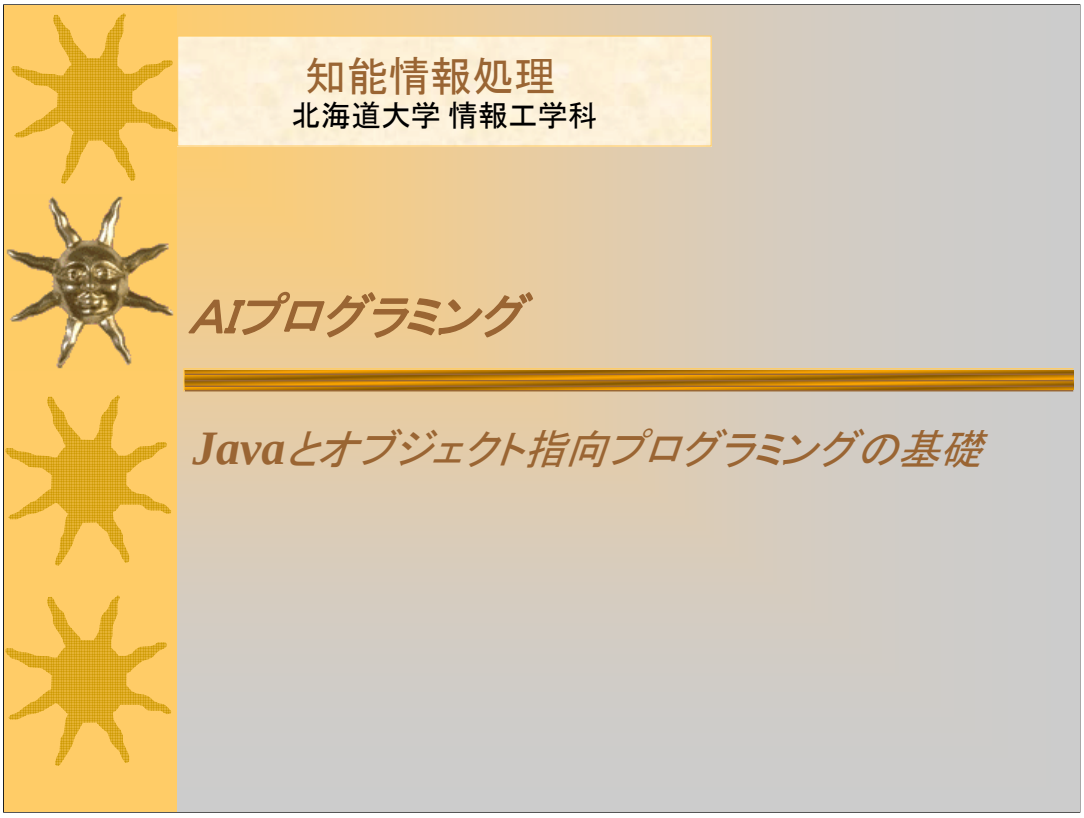
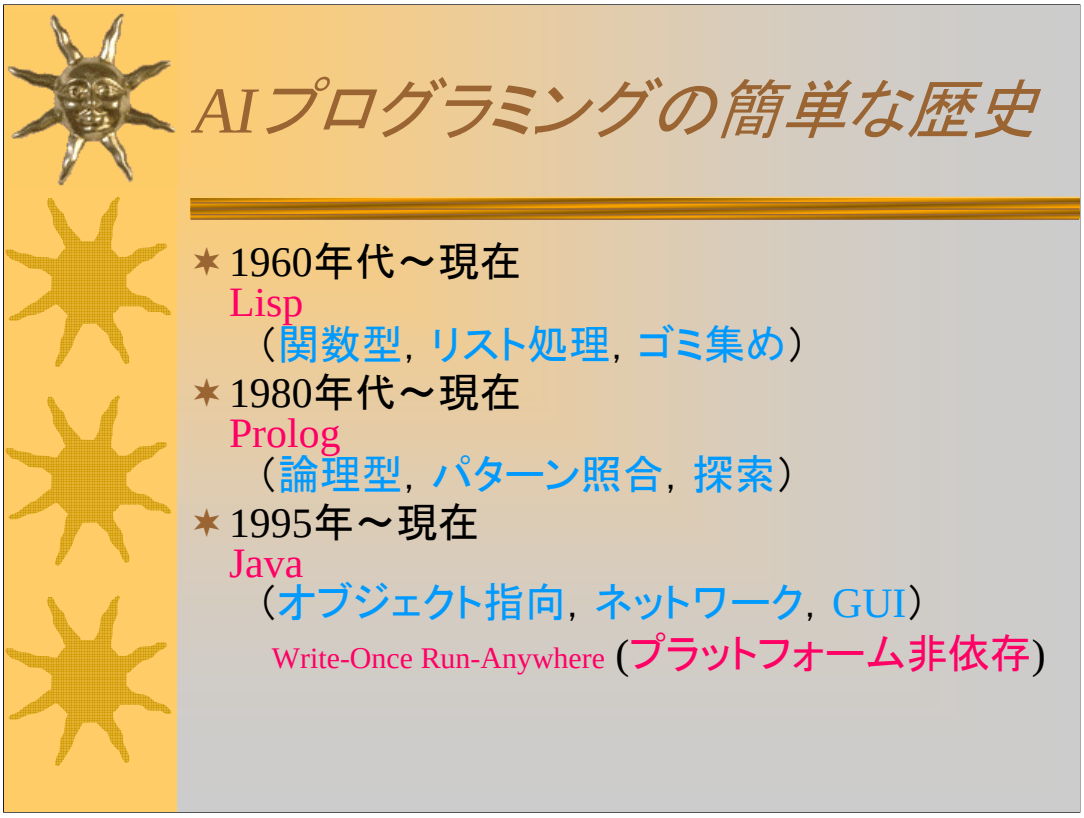


「知能情報処理」の授業では，人工知能のさまざまなアルゴリズムを学ぶことになるが，可能ならばそれを実際のプログラミング言語で実装してみることが大事である．

しかし，この授業では，時間の関係でそこまで扱うことができないので，興味のある人が将来，自立的に実装できるように，最小限のプログラミングの知識を今回の授業で学ぶ．



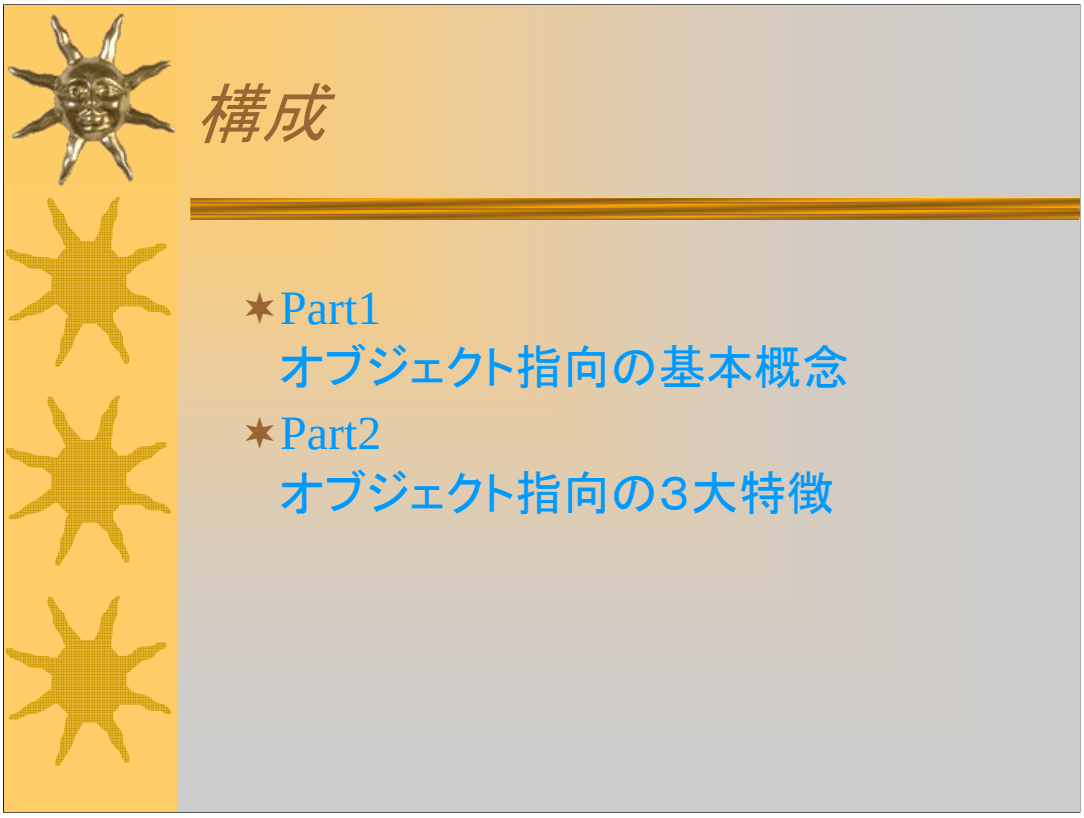
人工知能の歴史の中で，初期のころ（1960年代以降）は，AIプログラミングといえば，それ専用のプログラミング言語を使うものと相場が決まっていた．それは基本的には **LISP** という**関数型**の**リスト処理**言語だった．**ゴミ集め**(garbage collection)という革新的な機能はバグを減らすのに効果的で，現代のJavaに受け継がれている．

