK) を入手し、利用できるようにしておきます。(付録 A(13 ページ) 参照)
2. Apache のインストールを行います (付録 B(13 ページ) 参照)
3. 自分のノートパソコン、サーバ機用の IP アドレスを交付してもらい、それぞれ設定します。ネットワークコンピュータのプロパティを選び、使用しているネットワークアダプタのプロパティを選びます。そして、TCP/IP のプロパティの内容を設定します。なお、一台のパソコンを一人で使い、全ての実験をする場合は、ネットワークの設定は不要です。アドレスとして localhost を使います。
4. 必要なファイルを実験サーバからダウンロードします。

5.2 TCP の実験

1. 適当なコンテンツ (テキストファイルまたは HTML ファイル) を作る。例えば、 index.txt と言う名前で「Access Succeeded. This is a 99kc999's sample content.」と記入されたファイルなどを作る。Apache サーバに登録し、 WWW ブラウザから確認する。なお、日本語のデータは正しく表示されるとは限らないことに注意する。

2. Bench.java をコンパイルする。

3. Apache サーバ上の作成したコンテンツに Bench.java で 100 回、1000 回、 10000 回 アクセスし、アクセスにかかる時間を測定する。但し、それぞれ 3 回ずつ計り、平均値を求める。両対数グラフ用紙を使い、回数とかかる時間の関係を図示する。縦軸の最大は 10 分程度で良い。

   なお、Windows!Windows 2000 以降の OS では Ctrl+Alt+Del でタスクマネージャーを呼び出すと、コンピュータやネットワークの効率を調べることができる。

4. 100KB, 1MB, 10MB, 100MB のファイルを LargeFile.java (付録 K(48 ページ)) などを使用して作成し、apache サーバから取得可能にする。

5. 各ファイルに対して同様にアクセス回数とかかる時間の関係を調べる。但し、実験 2 と比べてファイルの容量が違うため、アクセス回数を事前に決定しておくと、実験が長時間になる恐れがある。しかも、長時間行うことが本質の実験ではない。

   そのため、次の手順で行う。

   (a) 一つのファイルに対して、まず 100 回のアクセスをし、時間を測定する。三回測定し平均値を求める。

   (b) 両対数グラフにプロットする。縦軸の最大を 10 分程度にする。

   (c) 同一のファイルに対して少なくとも 3 種類の測定点を確保できるよう、アクセス回数を予測する。但し、一回の測定時間が 5 分を越えないように調整する。予測がつかない場合は 1000 回、 10000 回とする。

   (d) 測定はそれぞれ 3 回ずつ行い、平均値をとる。そして両対数グラフに記入し、線形近似曲線を引く。

6. Http.java をコンパイルし、 Http.class ファイルができたディレクトリに htdocs ディレクトリを作成する。

7. その htdocs ディレクトリに index.txt と 100KB, 1MB, 10MB, 100MB のファイルを置く。

8. 「java Http」で起動した後、WWW ブラウザから http://localhost:8080/コンテンツ名 でアクセスし、Http.class が正常に動くか確かめる。なお、 Windows XP SP2 以降では Http.class を起動すると「このプログラムをブロックし続けますか?」警告が出るが、ブロックの解除を指定する。

9. 実験 2, 5 と同様に Http.class の性能評価をする。

10. なお、報告書において使用したコンピュータの CPU, CPU クロック、コンピュータの製品名、ネットワークアダプタの形式やメーカ、 OS 名、Java のバージョンなどコンピュータの性能を示す項目を記載すること。

5.3 UDP の実験

1. Greeting.java プログラムを入手し、Constant クラス に含まれる文字定数 msg の内容を適当に変更する (学籍番号を入れるなど他人と同じにならないように工夫すること)。そして、プログラムをコンパイルする。

2. 「java Greeting localhost」 として実行し、適当な文字列を打ち込んで Enter キーを押す事で、プロトコルが正しく動作していることを確認する。

3. 二人一組になり、相互に「java Greeting 相手ホストの」と実行する。文字列を打って Enter キーを押すと、相手のコンピュータに同じ文字列が送られることと、相手のサーバからメッセージが返されることを確認する。

4. 「java Greeting ブロードキャストアドレス」と実行する。文字列を打って Enter キーを押すと、そのネットワークに属しているブロードキャストネットワーク内で Greeting.class を実行しているホストからメッセージが返ってくることを確認する。そして、その実行しているホストへメッセージが伝わったことを確認する。

5.4 追加項目

以下の実験は余裕があったら行いなさい。

1. 二人ひと組になり、一方がサーバ、一方が計測と役割分担し、Apache サーバ、 Http.class の性能評価を行う。なお、コンピュータの性能差を吸収するためサーバ、計測は相互に交替して行うこと。
2. 6.2 節の 2 項に関する実験。
3. 6.2 節の 3 項に関する実験。
4. 6.2 節の 4 項に関する実験。

6 検討事項

6.1 必須項目

1. Apache サーバと Http.class サーバでパフォーマンスが異なる理由を考えなさい。
2. Greeting の通信のやりとりの例を図示して解説しなさい。
3. Greeting において、 '+' を付加する処理を取り除くと、プログラムの動作がどのようになるか考えなさい。
4. Greeing で大きなパケットをやりとりすることはできますか? プロトコルの性質などに触れながら理由も説明しなさい。

6.2 発展事項 (任意に選択)

1. 本演習で使用したプログラムに関する改善点があれば指摘しなさい。
2. 複数の人数で同じサーバに対して同時に Bench.class で性能評価を行い、同時アクセス数がどの程度パフォーマンスに影響するか調べなさい。
3. Bench.java を Thread を使って複数の接続ができるように改造し、性能評価がどのように変化するか調べなさい。
4. Http.java を Thread を使って複数の接続ができるように改造し、性能評価がどのように変化するか調べなさい。
5. Greeting.java を応用した実用システムを考案しなさい。

A Java の開発環境 JDK のインストール

1. http://java.sun.com/ に接続します。
2. その中で Popular Downloads の項目の中の J2SE5.0(日本語) などバージョンの大きな日本語版ををクリックします (J2SE とは Java2 Standard Edition の略)。
3. 必要なのは *JDK*(J2SE Development Kit) なので、最新のものをダウンロードします (2006 年 3 月 1 日現在は JDK5.0 Update6)。なお、*IDE*(開発環境) が必要な方は Eclipse を使うか、netBeans がバンドルされているものを選んで下さい。多少のバージョン違いは実験には大きな影響はありません。この実験では Java2SE 1.4 以降が必要です。
4. ここで、ダウンロード、インストール手順、 J2SE リリースノート、NetBeans リリースノート、使用許諾契約のメニューが出ます。
5. まずはダウンロードをするため、「ダウンロード」をクリックします。
6. ライセンスの表示が出ます。netBeans を選んだ場合は JDK と netBeans で異なるライセンスが表示されます。Accept ボタンを押し、Continue ボタンを押すと次に進みます。
7. 最終のダウンロードページではプラットフォームを選びます。Windows Platform を選んで下さい。netBeans 付き jdk-1_5_0_06-nb-4_1-bin-win.exe で 130.46MB あり netBeans なし jdk-1_5_0_06-windows-i586-p.exe で 59.86MB あります。ネットワーク環境の良い所でダウンロードして下さい。
8. ダウンロードが終ったら手順 4 で表示したページに戻り、インストール手順を読んでインストールします。基本的にはインストーラを起動して質問に答えていくだけです。
9. 最後に環境変数を追加します。Windows 2000 Professional と Windows XP 系の場合、マイコンピュータ→プロパティで「詳細設定」タブの「環境変数」ボタンを押します。「システム環境変数」の中の「Path」を選び「編集」ボタンを押します。その変数値の最後に 「`;\Program Files\Java\jdk1.5.0_06\bin`」など実際に JDK がインストールされたディレクトリ中の bin ディレクトリの位置を追加します。「Ok」ボタンを押してプロパティの窓を消します。
10. コマンドプロンプトを開き、「*javac*」と入力して使い方の説明が出ればインストール終了です。

B WWW サーバ:Apache のインストール

1. http://httpd.apache.org/download.cgi に載っている the best available version という項目の中の Other files というリンクをたどると、 mirror site にアクセスできます。そこで binaries/win32 にある最新の Apache サーバを取得します。ファイル名は apache_2.0.55-win32-x86-no_ssl.msi などとなっていますので、 apache 直後のバージョン番号が一番新しいものを選びます。
2. ダウンロードした apache-...msi を実行してインストールを始めます。
3. ライセンスの認証画面を認証して通過すると、サーバの基本情報を入れる画面が出ます (図 1)。ドメイン名は jikken.c.dendai.ac.jp を、サーバ名は自分のパソコン名. jikken.c.dendai.ac.jp(例: 99kc999.jikken.c.dendai.ac.jp)、メールアドレスは自分のメールアドレスを入れて下さい。また使用するポートは 80 番を指定して下さい。
4. あとは指示に従ってインストールをして下さい。インストールが終ると Apache サーバは自動的に起動します。
5. なお、Windows XP でサービスパック 2 が適用されていると Apache のサービスをファイアーウォールが検知します (図 2)。ここでブロックを解除して下さい。