1980年代になると**論理型**言語である **Prolog** も仲間に加わり，**パターン照合**や**探索**といわれる処理の初歩的な機能が言語に組み込まれた．

しかし，その後，プログラミング言語の研究が発展してきたことや，AIの研究テーマも多岐にわたってきたことから，最近では，通常のプログラミング言語でも十分AIプログラムを書けるようになってきた．

それは，プログラミング言語 `Java` の登場である．

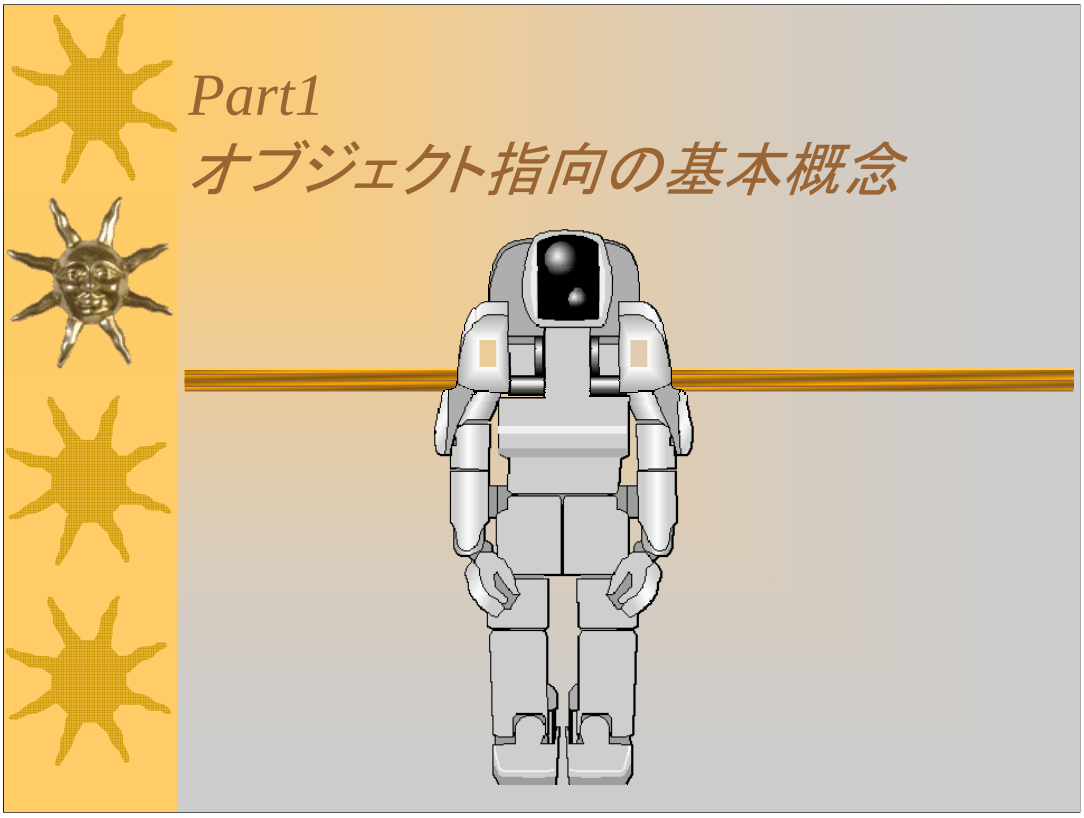
その特徴は，**オブジェクト指向**による再利用性の高いプログラミングができること．インターネット時代を反映して，**ネットワークプログラミング**が容易にできること．グラフィカル・ユーザ・インタフェース（**GUI**)のプログラミング機能が充実していることなどである．しかし，最大の特徴は **Write-Once Run-Anywhere**（一度書けば，どこででも走る）というキャッチフレーズで表現される可搬性（ポータビリティ），あるいはOSやコンパイラなどによらない**プラットフォーム非依存性**と呼ばれる性質で，一度作ったプログラムの実行コードを再コンパイルなしに事実上どこででもただちに実行できることである．

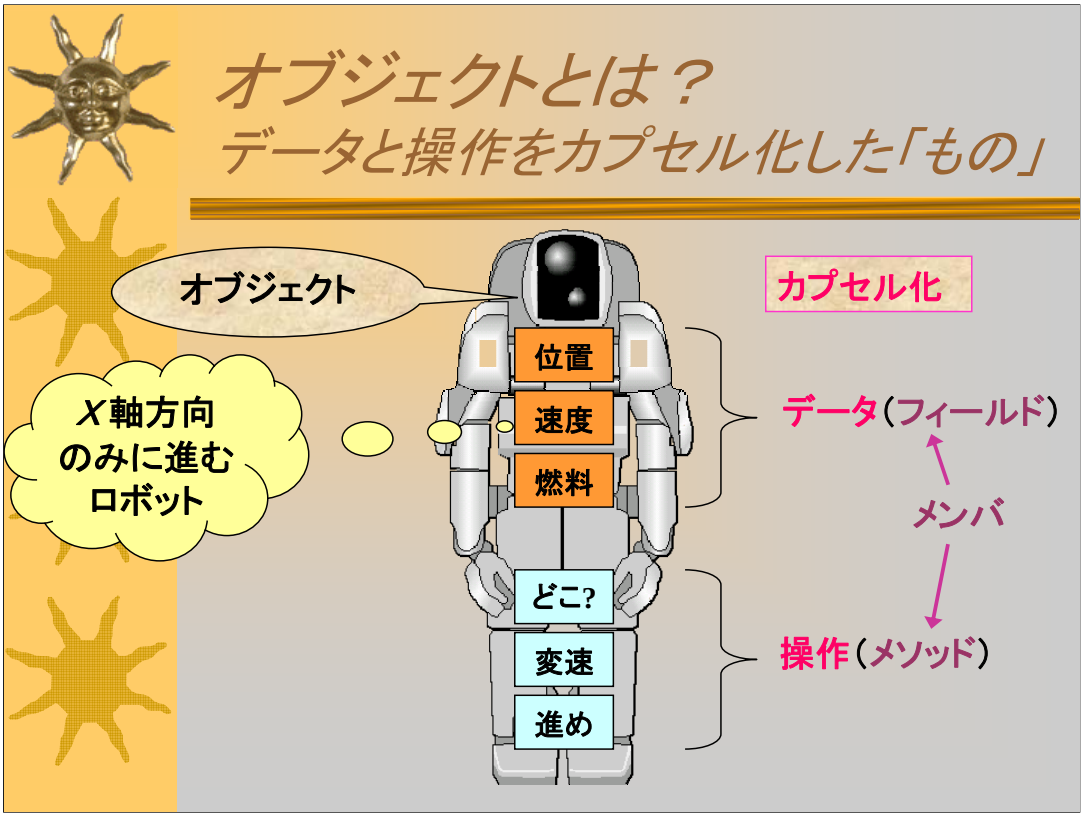


今回の授業では，オブジェクト指向プログラミングの最も基本的な考え方を学ぶ．

具体的なプログラミングはJavaを用いるが，Javaの詳細を学ぶことが目的ではない．

Part1でオブジェクト指向の基本概念を説明した後，Part2でオブジェクト指向の３大特徴を学ぶ．


Part1
オブジェクト指向の基本概念



オブジェクトとは何か？

その答えを短く言うと，「**データと操作をカプセル化した「もの」**」となる．

とはいえ，それで内容が理解できるとは思えないので，順を追って内容を見ていく．

この授業では，AI的な例題として，このロボットを動かすプログラムを考える．このロボットが「**オブジェクト**」のつもりである．

このロボットはx軸上を進むものである．

ロボットの状態は，現在の「位置」，現在の「速度」，現在の「燃料」の3つのデータで表すことができる．それらのデータはロボットの胸のあたりに内蔵されている．

このロボットはリモコンの3つの操作を受け付ける．

1つは，「どこ？」というボタンを押すことで，現在位置（x座標の値）を返してくれる．（それは，たとえば，リモコンの表示画面に表示される．）

2つめは，「変速」というボタンを押して，速度を設定できる．実際には，このボタンを押すほかに，さらに設定すべき速度の値を補助情報として入力する必要がある．

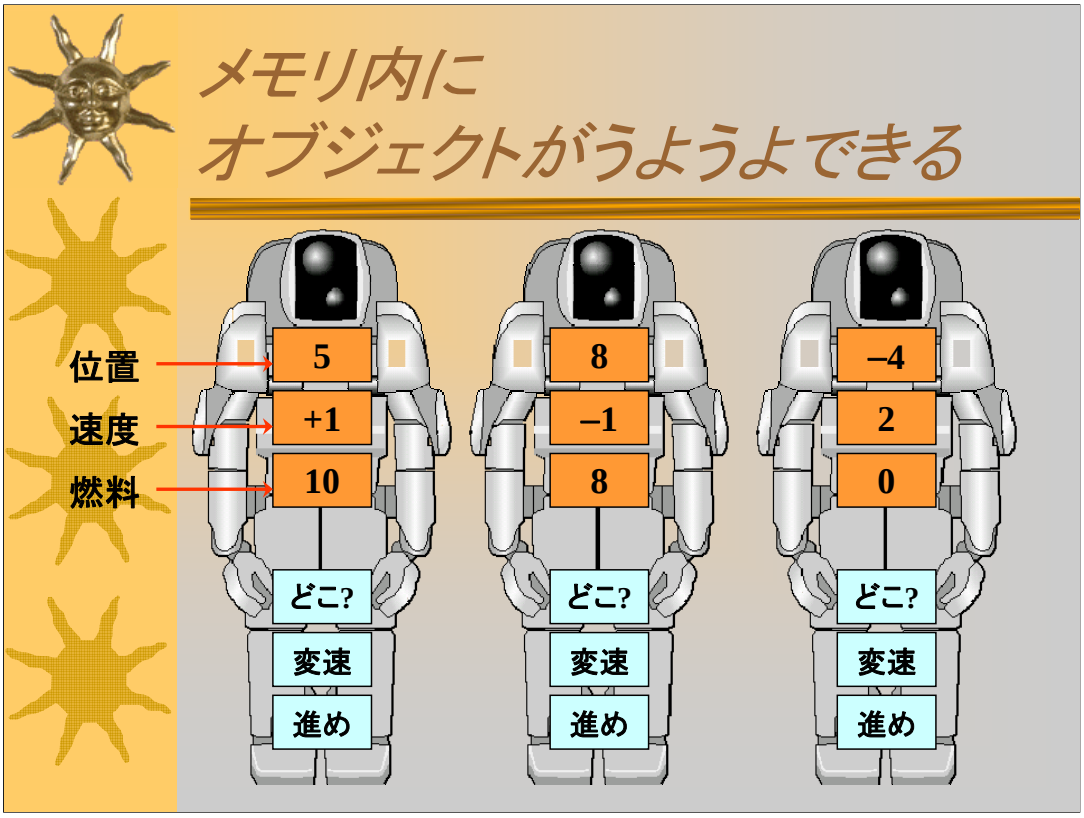
3つめは，「進め」ボタンで，これを押すと，1秒間だけ，現在の設定速度でロボットが前進する．

リモコンからのこれらの指示を実行するためのプログラムがロボットの足のあたりに内蔵されている．

オブジェクト指向プログラミングの用語では，いま述べた「データ」のことを「**フィールド**」，「操作」のことを「**メソッド**」という．「フィールド」と「メソッド」を総称して「**メンバ**」ということもある．

一般的なプログラミング用語で言えば，フィールドは変数，メソッドは関数のようなもので，いずれもディジタルなデータあるいはコードで表現できる．それらが，このロボットという「オブジェクト」の中にまとめて閉じこめられている様子を「**カプセル化**」というのだが，もっと正確な説明は後にわかってくる．

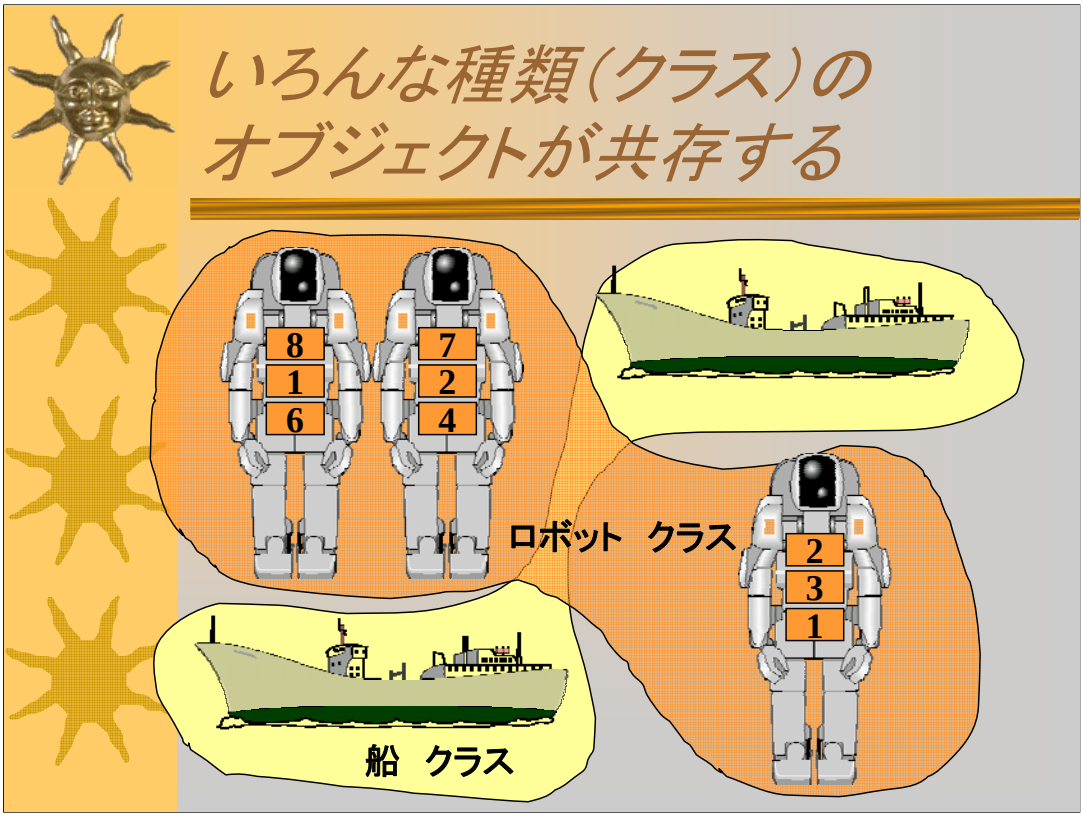
重要なのは，「オブジェクト」というのは単なる概念ではなく，これらのディジタルなデータまたはコードからなり，コンピュータのメモリの一定領域を占める具体的な「実体」だということである．この「実体」あるいは「もの」が英語の　object　という単語の意味である．