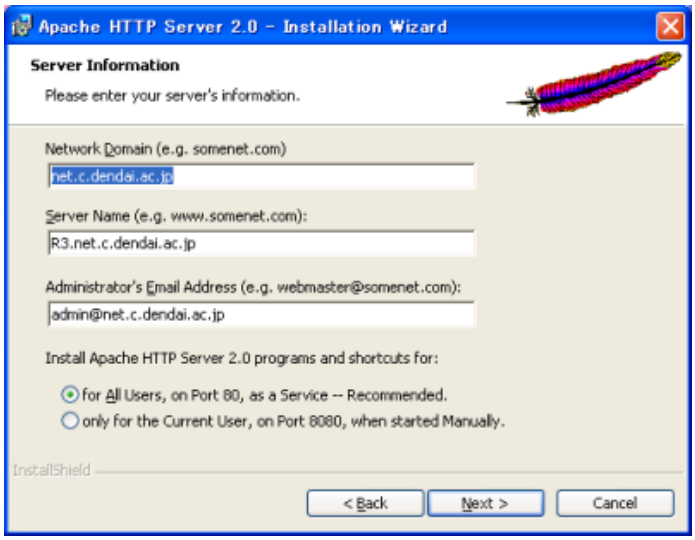

図 1: サーバの基本情報

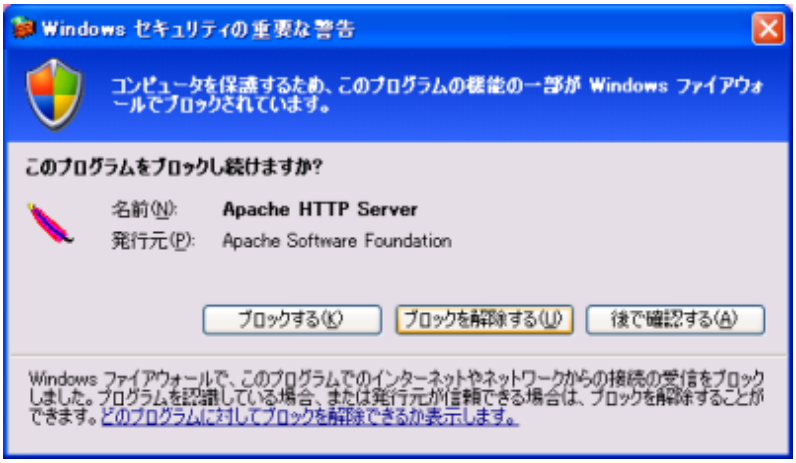

図 2: ファイアウォールによる検知

6. インストールが終ったら、WWW ブラウザで http://localhost/ にアクセスして、Apache のインストール完了画面が出ることを確認して下さい。なおドキュメントは `c:\Program Files\Apache Group\Apache2\htdocs` フォルダに入れて下さい。

7. msi ファイルでうまくインストールできない場合は exe ファイルで試してみて下さい。

C　簡易版 Java 言語入門

C.1　オブジェクト指向と基本構文

巨大なソフトウェアを作るためには、そのソフトウェアを分割して分析する必要があります。その時、データだけとか処理だけとかを分離して考えるのではなく、分割された部分自体がまた一つのソフトウェアとして機能するような分割を考えると効率的かつ有効に制作ができることが知られています。オブジェクト指向とは、プログラムを分割する際、このように単体で機能をもつ **オブジェクト**と呼ばれる部分に分割し、オブジェクトを他のプログラムのように操作して目的のプログラムを作る方法です。

プログラムを作成する際、オブジェクトを設計する部分と、実際にオブジェクトを操作する部分に分かれます。オブジェクトの設計は **クラス**と呼ばれます。クラスでは主にオブジェクトの使用するメモリと、オブジェクトを操作するための **メソッド**を定義します。このように定義されたオブジェクトは、オブジェクトの生成と、オブジェクトへのメッセージ伝達により操作されます。Java 言語でクラスからオブジェクトを生成するには *new* **演算子**にクラス名を指定します。するとそのクラスの性質を持ったオブジェクトが作られます。なお、クラス定義の中にクラスの名前と同じメソッドがある場合、そのメソッドは **コンストラクタ**と呼ばれ、オブジェクトの初期化に使われます。クラスから生成されたオブジェクトをそのクラスの **インスタンス**と呼ぶことがあります。

プログラムでは生成されたインスタンスを**変数**を使って取り扱います。Java の変数はインスタンスそのものを格納せず、インスタンスを**参照**します (次節に記述)。そのインスタンスに**メッセージ** (= メソッドと引数) を送り、戻り値を受けとる構文は図 3 のようになります。

C.2　基本型とオブジェクトクラス

Java の**変数型**には、**基本型**と**オブジェクト型**の二種類があります。

基本型には、C 言語と同様に int, double, char, byte などの**型**があります。これらは C 言語と同様に使えます。但し、 C 言語と異なりプログラムのどこでも**変数宣言**が可能です。宣言すると値を格納する領域が作られ、代入により値がコピーされます。

しかし、C 言語とは違い**配列**の扱いオブジェクト型に近くなります。配列は型の後に [] を付けて宣言します。しかし、宣言しただけでは領域は確保されず、 new 演算子を使用して領域を確保します。(図 4)

オブジェクト型の変数は C 言語で言えばポインタのような働きをします。変数宣言だけではオブジェクトは生成されません。オブジェクト型の変数がオブジェクトを取り扱うには、オブジェクトを返すファクトリ 関数の戻り値を与えるか、クラスのコンストラクタを new 演算子とクラス名と同じ名前のコンストラクタを指定してオブジェクトを生成します。図 5 はクラス A の変数 x を宣言した後、オブジェクトを生成して、 x がそのオブジェクトを参照するようにする例です。

なお、代入文により、基本型の変数は値がコピーされ、オブジェクト型の変数には参照のみがコピーされることは図 6 のプログラムにより分かります。なお、配列やオブジェクト自体をコピーするには *clone* **メソッド**を使います。

```
戻り値用変数名 = 変数名. メソッド名 (引数);
```

図 3: Java の基本構文

```
byte[]  配列名 ;
 配列名  = new byte[1024];
```

図 4: 配列の初期化

```
A x;
x = new A();
```

図 5: 変数の宣言とインスタンスの生成

なお、C 言語と異なりグローバル変数の概念はありません。Java の変数は常にどこかのクラスに属す必要があります。

C.3 クラスの定義

Java では **オブジェクト**を作るには**クラス**を定義します。そのためには図 7 のような構文で定義します。

オブジェクトとは**メッセージ**を受けて動作する一種のアプリケーションソフトのようなものです。オブジェクトは記憶領域として**内部変数**を持ち、また操作を行うため**メソッド**を持ちます。そして、メッセージはメソッドに**引数**を指定したものになります。**クラス定義**ではこのような指定を行いオブジェクトの仕様を明らかにします。このクラス定義にしたがってオブジェクトを生成することができます。このクラスの定義に従って生成されたオブジェクトを**インスタンス** と言います。インスタンスの記憶領域とメソッドをそれぞれ**インスタンス変数**、 **インスタンスメソッド** と言います。特に、クラス名と同じ名前を持ち、インスタンスを初期化して生成する戻り値の型を指定しないメソッドを **コンストラクタ** と言います。なお、クラス自体もオブジェクトなので、それ自体、記憶領域とメソッドを持ちます。これを**クラス変数**、**クラスメソッド** と呼びます。

インスタンス変数 インスタンス変数の定義は C 言語の構造体などと同じように定義します。インスタンス変数はインスタンスが生成された時に作られ、インスタンスが不要になる時まで領域が保持されます。

インスタンスメソッド インスタンスメソッドはインスタンス変数をアクセスする関数を定義します。図 8 のように戻り値の型の次にメソッドの名前を書きます。戻り値を返さないメソッドには型として *void* を指定します。インスタンスメソッドからはインスタンス変数やクラス変数を参照できます。また、手続きの中で一時的に使用したい変数は随時宣言して使用することもできます。

コンストラクタ インスタンスメソッドの中でクラス名と同じ名前のメソッドを**コンストラクタ**と言い、インスタンスを作る際の初期化のために使われます。コンストラクタだけは特別に返り値の型名を指定しません。

クラス変数 (静的変数) クラス変数は static キーワードの後に変数宣言をします (C 言語の static とは意味が違います)。クラス変数はプログラムが起動してから終了されるまで領域が保持されます。クラス変数にアクセスするためには「クラス名. クラス変数」とします。また、インスタンス変数はインスタンスが生成された数だけできますが、クラス変数はインスタンスをいくつ作っても一つだけしか存在しません。なお、**初期値**を = の後に書くことができます。インスタンスメソッドはクラス変数へアクセスすることはできます。

クラスメソッド クラスメソッドも static キーワードの後にメソッドの宣言をします。また、実行時に「java クラス名」を指定すると、 public static void *main*(String[] arg) クラスメソッドが呼ばれます。これは C 言語の関数とほぼ同等なものになります。クラスメソッドからはクラス変数をアクセスできますが、インスタンスメソッドやインスタンス変数はアクセスできません。クラスメソッドにアクセスするには「クラス名. クラスメソッド名 ();」とします。

定数 クラス変数の宣言の前に *final* キーワードを置き、初期化すると **定数**として扱うことができます。final は再代入などを禁じるためのキーワードです。

Java にはクラスの集まりとして**パッケージ**という概念があります。本実験で紹介するプログラムではプログラム毎にパッケージを一種類しか使わないので、何も指定しない**デフォルトパッケージ**となっています。クラス内の変数やメソッドはパッケージ毎に管理されて、一般にはパッケージを越えてアクセスできません。複数のパッケー

```
class Kazu {
    int num;
    Kazu(int n){num=n;} //コンストラクタ
    public static Kazu getInstance(int n){
      return new Kazu(n); //ファクトリ
    }
    void add(int n){num+=n;}
    int value(){return num;}
}
class Main{
    public static void main(String[] arg){
        int a,b; //基本型
        Kazu x,y,z,u; //オブジェクト型
        a=1;
        b=a; //値のコピー
        a++; // b には影響しない
        System.out.println(b); // 1
        x=new Kazu(1);
        y=x; //参照のコピー
        z=x.clone(); //オブジェクトのコピー
        x.add(1); // x と y の両方が参照している
        u=Kazu.getInstance(0);
        System.out.println(y.value()); // 2
        System.out.println(z.value()); // 1
        System.out.println(u.value()); // 0
    }
}
```

図 6: 変数の内容

ジを使うには *package* 文により宣言を行う必要があります。パッケージを越えてアクセスを許すには *public* キーワードをクラス、変数、メソッド宣言の前に置く必要があります。一方、変数やメソッドをクラス内のみのアクセスに制限するには *private* キーワードを使います (C.5 参照)。クラス宣言の例を図 9 に示します。

なお、配列はクラスインスタンスのように new 演算子を指定して作成しますが、配列には length メッセージも定義されていて、配列のサイズを得ることができます。

C.4 継承と Java の interface

効率良くクラスを作成するために、クラスを拡張して別のクラスを作る方法が提供されています。これを **継承**と言います。継承されたクラスを**サブクラス**と言い、継承したクラスを **親クラス** と言います。クラス A を継承してクラス B を作るには図 10 のように *extends* キーワードを使って宣言します。なお Java では変数やメソッドの追加はできますが、削除はできません。親クラスと同じ名前のメソッドを追加すると、親クラスのメソッドに優先してサブクラスのメソッドが有効になります。これを **オーバライド**と言います。

さて、ここで重要なのは、サブクラスのインスタンスの参照は親クラスの型の変数に代入できることです。つまり図 10 の例で言うと、サブクラス B のインスタンスは、親クラス A 型の変数に代入 (で参照) することができま

```
class クラス名 {
   クラスの構成要素 1;
   クラスの構成要素 2;
   ……
}                                    //
```

図 7: クラス定義

```
戻り値の型 メソッドの名前(引数 1 の型 引数 1 の名前,
                              引数 2 の型 引数 2 の名前, ...){
    手続き;
...                                                          //
}
```

図 8: インスタンスメソッド

す (図 11) [1]。但し、親クラス変数に対しては、親クラスに属するメソッドしか使用することはできません。つまりサブクラスで追加したメソッドにはアクセスできません。しかし、オーバライドされたメソッドに関してはサブクラス側のメソッドが使用されます。もし、 A のサブクラスとして B の他に C が存在し、それぞれ同じ名前のメソッドをオーバライドしているとします。この時、A 型の変数の参照するインスタンスに対してそのメソッドを送ると、B のインスタンスか C のインスタンスかを判別してメソッドを使い分けます。図 12 で Main.class を実行すると、 A 型の変数 x は B のインスタンスを参照し、 A 型の変数 y は C のインスタンスを参照しているので、「I am B.」 と 「I am C.」 が出力されます。このようにインスタンスの型によりメソッドが多様に選択されることを**ポリモーフィズム** と言います。

さて、親クラスでインスタンスを生成せず、サブクラスだけでインスタンスを生成してポリモーフィズムを利用する場合を考えましょう。この場合、オーバライドされるメソッドは、親クラスで宣言される必要はありますが、実装される必要はありません。こういう場合に *abstract* キーワードを指定すると実装する必要がなくなります。但し、一つでも abstract メソッドを持つ場合、クラス宣言にも abstract を指定する必要があります (図 13)。

ところで、 Java では複数のクラスを継承してサブクラスを作ることは禁じられています。しかし、現実には一つのオブジェクトをクラス A のサブクラスとしながら、別のクラス B のサブクラスとしてポリモーフィズムを利用したい場合があります。例えば、 A として人物を表すクラス、 B として名前を表示するためのメソッドを持つクラスがあった時、名前を表示できる人物のクラスを作りたい場合などがあります。そのため、原則としては継承するクラスは基本的に一つだけとしながら、他の継承したいクラス各々のすべてで変数を持たず、全てのメソッドが abstract であるという制約のもとでは実現する方法があります。それが Java 独特の interface という**クラス宣言**です。class 宣言の代わりに interface と宣言し、public abstract メソッドを public abstract キーワード無しで羅列します。なお、 interface 宣言内では変数の宣言はできません。

この interface で宣言したクラスを継承するには extends の代わりに *implements* を使って指定します。implements を指定したクラスでは interface で宣言されたメソッドを public キーワードを付けて宣言、実装します。図 14 は interface の使用例です。なお、ここでは親クラスのコンストラクタをサブクラス内で呼び出す *super*() メソッドを使用しています。interface B 型の変数 x, y がそれぞれクラス C, D のインスタンスを参照する時、 show() メソッドを送るとポリモルフィズムによりそれぞれのメソッドが使用されます。なお、 interface 宣言では public abstract メソッドの他に、**定数**を final static キーワードなしで、つまり変数宣言のように宣言できます。

[1]特に Java 言語ではあらゆるオブジェクトクラスは java.lang.Object クラスのサブクラスであるとされてますので、どんなインスタンスも java.lang.Object 型の変数で参照できます。

```
class A {
  int x; // インスタンス変数
  void set(int y){ // メソッド (setter)
    x=y;
  }
  int get(){ // メソッド (getter)
    return x;
  }
  A(){     // コンストラクタ
    x=0;
  }
  static int z=0; // クラス変数
  static int next(){ // クラスメソッド
    return z++;
  }
  final static int a=1; // 定数
}                                                    //
```

図 9: クラス宣言の例

```
class B extends A {
 追加する変数宣言;
 追加するメソッド;
...
}
```

図 10: 継承

C.5 パッケージとクラスライブラリ

Java ではクラスの集まりを **パッケージ**という単位で管理します。パッケージは通常同一ディレクトリに含まれるクラスで構成されます。メソッドや変数などのアクセスは基本的には同一パッケージ内に限ります。

パッケージには *package* 宣言で名前をつけることができます。省略すると無名のパッケージになります。またそのパッケージ名と同じ名前のディレクトリに含まれている必要があります。一方、他のパッケージに含まれるクラスを使うには「パッケージ名. クラス名」とピリオド (.) で区切って表示します。このパッケージ名をいちいち書くのが煩瑣な場合、あらかじめ *import* 宣言をしておくことで、パッケージ名を省略することができます。なお、他のパッケージからクラスやメソッドを呼出すためにはそれらが *public* 宣言されている必要があります。

Java では言語仕様として提供されているクラスライブラリが *java*, *javax* を先頭にもつパッケージ名で提供されています。例えば *java.io* には入出力関係のクラスが、 *java.net* にはネットワーク関係のクラスが集められています。

```
A x = new B();
```

図 11: 親クラス型変数への代入

```
1  class A {
2      void show(){
3          System.out.println("I am A.");
4      }
5  }
6  class B extends A{
7      void show(){
8          System.out.println("I am B.");
9      }
10 }
11 class C extends A{
12     void show(){
13         System.out.println("I am C.");
14     }
15 }
16 class Main {
17     public static void main(String[] arg){
18         A x = new B();
19         A y = new C();
20         x.show();
21         y.show();
22     }
23 }
```

図 12: 親クラス型変数への代入

なお、*java.lang* には様々な重要なクラスが入ってますが、あらかじめ import されてますので、**import** java.lang.∗; は不要です。

なお、逆にアクセスを制限する宣言として次のものがあります。同一パッケージ内でもアクセスを禁止し、クラスの中だけでアクセス可能にするには *private* 宣言をします。また同一パッケージ内ではアクセス禁止にし、クラスとそのクラスを継承するサブクラス内でアクセス可能にするには *protected* 宣言をします。