「オブジェクト」は一定の実メモリ領域を占める「もの」であるが，ふつう，オブジェクト指向でプログラムを書くと，そのようなオブジェクトがたくさん生成される．それぞれのオブジェクトごとにメモリ領域が占められる．

ロボットの場合には，このスライドのように，状態（位置，速度，燃料のそれぞれの値）の異なるロボットが生成されるわけである．

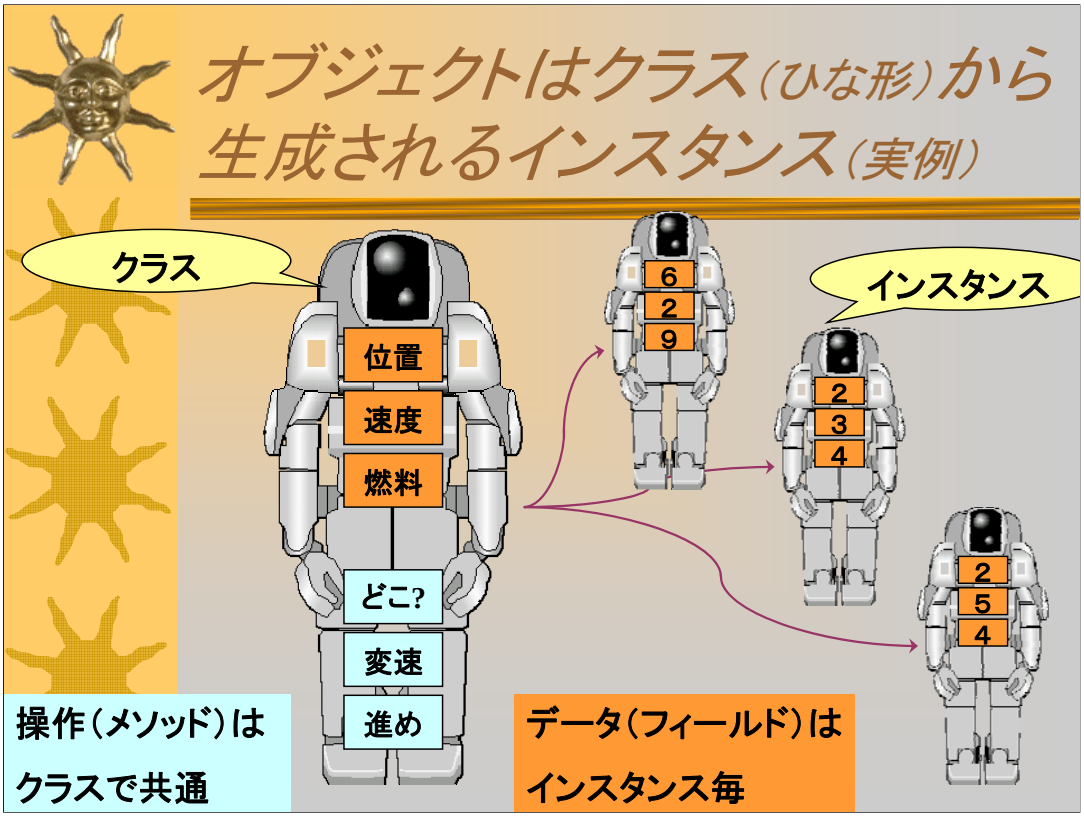
ただし，操作に相当するプログラムコードはこれらのオブジェクトに共通である．



オブジェクトにはいろいろな種類のものがあってよい.

たとえば, オブジェクト指向で書かれたあるコンピュータゲームの中には, ロボットのほかに船が出てきて, ロボットが船に乗って, いろいろな島に渡って冒険をするのかもしれない. この場合, 「ロボット」と「船」は異なる種類のオブジェクトである.

オブジェクトの種類のことを, 正式には「**クラス**」という. この例の場合, ロボットクラス, 船クラスという2つのクラスのオブジェクトが存在することになる.



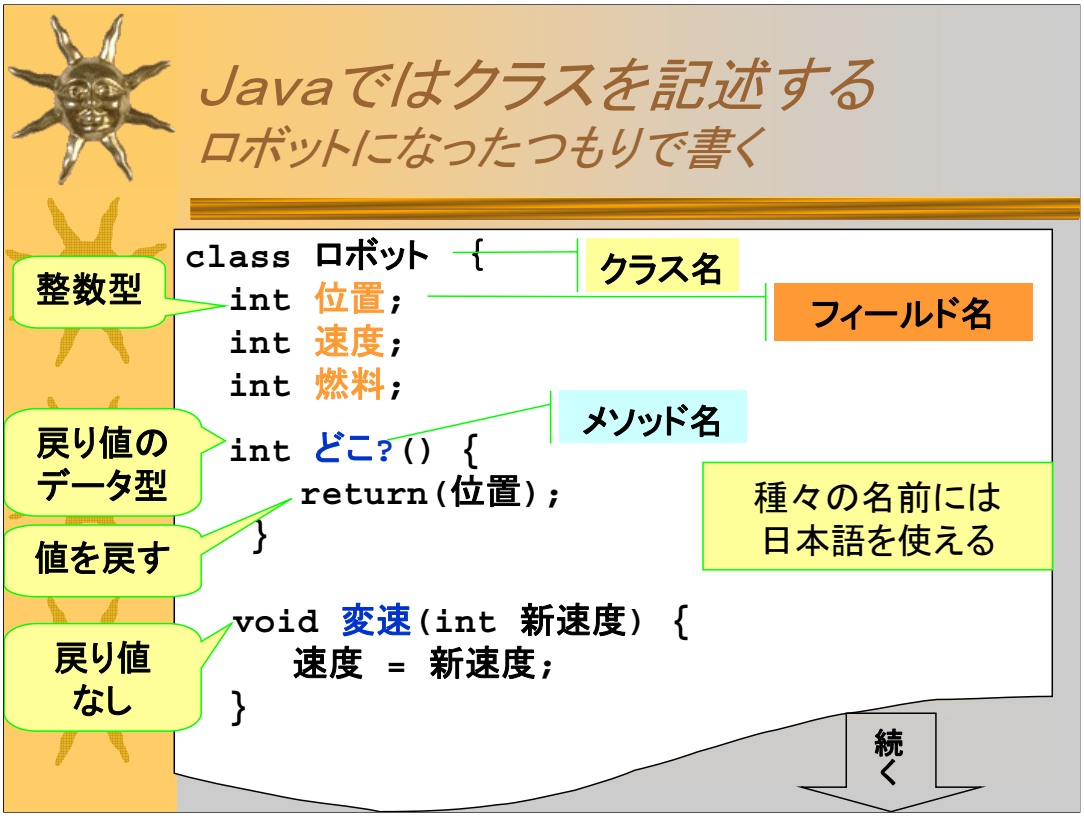
オブジェクト指向プログラミングで初心者がまずつまづく点は，「クラス」と「インスタンス」の区別である．

「**クラス**」とは，オブジェクトを作るための「**ひな形**」あるいは「**設計図**」のようなものである．ロボットの例の場合，「位置」，「速度」，「燃料」というフィールドがあることがこの設計図に書かれている．しかし，当然だが，その「現在の値」というのはない．

一方，「**インスタンス**」とは，この設計図から作られた「**実例**」のことであり，各フィールドには「現在の値」が記録されている．このスライドでは，1つのクラスから異なる3つのインスタンスが生成されている．これまで出てきた「オブジェクト」という言葉は，この「インスタンス」と同じ意味である．

メソッドの具体的なコードもクラスに記述されている．このクラスからインスタンスを生成すると，各インスタンスにはこれらのメソッドがそのままコピーされる．（ただし，これは概念上の話で，実際には，同じものをたくさんコピーしておくのは無駄が多いので，ポインタなどを用いて効率的に実装されている．）

以上の結果，操作(メソッド)はクラスで共通に使用され，データ(フィールド)はインスタンス毎に異なるものとして使われることになる．



これがこれまで設計したロボットをJavaで記述したものである.