C.6 エラー処理

Java では、実行時の障害を回復不能な誤り java.lang.Error と回復可能な例外 java.lang.Exception に分類し、Exception についてはプログラム内で処理できるようになっています。いろいろな種類の Error, Exception があり、それらはすべてサブクラスとして定義されています。プログラム中で発生する Exception を *try*, *catch* 文を使用することにより、**例外処理**をプログラムを止めずにできます。try 文中で Exception が発生すると、発生した Exception の型と一致する引数を含んでいる catch 文を実行します。なお、例外処理を必要とせず、メソッドを使用するプログラムに例外処理を委ねる場合、メソッドの定義部で *throws* の後に発生する Exception の型名を列挙する宣言をします。すると、プログラム中で発生した Exception は宣言されているものであれば、そのままメソッドの実行を中止して、メソッドの使用者に Exception が伝わります。またさらに *throw* 命令を使って Exception を発生させることもできます。なお、詳しい使い方については、付録 I(40 ページ) の HTTP サーバを御覧下さい。

```
1  abstract class A {
2      abstract void show();
3  }
4  class B extends A{
5      void show(){
6          System.out.println("I am B.");
7      }
8  }
9  class C extends A{
10     void show(){
11         System.out.println("I am C.");
12     }
13 }
14 class Main {
15     public static void main(String[] arg){
16         A x = new B();
17         A y = new C();
18         x.show();
19         y.show();
20     }
21 }
```

図 13: 抽象クラス

C.7 スレッド

Java では複数の処理を並行して行うことができます。並行して行われるそれぞれの処理を **スレッド** と言います。スレッドを使用するクラスを作るには、 java.lang.Thread クラスを継承したサブクラスを作ります。そして、public void run() メソッドに実際に並行処理させる内容を記述します。一方、スレッドを実際に動かすには Thread のサブクラスのインスタンスを生成し、 start() メソッドを送ります。スレッドが現在実行されているかどうかは isAlive() メソッドを送ることにより調べます。他にも実行を制御するメソッドが java.lang.Thread に含まれています。詳しい使い方は付録 E(27 ページ) で解説します。また使用例は、付録 J(45 ページ) のプログラムを御覧下さい。ここではサーバプログラムをスレッドにして起動してからクライアントの処理を行っています。

C.8 コンパイルと実行

Java のソースファイルには拡張子 *java* をつけます。また、特に public クラスにはそのクラス名とおなじファイル名を付けます。つまり public クラスは一つのファイルに一つしか入れることができません。

コンパイルは「javac ソースファイルの名前」とします。拡張子の java は省略しません。コンパイルに成功するとファイル内で定義されているクラスが、そのクラスの名前で別々のファイルになります。拡張子は *class* です。

実行は public static void main メソッドが定義されているクラスの名前に対して、「java クラスの名前」です。拡張子 .class は付けません。

```
1  class A {
2      String name;
3      A(String s){
4          name=s;
5      }
6  }
7  interface B {
8      void show();
9  }
10 class C extends A implements B{
11     int value;
12     C(int n){
13         super("I am C");
14         value=n;
15     }
16     public void show(){
17         System.out.println(name+" having an interger value="+value+".");
18     }
19 }
20 class D extends A implements B{
21     double value;
22     D(double x){
23         super("I am D");
24         value=x;
25     }
26     public void show(){
27         System.out.println(name+" having a double value="+value+".");
28     }
29 }
30 class Main {
31     public static void main(String[] arg){
32         B x = new C(6);
33         B y = new D(2.5);
34         x.show();
35         y.show();
36     }
37 }
```

図 14: interface

D　オブジェクト指向のイデオムと デザインパターン

本章ではオブジェクト指向プログラミングにおいて基礎的かつ頻繁に使われるイデオムやデザインパターンを紹介します。

D.1　getter, setter

オブジェクト指向でもっとも重要なコンセプトは**カプセル化**です。これはプログラムの部分部分が、それぞれ完全に独立し、相互の内部構造の情報へのアクセスを遮断することです。これにより、各オブジェクトのプログラミングにおいて、他のオブジェクトに関する情報が必要最低限で済みます。このカプセル化の基本となるのが、**メンバ変数**などオブジェクトの持つ情報への直接のアクセスの禁止です。そのため、基本的に全てのメンバ変数を *private* 宣言をします。

しかし、private 宣言をすると、外部からその情報へのアクセスができなくなります。そのため、外部からのアクセスを許可する変数に関しては、単に値を取り出すだけのメソッド *getter* と、値を格納するだけの *setter* を用意する (図 15)。

```
1 class Example {
2   private int number;
3   public int getNumber(){return number;}
4   public void setNumber(int n){number=n;}
5 }
```

図 15: getter, setter

D.2　コンポジション

他のクラスを利用して、拡張したクラスを作るのに、 継承という方法があることは C.4 節で説明しました。これは既存のクラスに変数やメソッドを追加するものです。継承は親クラスの全てのメソッドを引き継ぐため、基本的には親クラスの性質を引き継ぎます。

一方、もうひとつ別の方法として、**コンポジション** という方法があります。これは言語の機能というよりデザインパターン と呼べるものです。簡潔にいうと他のオブジェクトをメンバ変数に持ち、そのオブジェクトのメソッドをそのまま実行するものです。このようにすると、他のオブジェクトの性質を全て引き継ぐことは無く、特定の性質のみを引き継ぐことができます。

継承とコンポジションの使い分け方は、継承は *is-a* **関係**、コンポジションは *has-a* **関係** と言われています。つまり、「an employee is a member.」のように 親クラスの性質を持つときは継承を使い、「Each car has a set of tyres.」のように親クラスのオブジェクトを所有するような関係の時はコンポジションを使います。

D.3　ファクトリ

Java ではオブジェクトを作るとき、通常は**コンストラクタ**を使用します。但し、オブジェクトを作成する際に、何らかの条件を付加させたい場合があります。その場合、そのクラスにオブジェクトを作成する**静的関数**を定義します。このようにコンストラクタ以外でオブジェクトを作成する関数を**ファクトリ**と言います。なお、ファクトリのみでオブジェクトを作らせ、外部でのコンストラクタの使用を禁ずる場合は、コンストラクタを private 宣言で定義します。java.net.InetAddress は、必ずアドレスを保持するようにするため、空のコンストラクタの使用を禁じられており、ファクトリにアドレスを与えてオブジェクトを作るようになっています。

```
1  class Tyre {
2    String maker;
3    double airPressure;
4    public double getAirPressure(){ return airPressure;}
5    public void setMaker(String _maker){ maker=_maker;}
6  }
7  class Car {
8    Tyre tyre;
9    String maker;
10   public double getAirPressure(){ return tyre.getAirPressure();}
11   // コンポジション
12   public void setMaker(String _maker){ maker=_maker;}
13   // 引き継がない
14 }
```

図 16: コンポジション

```
1  class Example {
2    private Example(){} // コンストラクタを封じる public static Example getInstance()return new Example();class Example2 private int counter=0;private Example2()public static Example2 getInstance()counter++;return new Example2();public static int getCounter()return counter;
```

図 17: ファクトリ

D.4 シングルトン

アプリケーションの土台となるウィンドウなど、常に一つだけオブジェクトを存在させたい時があります。この時使用するのが**シングルトン**デザインパターンです。これはファクトリを利用し、ファクトリが内部で一つだけ発生させたオブジェクトを返すようにします。

D.5 Status

オブジェクトの動作を状況に応じて変化させるデザインパターンを *status* デザインパターンと呼びます。これには二種類のクラスが必要です。一つは実際に動作を行う複数のクラス (共通の親クラスを持ちます)。もう一つは動作を行うオブジェクトを一つ保持して、オブジェクトに実行を行わせるクラスです。

まず、実際に動作を行わせるクラスですが、仮想クラスを継承し、同じメンバ関数名で動作を行わせます。

前に説明した、仮想クラスのメンバ変数を持ち、その変数への setter と、その変数に動作を行わせるメンバ関数を送るメソッドを用意します。これにより、状態を変える側が setter を扱い、実行を行わせる側がメソッドを送ることで、状態により動作を変えることができます。

なお、Java では実装をまったく行わない仮想クラスは interface として定義できます。この場合、 abstract public というキーワードを省略できます。

次にこの動作を行うオブジェクトを保持するクラスを作成します。

```
class Example {
  private static Example anInstance;
  public static Example getInstance(){
    if(anInstance==null){
      anInstance = new Example();
    }
    return anInstance;
  }
}
```

図 18: シングルトン

```
interface Action {
  void showMessage();
}
class GoodAction implements Action {
  public void showMessage(){
    System.out.println("Good!");
  }
}
class BadAction implements Action {
  public void showMessage(){
    System.out.println("Bad!");
  }
}
class Test {
  public static void main(String[] arg){
    Action action;
    action = new GoodAction();
    action.showMessage();
    action = new BadAction();
    action.showMessage();
  }
}
```

図 19: Status デザインパターン 前半

```
1  class ActionHandler {
2    private Action action;
3    public void setAction(Action act){
4      action = act;
5    }
6    public void proceed(){
7      action.showMessage();
8    }
9  }
10 class Example {
11   public static void main(String[] arg){
12     ActionHandler ah = new ActionHandler();
13     ah.setAction(new GoodAction());
14     ah.proceed();
15     ah.setAction(new BadAction());
16     ah.proceed();
17   }
18 }
```

図 20: Status デザインパターン 後半

E　スレッドプログラミング

E.1　基本プログラミング

スレッド とはプログラムの実行を分岐することで、いくつかのプログラムを並行して走らせることができます。そのためには java.lang.Thread クラスのサブクラスを作り、並行させて走らせるプログラムを public void run() メソッドとして実装します。呼び出す側はそのクラスのオブジェクトを生成して start() メソッドを送ります。するとプログラムを並行して実行することができます (図 21)。

さて、ここでは簡単な例を考えてみましょう。整数 max, num を入力した時、1 から max までを表示することを考えます。但し、 num 個のスレッドに分割して**並列処理**をして出力することを考えます。つまり 0 番目のスレッド は $f_0 = 1$ から t_0 まで、 1 番目の $f_1 = t_0 + 1$ から t_1 まで、...、$num - 1$ 番目のスレッドは f_{num-1} から $t_{num-1} = max$ までを出力することになります (図 22)。

このスレッドのクラスの名前を *Counter* とし、そのコンストラクタを Counter(from,to) とすると、クラス Counter は図 23 のようになります。run メソッドは単純に from から to までを出力するだけということに注意して下さい。

main 関数では各 f_i, t_i $(i = 0, ...num - 1)$ を求めて num 個のオブジェクトを作り、全てに start メソッドを送れば並列して動作できます。

ここでここでこの f_i, t_i の求め方を説明します。i は 0 から $num - 1$ までの値とし、max を num で割った商を q とし、余りを r とします。余り $r = 0$ の時はそれぞれのスレッドで q 個の数を出力すれば良いので、 i 番目はそれぞれ $f_i = qi + 1, t_i = q(i + 1)$ となります (図 24)。

一方、余り r が 0 で無い時はいろいろな考え方があると思いますが、今回は最初の r 個 (つまり 0 から $r - 1$ 番目) のスレッドは $q + 1$ ずつ担当し、残りのスレッドは q ずつ担当するようにしました (図 25)。

このように計算するクラス Divider を図 26 に、main 関数を図 27 に示します。

さて、これで一応単純なスレッドプログラミングはできました。ただ、ここで分割数による効率に興味があります。そこで、全てが終る時間を計測することにしましょう。それにはスレッドが全て終了することを知る必要があります。スレッドの状況を調べる方法もありますが、ここではスレッドの終了を待つことを考えることにします。すると全てのスレッドの終了を待ってから時計を止めれば所用時間が分かることになります。これを実現するのが join メソッドです。全てのスレッドに対して join をすれば、全てのスレッドが終了するまで待つことになります。上の単純な Main クラスに時間計測などの仕上げをしたプログラムを E.4 に示します。なお、 join 中に InterruptedException 例外が発生した場合は for を中断します。

E.2　スレッドのコントロール

次に、スレッドを途中で止めることを考えます。java.lang.Thread クラスには stop メソッドがありますが、これはオブジェクトが壊れるなどの危険性があるとされ、推奨されてません。ここでは stop メソッドを使わないスレッドの止め方を考えます。

さて、このために考える例として、キーボードから文字列を入れると、その文字列を 10 秒毎に 10 回表示するものを考えましょう。これは、逐次入力を受け付け、入力が得られるごとに出力するスレッドを生成します。すると、最初のメッセージの後、 9 回表示することになるので、スレッドは 90 秒間生きつづけます。このように、スレッドで一定時間待たせるには java.lang.Thread.sleep に待つ時間をミリ秒で与えます。なお、この sleep クラスメソッドは InterruptedException 例外を発生します。とりあえず停止のことを考えないで作ったのが図 28 です。Writer クラスのスレッドはコンストラクタで受けとったメッセージを単純に 10 回出力して停止します。main 関数では BufferedReader を定義し、標準入力から行を読み続けます。読み込んだ行に対して Writer クラスのインスタンスを作り、start メソッドを送るだけです。実行中に標準入力から EOF が送られると main は終了しますが、スレッドはすべての出力を終えるまで停止しません。

```
class A extends Thread {
  public void run(){
  //  並行して実行したいもの
  }
}
class Main {
  public static void main(String[] args){
    A a = new A();
    a.start();
    // a.run() の実行終了を待たずに次の動作に移る
  }
}
```

図 21: Thread の基本

$$
\begin{array}{rc}
0 & \\
1 & \\
2 & \\
\vdots & \\
num-1: &
\end{array}
\boxed{
\begin{array}{ccc}
f_0 = 1 & \sim & t_0 \\
f_1 = t_0 + 1 & \sim & t_1 \\
f_2 = t_1 + 1 & \sim & t_2 \\
 & \vdots & \\
f_{num-1} = t_{num-2} + 1 & \sim & t_{num-1}
\end{array}
}
$$

図 22: スレッド番号と担当範囲

次に、このプログラムの実行時に EOF を送って、すべて止めることを考えます。このために interrupt メッセージを各スレッドに送り、各スレッドで interrupt メッセージを処理することを考えます。しかし、図 28 のプログラムではスレッドを作るだけ作り、 start を送った後は参照もできない状況ですので、このままでは interrupt メッセージを送ることができません。そこで、作ったスレッドを管理するために java.lang.ThreadGroup クラスを使います。始めに ThreadGroup インスタンス tg を一つ作ります。そして、Thread のサブクラスにおいて、そのコンストラクタ内で ThreadGroup インスンタス tg を引数とした Thread のコンストラクタを呼びます (super(tg,"");。同時に与える文字列は ThreadGroup の名前を意味します)。このようにすると tg で参照される ThreadGroup インスタンスだけに interrupt メッセージを送ると、すべてのスレッドに interrput メッセージが届きます。

我々が作成するスレッドのクラスは java.lang.Thread のサブクラスですので、もし作成したクラス内で interrupt メソッドがオーバライドされてなければ、java.lang.Thread に送られ、 InterruptedException が発生します。 一方、interrupt メソッドがオーバライドされていれば、それが呼ばれます。ここでは、 interrupt メソッドとして、終了条件を表すフラグを設定した後、*super.interrupt*() により Thread クラスに interrupt メッセージを送ります。こうすると、 Thread.sleep() は定められた時間待たずに InterruptedException 例外を発生します。main 関数では EOF を受け取ったら処理を終え、 ThreadGroup tg に interrupt メッセージを送り、終了します。

このようにして EOF ですべてのスレッドを終了させるプログラムを図 29 に示します。

E.3 課題

1. 図 E.4 のプログラムにおいて、 10 万まで数を表示させる時、分割数をいくつにするのが最適かを調べたいと思います。横軸を分割数、縦軸を所用時間とし、両対数グラフ用紙を使ってグラフを作成しなさい。但し、誤差を考慮するため、各分割数での計測は 3 回以上行い、平均値を使用しなさい。

```
1  class Counter extends Thread {
2      private int from,to;
3      public Counter(int _from, int _to){
4          from=_from;
5          to=_to;
6      }
7      public void run(){
8          for(int i=from;i<=to;i++){
9              System.out.println(i);
10         }
11     }
12 }
```

図 23: Counter クラス

0	1	$\cdots$	q
1	$q+1$	$\cdots$	$2q$
$\vdots$			
$num-1$	$(num-1)q+1$	$\cdots$	$num \cdot q = max$

図 24: $r = 0$ の時

2. スレッドを stop で停止させるのはなぜ危険か、例を挙げて詳しく説明しなさい。

3. 図 29 のプログラムでは、メッセージを for を使って回数だけ表示してました。これに対して、一定時間待った後、メッセージを 1 回だけ出力するだけの java.lang.Thread のサブクラスを作り、このスレッドに対して、 0 秒、10 秒、 ... 90 秒待たせるようにスレッドを作成するようなプログラムに変更しなさい。

0	1	$\cdots$	q	$q+1$
1	$q+1+1$	$\cdots$	$q+1+q$	$2q+2$
$\vdots$				
$r-1$	$(r-1)q+r-1+1$	$\cdots$	$(r-1)q+r-1+q$	$rq+r$
r	$rq+r+1$	$\cdots$	$rq+r+q$	
$r+1$	$(r+1)q+r+1$	$\cdots$	$(r+1)q+r+q$	
$\vdots$				
$num-1$	$(num-1)q+r+1$	$\cdots$	$num \cdot q+r = max$	

図 25: $r > 0$ の時

```
1  class Divider {
2      private int q,r;
3      public Divider(int num, int max){
4          q=max/num;
5          r=max%num;
6      }
7      public int getF(int i){
8          return (i<r) ? i*(q+1)+1 : i*q+r+1;
9      }
10     public int getT(int i){
11         return (i<r) ? (i+1)*(q+1) : i*q+r+q;
12     }
13 }
```