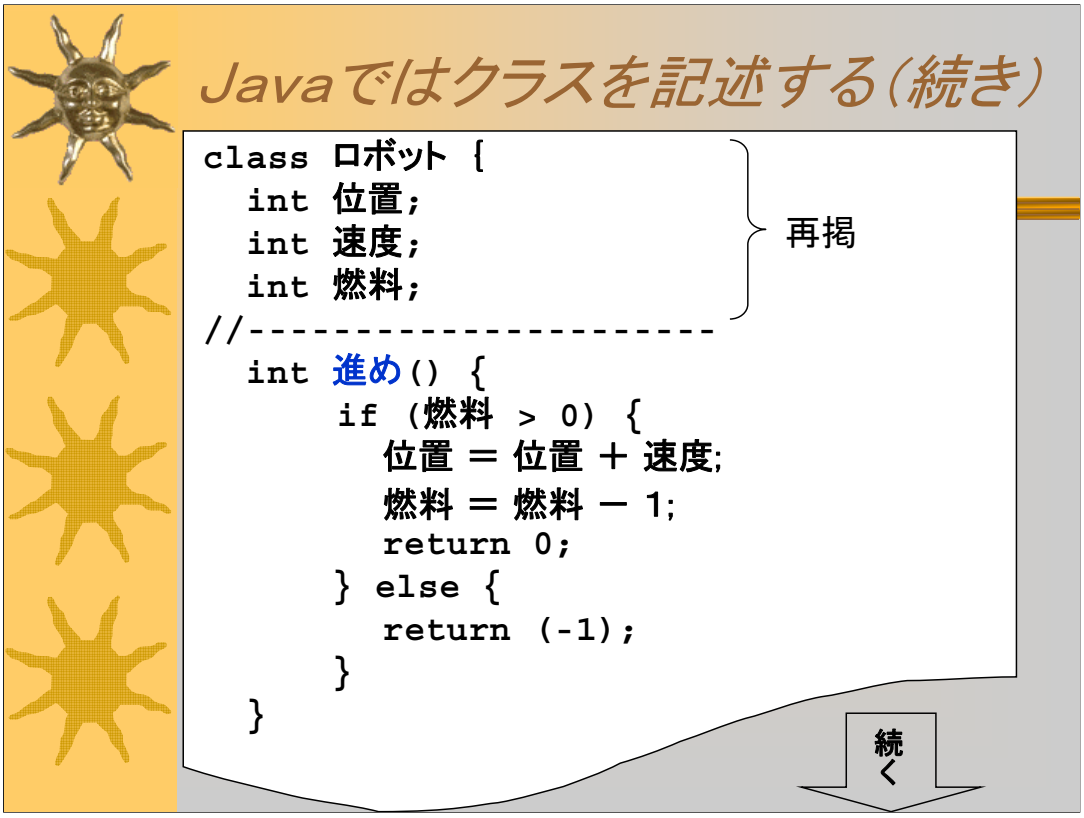
Javaで記述するものは，実際にはクラスである. ここではクラス名を「ロボット」としている.

フィールドは，従来からある他のプログラミング言語（たとえば，C言語）における変数宣言のような形で宣言される.

メソッドは，（C言語における）関数定義のような形で定義される. メソッドのプログラムを書くときのコツは，自分自身がロボットになったつもりで書くことである.

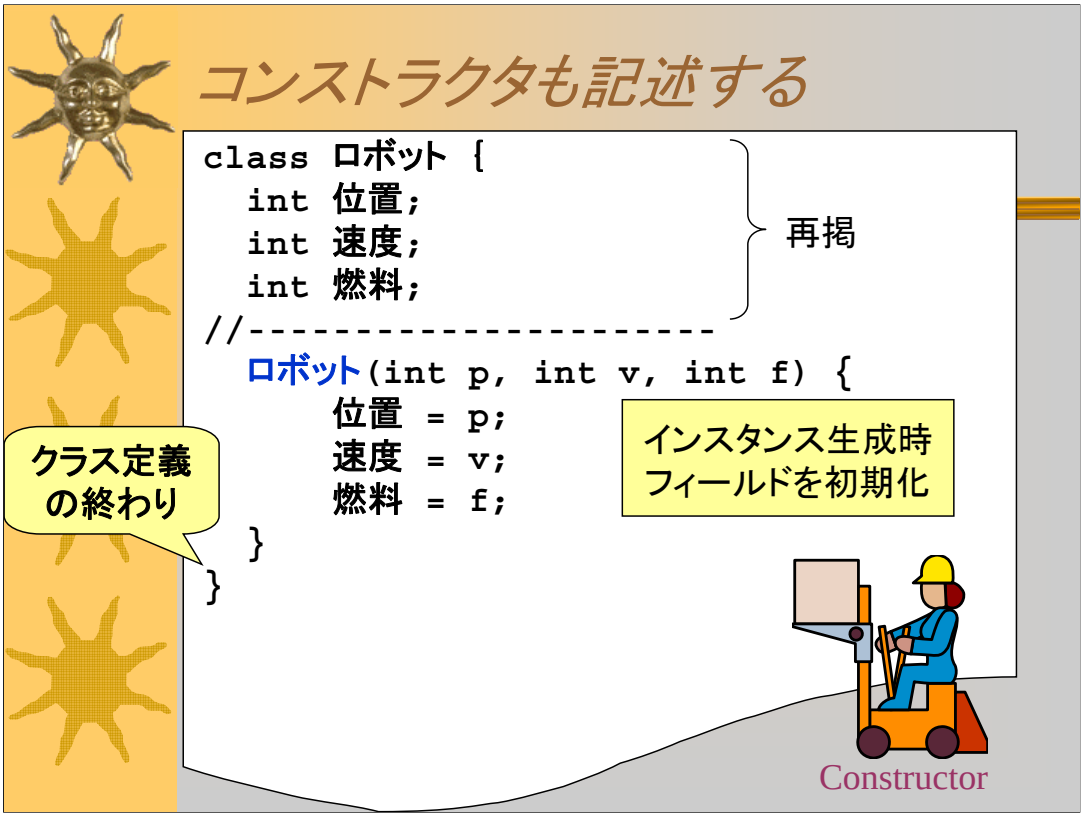
「どこ？」メソッドは，自分が「どこ？」と質問されたつもりで考えて，戻り値として，現在位置の値を返すことにする.

「変速」メソッドは，指定された新速度の値に変えよと命令されたつもりになって，自分自身の状態変数である「速度」フィールドに新速度を保存する.