図 26: Divider クラス

```
1  class Main {
2      public static void main(String[] arg){
3          int max=Integer.parseInt(arg[0]);
4          int num=Integer.parseInt(arg[1]);
5          Counter[] c = new Counter[num];
6          Divider div = new Divider(num,max);
7          for(int i=0; i<c.length; i++){
8              c[i]=new Counter(div.getF(i),div.getT(i));
9          }
10         for(int i=0; i<c.length; i++){
11             c[i].start();
12         }
13     }
14 }
```

図 27: 単純な Main クラス

```
1  import java.io.*;
2  class Main {
3      public static void main(String[] arg) throws IOException {
4          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5          String line;
6          while((line=br.readLine())!=null){
7              Writer w = new Writer(line);
8              w.start();
9          }
10     }
11 }
12 class Writer extends Thread {
13     private String message;
14     public Writer(String _message){
15         message=_message;
16     }
17     public void run(){
18         for(int i=1;i<=10;i++){
19             System.out.println(i+": "+message);
20             try{
21                 Thread.sleep(10000);
22             }catch(InterruptedException e){}
23         }
24     }
25 }
```

図 28: メッセージを間欠に出力するプログラム

```
1  import java.io.*;
2  class Main {
3      public static void main(String[] arg) throws IOException {
4          BufferedReader br = new BufferedReader
5              (new InputStreamReader(System.in));
6          ThreadGroup tg = new ThreadGroup("");
7          String line;
8          while((line=br.readLine())!=null){
9              Writer w = new Writer(tg,line);
10             w.start();
11         }
12         tg.interrupt();
13     }
14 }
15 class Writer extends Thread {
16     private boolean active;
17     private String message;
18     public Writer(ThreadGroup tg, String _message){
19         super(tg,"");
20         active=true;
21         message=_message;
22     }
23     public void interrupt(){
24         active=false;
25         super.interrupt();
26     }
27     public void run(){
28         for(int i=1; (i<=10)&&active ;i++){
29             System.out.println(i+": "+message);
30             try{
31                 Thread.sleep(10000);
32             }catch(InterruptedException e){
33                 System.out.println(message+" interrupted");
34             }
35         }
36     }
37 }
```

図 29: 停止可能なスレッドプログラム

E.4 分割数え上げ時間計測プログラム

```
1  import java.util.Date;
2  class Main {
3      public static void main(String[] arg){
4          if(arg.length != 2){
5              System.err.println("Give me two arguments:");
6              System.err.println("first: Max number to count");
7              System.err.println("second: number of threads");
8              System.exit(2);
9          }
10         int max=Integer.parseInt(arg[0]);
11         int num=Integer.parseInt(arg[1]);
12         StopWatch.start();
13         Counter[] counter= new Counter[num];
14         Divider div=new Divider(num,max);
15         for(int i=0; i<c.length; i++){
16             c[i]=new Counter(div.getF(i),div.getT(i));
17         }
18         for(int i=0; i<c.length; i++){
19             c[i].start();
20         }
21         try{
22             for(int i=0; i<c.length; i++){
23                 c[i].join();
24             }
25         }catch(InterruptedException e){}
26         StopWatch.stop();
27         System.out.println(StopWatch.getTime());
28     }
29 }
30 class StopWatch {
31     private static Date from,to;
32     public static void start(){
33         from=new Date();
34     }
35     public static void stop(){
36         to=new Date();
37     }
38     public static double  getTime(){
39         return (double)(to.getTime()-from.getTime())/1000.0;
40     }
41 }
42 class Divider {
43     private int q,r;
44     public Divider(int num, int max){
```

```
45          q=max/num;
46          r=max%num;
47      }
48      public int getF(int i){
49          return (i<r) ? i*(q+1)+1 : i*q+r+1;
50      }
51      public int getT(int i){
52          return (i<r) ? (i+1)*(q+1) : i*q+r+q;
53      }
54  }
55  class Counter extends Thread {
56      private int from,to;
57      public Counter(int _from, int _to){
58          from=_from;
59          to=_to;
60      }
61      public void run(){
62          for(int i=from;i<=to;i++){
63              System.out.println(i);
64          }
65      }
66  }
```

F　RFC1945 HTTP/1.0 の抜粋

F.1　メッセージのやりとりの基本

HTTP/1.0[2] は**ステートレス**な (状態のない) プロトコルです。クライアントの出す一つの**リクエスト**に対して、単純に一つだけ**レスポンス**を返します。メッセージのフォーマットは RFC1036[6] で規定される形式で、これは電子メールの形式 [1] に準拠しています。ただし、HTML/1.0 ではリクエスト、レスポンスともメッセージの一行目に特別な行を拡張します。なお、*RFC1036* **メッセージ**とは、ヘッダ部とボディ部が一つの空行で区切られており、ヘッダ部は各行が「**フィールド名**:**フィールドの値** CRLF」という形式になっているものです。ここで *CR*(復帰) は 0x0d, *LF*(改行) は 0x0a の文字コードで表される文字で、この二文字の組でメッセージの改行を表します。この改行の方式は Microsoft Windows のテキストファイルの改行と一致しています (MacOS や UNIX では異なります)。

F.2　リクエストメッセージ

ヘッダの一行目は「**リクエスト行**」と呼ばれ、「**メソッド** (一つの空白) **リクエスト** *URI*(一つの空白)HTTP バージョン CRLF」という構造になっています。「メソッド」には GET, HEAD, POST(すべて英大文字で表記) が定義されてますが、本実験では GET のみを実装することにします。*GET* **メソッド**は「リクエスト URI」で示すデータを取り出すことを意味します。「リクエスト URI」 は /で始まる「絶対パス」と言う表現で指定するもので通常は特定のファイルを指します。「HTTP バージョン」は HTTP/1.0(英大文字のみ) です。したがって、 index.html というファイルを要求するリクエスト行は図 30 のようになります。

この後、 RFC1036 メッセージが続くことになりますが、POST メソッド以外はそれほど重要な意味を持ちません。付けた方が良さそうなヘッダとしては User-Agent フィールド (利用者の使用するプログラムを名乗る) がありますが、これは必須ではありません。したがって、ヘッダ、ボディとも空で構わないので、区切りである空行のみが最小の RFC1036 メッセージになります。したがって、上記の index.html を要求する最低限のリクエストメッセージは図 31 のようになります。

F.3　レスポンスメッセージ

最小のリクエストメッセージはリクエスト行と空行のみで済みましたが、**レスポンスメッセージ**はそれほど簡単ではありません。まずレスポンスメッセージの最初の行は「*Status* **行**」と呼ばれる行が必要です。これは「HTTP バージョン (空白)*Status* **コード** (空白) 理由を示す言葉 CRLF」という書式になっています。ここでまず、HTTP バージョンは HTTP/1.0 になります。一方、Status コードは 3 桁の数字です。一番上の桁はおおまかな意味を表し、下の二桁は通し番号を表します。但し、下二桁が 00 の状態はメタな意味 (他に含まれるメッセージを統括する意味) になってます。Status コードの一番上の桁は次のような意味です。

1xx 情報 (未使用)

```
GET /index.html HTTP/1.0
```

図 30: HTTP/1.0 でのリクエスト行

```
GET /index.html HTTP/1.0
空行
```

図 31: HTTP/1.0 の最も基本的なリクエスト

```
Content-Type: text/plain ; charset=Shift_JIS
```

図 32: Content-Type の設定

2xx 成功

3xx 転送

4xx クライアントの誤り

5xx サーバの誤り

ここでは最小のサーバを作るだけのメッセージを紹介します。重要なメッセージを列挙します。200 は Ok という意味です。アクセスが成功したことを示します。400 は Bad Request という意味になります。リクエストメッセージの構文がおかしい場合などはこれに当たります。要求した情報がない場合も 400 BadRequest を返してもいいのですが、404 Not Found という専用のメッセージが用意されています。500 は Internal Server Error です。サーバがエラーを出さなければならない状態ではこのエラーを返します。但し、リクエストメッセージは正しいが、指定されているメソッドに対応してない場合 (本実験の場合 GET 以外)、501 Not Implemented を返すべきです。つまり、最小のサーバが送り返す Status コードとその意味をまとめると次のようになります。

200 Ok

400 Bad Request

404 Not Found

500 Internal Server Error

501 Not Implemented

この Ok などの言葉は「理由を示す言葉」として Status 行に埋め込まれます。

次に Status 行に続く RFC1036 メッセージですが、最低でも次のフィールドが必要です。

Date 情報を作成した日時 (転送を始めた日時ではない)

Content-Length 情報の長さ

Content-Type 情報の型

Date フィールドには情報を作成した日時を入れます。書式は「Sun, 06 Nov 1994 08:49:37 GMT」のようにします。日時は日本標準時 (JST) ではなく、必ず世界時 (UT) で表現します。但し歴史的理由から世界時を表す時は UT の代わりに GMT と表示します。

Content-Length フィールドは情報の長さを 10 進法で表現します。

Content-Type フィールドには送る情報の型を表示します。これは RFC2046[5] で規定されているものです。テキストファイルだったら text/plain、HTML 文書なら text/html、JPEG 画像だったら image/jpeg などと指定します。Content-type が決まらない場合、デフォルト値 application/octet-stream を指定します。特に Content-type が text の場合、文字コードが重要になります。Shift_JIS(マイクロソフト漢字コード) のテキストファイルだと図 32 のように指定します.