「進め」メソッドは，自分が「進め」と命令されたらどうするかを考えて書く．ここでは，燃料が1単位消費され，現在設定されている速度で1単位時間だけ前進することとしよう．その結果，「位置」と「燃料」が更新される．この場合，正常終了したので，エラーコードとして戻り値0を返すような設計としてある．

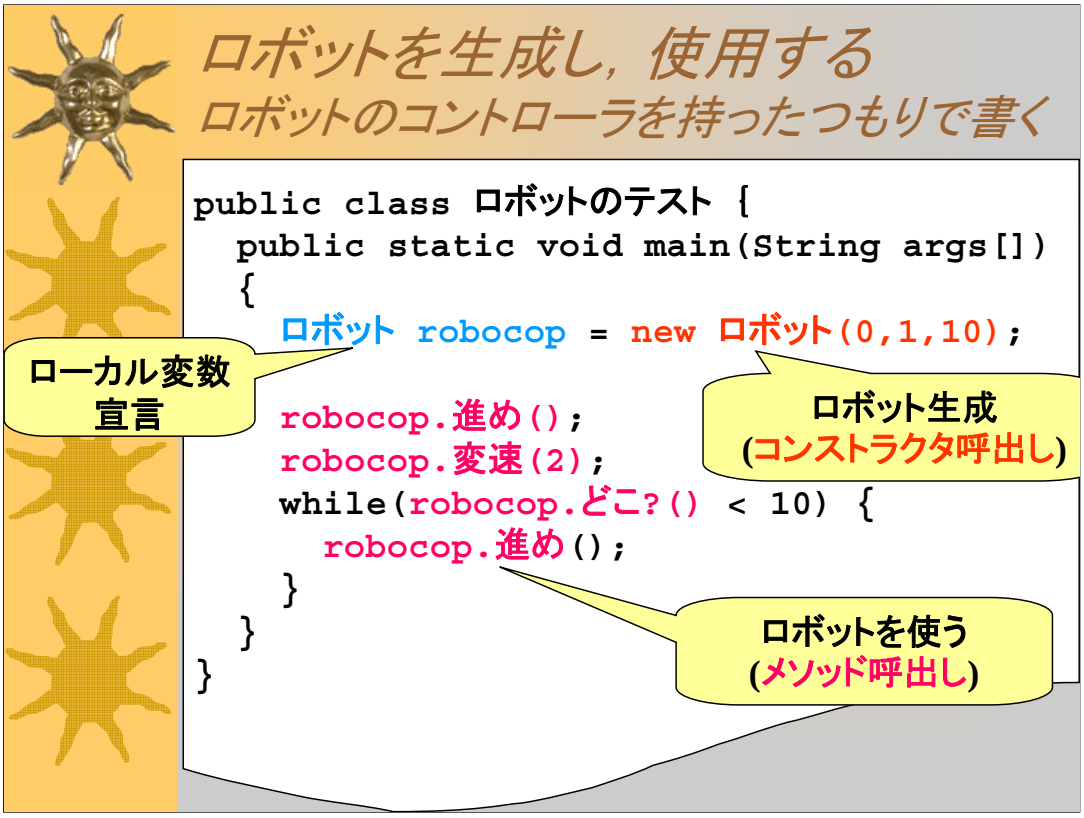
燃料が0なら，エラーコードとして－1を返すことにした．



ロボットのインスタンスが生成されたときに，そのフィールドを初期化する（などの）ために，**コンストラクタ**(constructor)という特別なメソッドを定義しておく．

Javaではコンストラクタの名前は，クラス名と同じにするという約束になっている．

この例は，「ロボット」というコンストラクタが，ロボットのインスタンス生成時にプログラマから引数として渡される p,v,t という3つの整数値を，それぞれ，「位置」，「速度」，「燃料」の3つのフィールドの現在地として記録することを表している．



これはロボットを使う簡単なプログラムである．ロボットを使うプログラムを書くときのコツは，自分がロボットのコントローラを持ったつもりで，いろいろなボタンを押しまくるように書く．ここで比喩的に言っている「ボタンを押す」とは，すでに定義した3つの「メソッドを呼び出す」ことである．

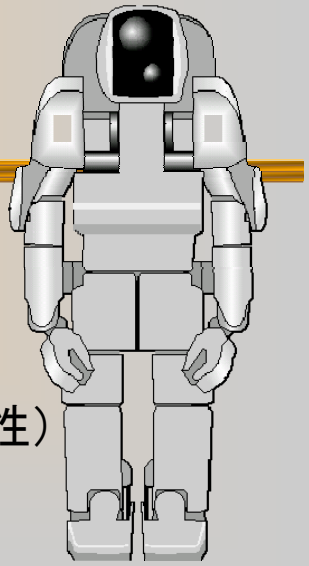
この例では，**new** ロボット(. . .)によって，ロボットのインスタンスを生成し，コンストラクタの記述にしたがって，位置＝0，速度＝1，燃料＝10に設定している．そのロボットを robocop と名付ける．専門的にいうと，これは robocop という**変数**を「ロボット型」と**宣言**し，その変数にいま生成されたロボットへのポインタを**代入**するということである．

つぎに，1単位時間だけ進ませ，速度を2に変え，位置が10以上になるまで，ループの中で進めボタンを連打する．具体的には，

robocop.**メソッド名**(**引数リスト**)