テキストファイルの文字コードをファイルの内容から自動的に判別するには限界がありますので、Content-Type フィールド の text/plain のファイルの charset をサーバが自動的に決定することは難しいです。一方、HTML[13] 文書の場合、*meta* **要素**により、Content-Type を指定することができますので、サーバ側の立場としては、文書中から *HTTP-EQUIV* **属性**が Content-Type である meta 要素を探し、 *CONTENT* **属性**を得る必要があります。なお HTML 文書のデフォルトの文字コードは西ヨーロッパ圏の言語を表示できる *iso-8859-1* です。一方、 *XHTML* などの *XML* **文書** のデフォルトの文字コード は *UTF-8* または *UTF-16* です。

F.4 補足

WWW の標準化団体 World Wide Web Consortium(W3C) は HTTP/1.0 を使用しないように勧めています [9]。これは HTTP/1.0 にはスケーラビリティとパフォーマンスに関して深刻な問題を抱えているのが理由です。したがって新しいアプリケーションは HTTP/1.1[3] を用いるべきであるとしています。

本実験では単純なクライアント/サーバシステムとして HTTP/1.0 を選びました。この実験を応用して実用的なサービスを行うプログラムなどを制作するには HTTP/1.1 などの上位バージョンのプロトコルを使用して下さい。

HTTP/1.1 では次のような機能が追加されてます。

- **持続性接続** (指定しない限り接続が維持される)
- **ヴァーチャルホスト** (一台のサーバが複数の Web サーバのように振舞う)

これを実現するために、 HTTP/1.1 のやりとりには次のヘッダ行が必要になります。

Connection 値 *close* を指定することで、レスポンスが終了した後、接続を終了します。

もし、この持続性接続をサポートしないアプリケーションでは常にこの Connection: close 行をヘッダに含める必要があります。

Host HTTP/1.1 メッセージではクライアント、サーバとも Host ヘッダが必須になります。もし、クライアントから Host ヘッダを含まないメッセージが来た場合、サーバは 400 Bad Request を送らなければなりません。

また、サーバは Request URI として、絶対 URI(サーバ名を含む URI) を受け付ける必要があります。

したがって、 HTTP/1.1 サーバはこのヘッダの正当性の確認や、持続性接続の処理、ヴァーチャルホストの処理をする必要があります。

G BenchSample.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  class StopWatch {
5      private static Date from;
6      private static Date to;
7      public static void start(){
8          from=new Date();
9      }
10     public static void stop(){
11         to=new Date();
12     }
13     public static double  getTime(){
14         return (double)(to.getTime()-from.getTime())/1000.0;
15     }
16 }
17 class BenchSample {
18     public static void main(String argv[]) throws MalformedURLException,IOException{
19         StopWatch.start();
20         for(int i=0;i<=Integer.parseInt(argv[3]);i++){
21             Socket sock = new Socket(InetAddress.getByName(argv[0]),
22                                      Integer.parseInt(argv[1]));
23             InputStream is = sock.getInputStream();
24             PrintWriter pr = new PrintWriter(sock.getOutputStream());
25             pr.println("GET␣"+argv[2]+"␣HTTP/1.0");
26             pr.println();
27             pr.flush();
28             int c;
29             while((c=is.read())!=-1){
30                 //System.out.write(c);
31             }
32             sock.close();
33         }
34         StopWatch.stop();
35         System.out.println(StopWatch.getTime());
36     }
37 }
```

H Bench.java

```
1  import java.io.*;
2  import java.net.URL;
3  import java.net.MalformedURLException;
4  import java.util.Date;
5  class StopWatch {
6      private static Date from;
7      private static Date to;
8      public static void start(){
9          from=new Date();
10     }
11     public static void stop(){
12         to=new Date();
13     }
14     public static double  getTime(){
15         return (double)(to.getTime()-from.getTime())/1000.0;
16     }
17 }
18 class Bench {
19     public static void main(String argv[])
20             throws MalformedURLException,IOException{
21         StopWatch.start();
22         for(int i=0;i<=Integer.parseInt(argv[1]);i++){
23             URL url= new URL(argv[0]);
24             InputStream is = url.openStream();
25             int size;
26             do{
27                 size = is.available();
28             }while(is.read(new byte[size])!=-1);
29             is.close();
30         }
31         StopWatch.stop();
32         System.out.println(StopWatch.getTime());
33     }
34 }
```

I HTTP サーバ Http.java

```
1  import java.net.*;
2  import java.io.*;
3  import java.text.*;
4  import java.util.*;
5  class Constants {
6      private Constants(){}
7      final public static int PORT=8080;
8      final public static String DOCUMENTROOT="htdocs/";
9      final public static final String SERVERNAME="Sakamoto Server ver.1.2";
10 }
11 class Http {}
12     public static void main(String[] argv) throws IOException{
13         final ServerSocket servs = new ServerSocket(Constants.PORT);
14         for(;;){
15             Socket sock = servs.accept();
16             BufferedReader sockbr
17                 = new BufferedReader(
18                     new InputStreamReader(sock.getInputStream()));
19             PrintStream sockps = new PrintStream(sock.getOutputStream());
20             Reply rep=new Reply(new Request(sockbr));
21             rep.send(sockps);
22             sock.close();
23         }
24     }
25 }
26 class Reply {
27     final private ReplyData repData;
28     public Reply(Request req){
29         ReplyData tmpRepData
30         try{
31             tmpRepData = new FileData(new File(Constants.DOCUMENTROOT+req.read()));
32         }catch(FileNotFoundException e){//ファイル無し
33             tmpRepData = ErrorMessage.getInstance(404);
34         }catch(SyntaxException e){//ヘッダ書式エラー
35             tmpRepData = ErrorMessage.getInstance(400);
36         }catch(ProtocolException e){//プロトコルエラー
37             tmpRepData = ErrorMessage.getInstance(400);
38         }catch(NotImplementedException e){//メソッド not implement
39             tmpRepData = ErrorMessage(501);
40         }catch(Exception e){ //報告されないエラー
41             tmpRepData = ErrorMessage(500);
42         }
43         repData = tmpRepData;
44     }
```

```
45      public void send(PrintStream ps) throws IOException{
46          repData.send(ps);
47      }
48  }
49
50  abstract class ReplyData {
51      final protected int errorCode;
52      final protected String errorMessage;
53      public ReplyData(int errorCode, String errorMessage){
54          this.errorCode = errorCode;
55          this.errorMessage = errorMessage;
56      }
57      public String getHeader(){
58          return "HTTP/1.0 "+errorCode+" "+errorMessage;
59      }
60      public abstract String getLastModified();
61      public abstract long getContentLength();
62      public abstract String getContentType();
63      protected void sendHeader(PrintStream ps) throws IOException{
64          ps.println(getHeader());
65          ps.println("Date: "+getLastModified());
66          ps.println("Content-Length: "+getContentLength());
67          ps.println("Content-Type: "+getContentType());
68          ps.println("Server: "+Constants.SERVERNAME);
69          ps.println();
70      }
71      public abstract void send(PrintStream ps) throws IOException;
72  }
73  class ErrorMessage extends ReplyData {
74      private static HashMap<Integer,ErrorMessage> errorTable;
75      final private static String[][] errorCodeTable
76          ={{"400","Bad Request"},
77            {"404","File Not Found"},
78            {"501","Not Implemented"},
79            {"500","Internal Server Error"}};
80      public static ErrorMessage getInstance(int error){
81          if(errorTable==null){
82              errorTable
83                  =new HashMap<Integer,ErrorMessage>(errorCodeTable.length);
84              for(String[] ec : errorCodeTable){
85                  errorTable.put(new Integer(ec[0]),
86                                 new ErrorMessage(Integer.parseInt(ec[0]),ec[1]));
87              }
88          }
89          if(errorTable.containsKey(error)){
90              return errorTable.get(error);
```

```
91          }else{
92              return errorTable.get(500);
93          }
94      }
95      private ErrorMessage(int errorCode, String errorMessage){
96          super(errorCode, errorMessage);
97      }
98      public String getLastModified(){
99          return Rfc822.getDateFormat().format(new Date());//現在時刻
100     }
101     public long getContentLength(){
102         return errorMessage.length();
103     }
104     public String getContentType(){
105         return "text/plain";
106     }
107     public void send(PrintStream ps) throws IOException{
108         this.sendHeader(ps);
109         ps.println(errorMessage);
110     }
111
112 }
113 class FileData extends ReplyData {
114     final private File file;
115     final private FileInputStream fis;
116     public FileData(File f) throws FileNotFoundException{
117         super(200,"Ok");
118         file = f;
119         fis = new FileInputStream(f);
120     }
121     public String getLastModified(){
122         return Rfc822.getDateFormat().format(new Date(file.lastModified()));
123     }
124     public long getContentLength(){
125         return  file.length();
126     }
127     public String getContentType(){
128         return file.getName().endsWith(".txt") ? "text/plain"
129             : "application/octet-stream";
130     }
131     public void send(PrintStream ps) throws IOException{
132         this.sendHeader(ps);
133         FileInputStream fis = new FileInputStream(file);
134         int c;
135         while((c=fis.read())!=-1){
136             ps.write((char)c);
```

```
137         }
138         fis.close();
139     }
140 }
141 class Rfc822 {
142     private static DateFormat rfc822;
143     public static DateFormat getDateFormat(){
144         if(rfc822==null){
145             rfc822=new SimpleDateFormat("EE, dd MMM yyyy HH:mm:ss zz",Locale.US);
146             rfc822.setCalendar(Calendar.getInstance(TimeZone.getTimeZone("GMT")));
147         }
148         return rfc822;
149     }
150 }
151 class Request {
152     final private BufferedReader br;
153     public Request(BufferedReader br){
154         this.br = br;
155     }
156     public String read() throws SyntaxException, ProtocolException,
157                                 NotImplementedException, IOException {
158         final String filename=readFirstLine();
159         int contentLength=readHeader();
160         while(contentLength>0){
161             contentLength-=readBody();
162         }
163         return filename;
164     }
165     private String readFirstLine() throws SyntaxException,
166                 ProtocolException, NotImplementedException,IOException {
167         final String line=br.readLine();
168         final StringTokenizer st = new StringTokenizer(line," ");
169         if(st.countTokens()!=3)
170             throw new SyntaxException();
171         final String method=st.nextToken();
172         final String filename=st.nextToken();
173         final String protocol=st.nextToken();
174         if(!protocol.startsWith("HTTP/1."))
175             throw new ProtocolException();//プロトコルエラー
176         if(!method.equals("GET"))
177             throw new NotImplementedException();//メソッド not implemented
178         return filename;
179     }
180     private int readHeader() throws SyntaxException,IOException{
181         int contentLength=0;
182         String line;
```

```
183         while((line=br.readLine()).length()!=0){
184             StringTokenizer st = new StringTokenizer(line,":");
185             // 継続行は処理しない
186             if(st.countTokens()<2)
187                 throw new SyntaxException(); //ヘッダのフォーマットエラー
188             if(st.nextToken().equalsIgnoreCase("content-length")){
189                 try{
190                     contentLength=Integer.parseInt(st.nextToken());
191                 }catch(NumberFormatException e){
192                     throw new SyntaxException();//ヘッダのフォーマットエラー
193                 }
194             }
195         }
196         return contentLength;
197     }
198     private int readBody() throws IOException{
199         return br.readLine().length();
200     }
201 }
202 class SyntaxException extends Exception {
203     public SyntaxException(){}
204     public SyntaxException(String message){
205         super(message);
206     }
207 }
208 class ProtocolException extends Exception {
209     public ProtocolException(){}
210     public ProtocolException(String message){
211         super(message);
212     }
213 }
214 class NotImplementedException extends Exception {
215     public NotImplementedException(){}
216     public NotImplementedException(String message){
217         super(message);
218     }
219 }
```

J Greeting.java

```
1  import java.io.*;
2  import java.net.*;
3  class Constant {
4      private Constant(){}
5      final public static String MSG="Naoshi has received your message.";
6      final public static int PORT=49152;
7      final public static int BUFSIZE=1024;
8  }
9  class Sender {
10     final private DatagramSocket sock;
11     private static Sender sender;
12     private Sender() throws SocketException {
13         sock = new DatagramSocket();
14     }
15     public static Sender getInstance() throws SocketException {
16         if(sender == null){
17             sender = new Sender();
18         }
19         return sender;
20     }
21     public void sendMessage(String str, InetAddress ipaddr)
22         throws IOException {
23         final byte data[]=str.getBytes();
24         final DatagramPacket pack
25             = new DatagramPacket(data, data.length, ipaddr, Constant.PORT);
26         sock.send(pack);
27     }
28     public void close(){
29         sock.close();
30         sender = null;
31     }
32 }
33 class Greeting {
34     public static void main(String arg[])
35             throws UnknownHostException,SocketException,IOException{
36         final InetAddress toaddr = InetAddress.getByName(arg[0]);
37         final Sender sender = Sender.getInstance();
38         final Server serv=new Server();
39         serv.start();
40         final BufferedReader br = new BufferedReader(
41                     new InputStreamReader(System.in));
42         String line;
43         while((line=br.readLine())!=null){
44             sender.sendMessage(line, toaddr);
```

```
45          }
46          serv.halt();
47          sender.close();
48      }
49  }
50  class Server extends Thread {
51      final private DatagramSocket receiver;
52      final private Sender sender;
53      private volatile boolean ok;
54      public Server() throws SocketException{
55          sender = Sender.getInstance();
56          receiver = new DatagramSocket(Constant.PORT);
57          receiver.setSoTimeout(1000);
58          ok=true;
59      }
60      public void halt(){
61          ok=false;
62      }
63      private void printPacket(DatagramPacket pack){
64          System.out.print("from ");
65          System.out.print(pack.getAddress().getHostAddress());
66          System.out.print(": ");
67          System.out.println(packToString(pack));
68          System.out.flush();
69      }
70      private String packToString(DatagramPacket pack){
71          return new String(pack.getData());
72      }
73      public void run(){
74          final DatagramPacket pack
75              = new DatagramPacket(new byte[Constant.BUFSIZE], Constant.BUFSIZE);
76          while(ok){
77              try{
78                  pack.setData(new byte[Constant.BUFSIZE]);
79                  receiver.receive(pack);
80                  printPacket(pack);
81                  if(!packToString(pack).startsWith("+")){
82                      sender.sendMessage("+"+Constant.MSG, pack.getAddress());
83                  }
84              }catch(InterruptedIOException e){
85              }catch(IOException e){
86                  System.exit(1);
87              }
88          }
89      }
90  }
```

K　特定のサイズのファイルを作る LargeFile.java

```
1  import java.io.*;
2  class LargeFile {
3      public static void main(String[] arg) throws
4                      FileNotFoundException,IOException {
5          if(arg.length!=2){
6              System.err.println("Usage: java LargeFile (filename) (filesize)");
7              System.exit(1);
8          }
9          String filename=arg[0];
10         int totalsize=Integer.parseInt(arg[1]);
11         DataOutputStream dos = new DataOutputStream(
12                                     new FileOutputStream(filename));
13         for(int i=0; i<totalsize ; i++){
14             dos.write((byte)(Math.random()*256));
15         }
16     }
17 }
```

参考文献

[1] P. Resnick, editor. Rfc2822: Internet message format, April 2001. Obsolates RFC822. Status: Standards Track.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL.

[3] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet Request for Comment RFC 2616, 1999. Obsoletes RFC2068. Status: Standards Track.

[4] David Flanagan. Java クイックリファレンス. オライリー・ジャパン, 第 4 版, 2003. 訳: イデア コラボレーションズ株式会社.

[5] N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types, November 1996. Obsoletes RFC1521, RFC1522, RFC1590. Status: DRAFT STANDARD.

[6] M. R. Horton and R. Adams. RFC 1036: Standard for interchange of USENET messages, December 1987. Obsoletes RFC0850. Status: UNKNOWN.

[7] Internet corporation for assigned names and numbers. http://www.icann.org/.

[8] Japan network information center. http://www.nic.ad.jp/.

[9] Yves Lafon. Http - hypertext transfer protocol. http://www.w3.org/Protocols/Specs.html.

[10] J. Postel. RFC 768: User datagram protocol, August 1980. Status: STANDARD. See also STD0006.

[11] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760. See also STD0005. Status: STANDARD.

[12] J. Postel. RFC 793: Transmission control protocol, September 1981. See also STD0007. Status: STANDARD.

[13] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, editors. *HTML 4.01 Specification*. W3 Consortium Recommendation, December 1999. Available from http://www.w3.org/TR/html4/.

[14] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. RFC 1918: Address allocation for private internets, February 1996. Obsoletes RFC1627, RFC1597. Status: BEST CURRENT PRACTICE.

[15] D. Waitzman. RFC 2549: IP over avian carriers with quality of service, April 1999. Obsoletes RFC1149. Status: EXPERIMENTAL.

[16] 志村拓, 榊隆. インターネットを 256 倍使うための本, 第 1 巻. アスキー出版局, 1996.

[17] 志村拓, 榊隆, 岩井潔. インターネットを 256 倍使うための本, 第 2 巻. アスキー出版局, 1997.

索 引