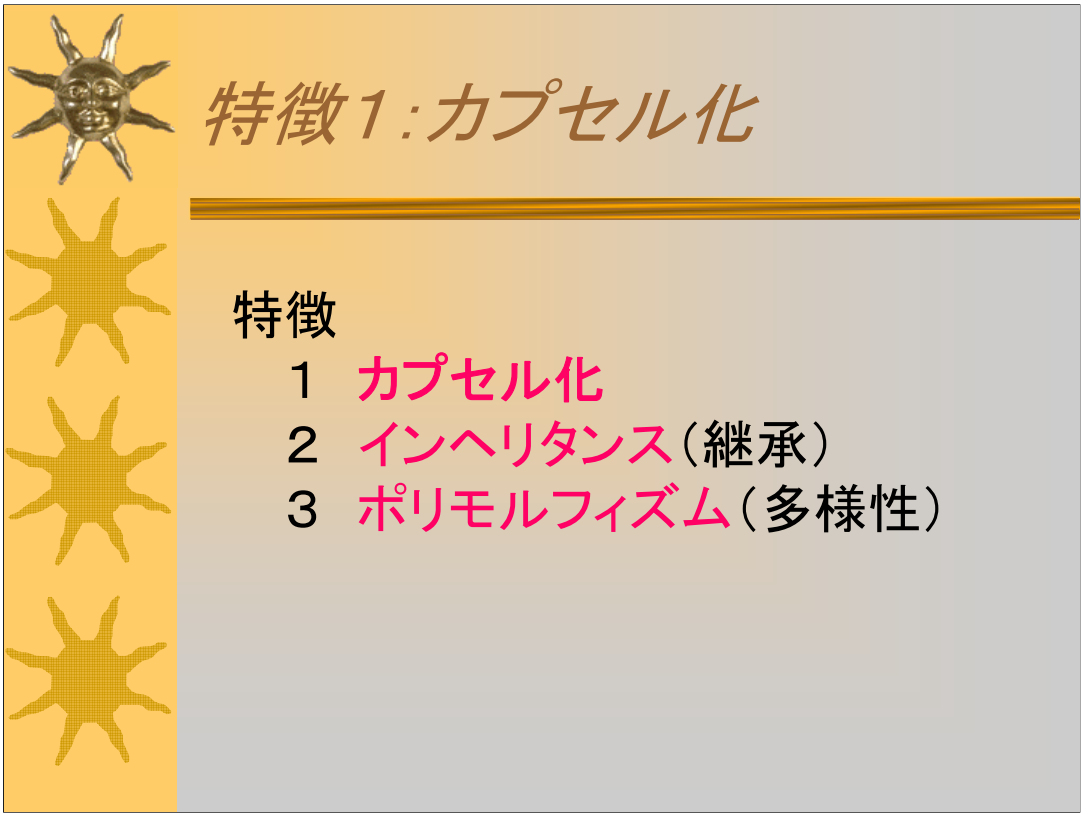
の形の式を書くことによって，メソッドを呼び出す(実行する)ことができる．

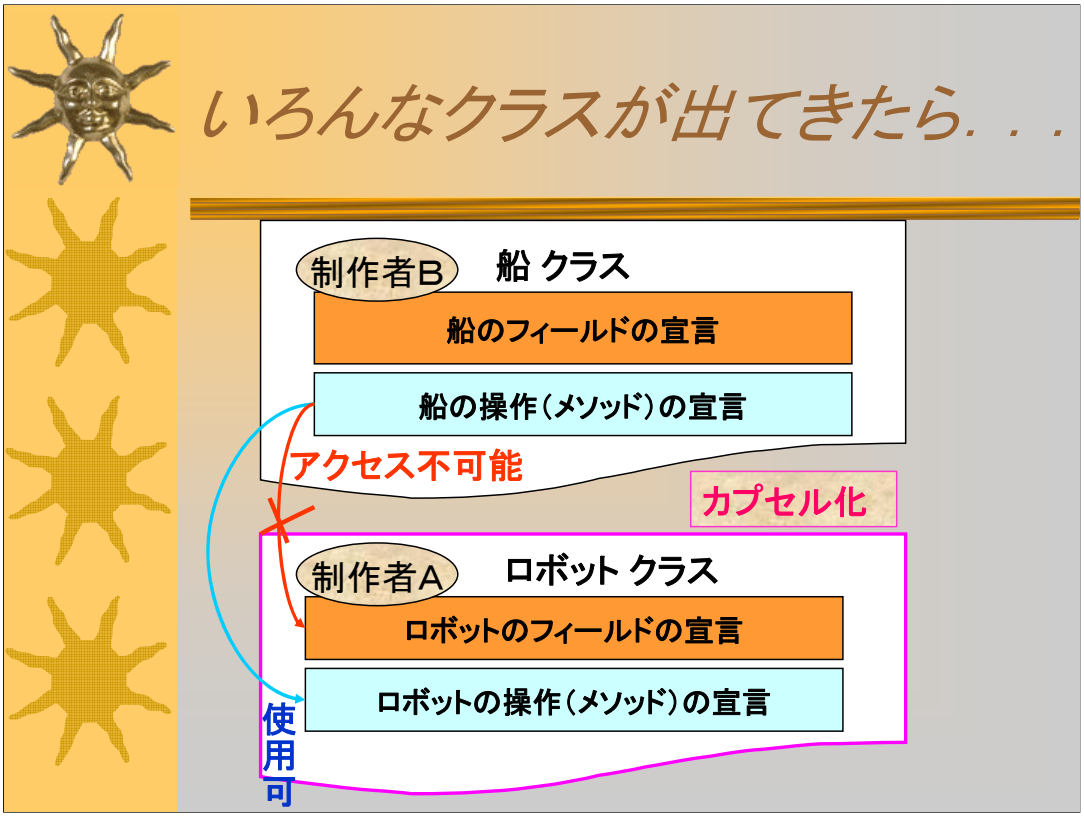
Part2
オブジェクト指向の3大特徴



特徴

1　カプセル化
2　インヘリタンス（継承）
3　ポリモルフィズム（多様性）


特徴1:カプセル化
特徴
1 カプセル化
2 インヘリタンス（継承）
3 ポリモルフィズム（多様性）



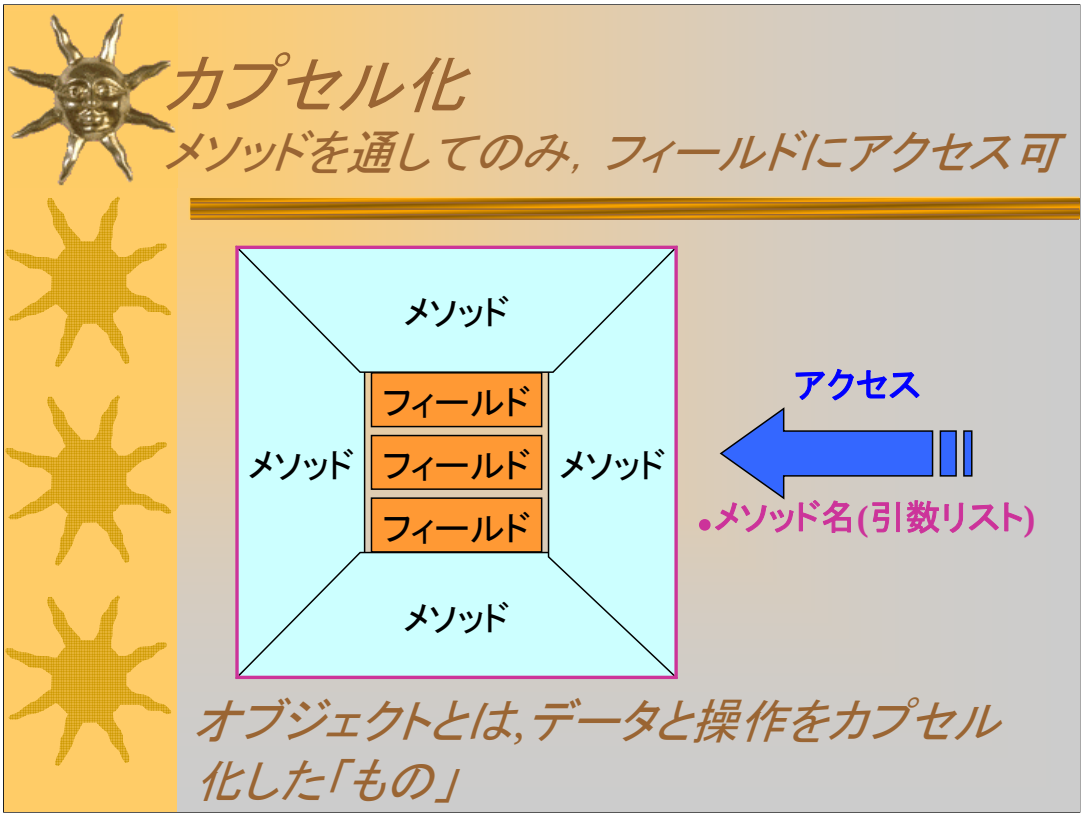
制作者Aがロボットクラスをプログラミングし，別の制作者Bが船クラスをプログラミングしている状況を考える．これらのプログラムは最終的に1つの計算機の同一メモリ内で実行されるので，この2人の制作者はじゅうぶんに連絡を密にして注意深くプログラムを作らないと思わぬミスが生じることがある．

たとえば，制作者Bが，船のメソッドの中で，ロボットのフィールドの値を勝手に変更したりするプログラムを書くと，それはロボットがロボットクラスの制作者Aの思わぬ状態変化をするということで，大変都合が悪い．

そのため，オブジェクト指向の考え方では，基本的に，1つのクラスの中のプログラムが，他のクラスのフィールドの値を直接読み取ったり変更することを禁じている．

そのかわり，メソッドの使用を通して，フィールドにアクセスできるようにする．（「読み書き」をまとめて「アクセス」という．）これにより，意図しないプログラムの動作を防止して，バグの生成を抑制したり，セキュリティを高めている．

つまり，各クラスはある種の殻によって，内部へのアクセス方法が制限されている．あるいは，保護されている．そこで，このような機能を「**カプセル化**」と呼ぶ．スライドのピンク色の部分が殻（あるいはカプセル）のつもりである．

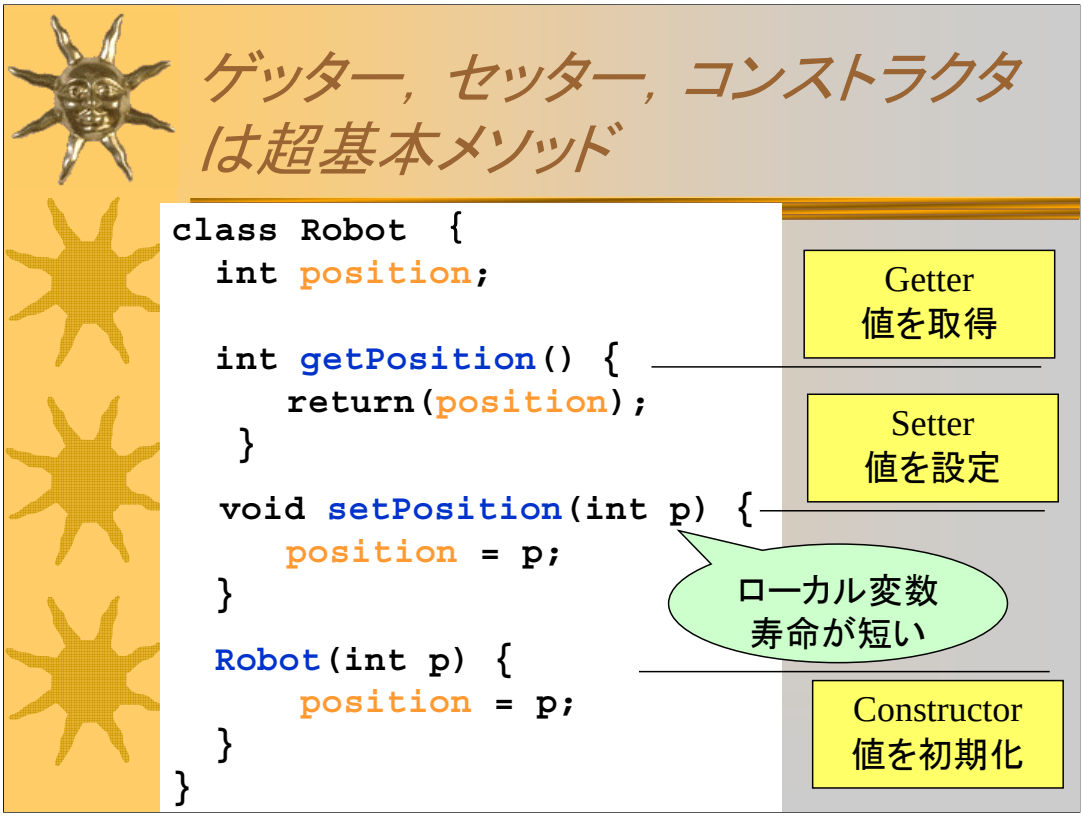


「**オブジェクトのフィールドには，メソッドを通してのみアクセスできる**」という概念を図にするとこのような感じとなる．フィールドはメソッドの殻によって守られている．

フィールドにアクセスするには，

（オブジェクト）．メソッド名（引数リスト）

の形のプログラムコードを書いて，オブジェクト内のメソッドを起動するしかない．

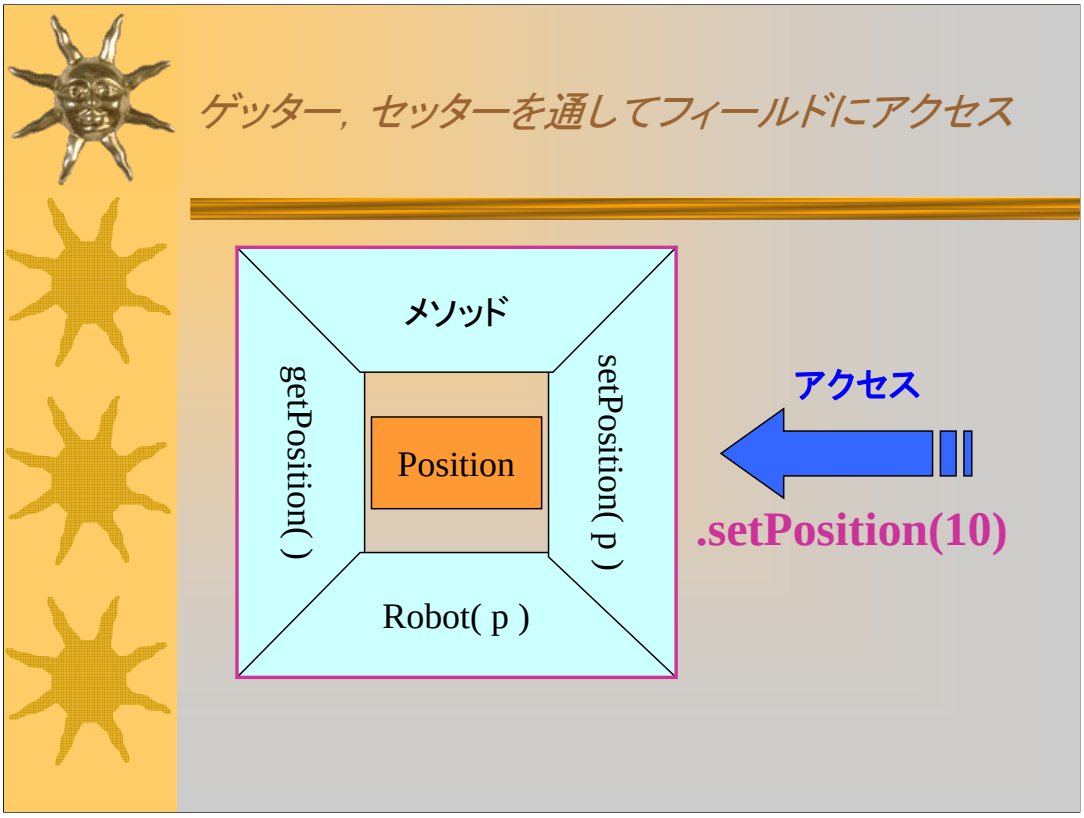


最も基本的なメソッドは，**ゲッタ**（getter），**セッタ**（setter），**コンストラクタ**（constructor）である．

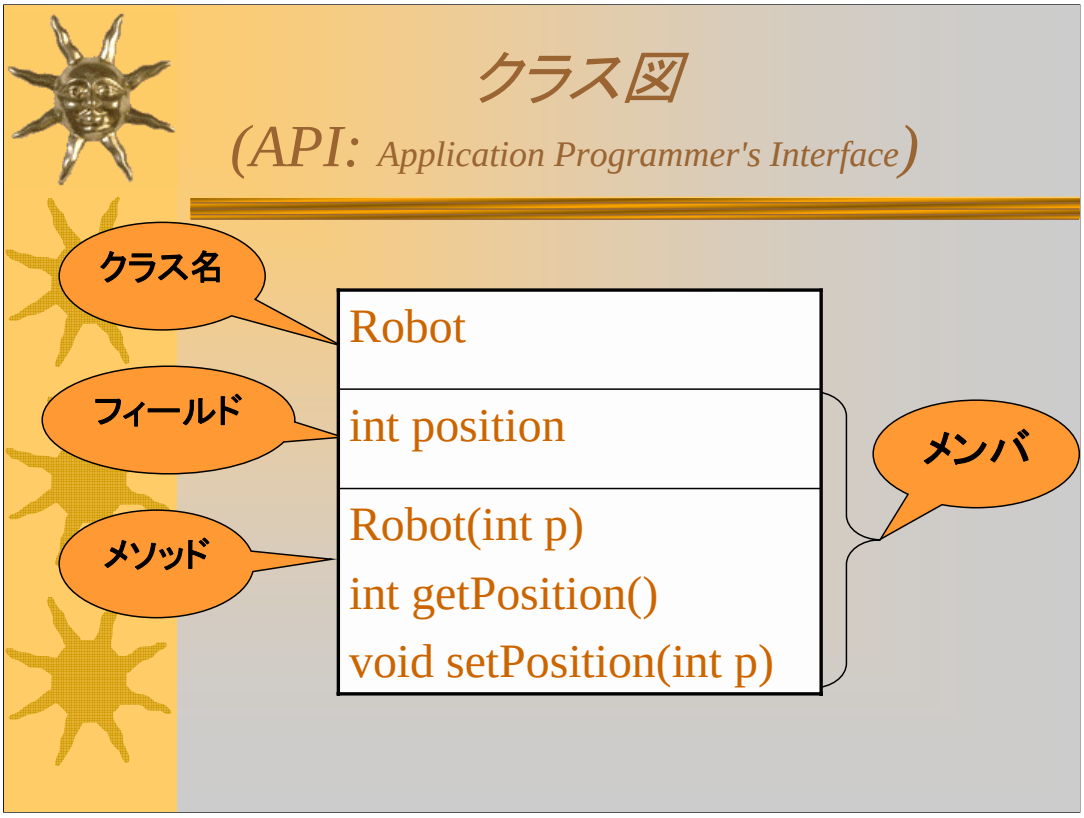
getterは，値を読み取る（ゲットする）ためのもの．getterには，慣習的に，getの後ろにフィールド名を付加した名前を付けることが多い．

setterは，値を書き込む（セットする）ためのもの．setterには，慣習的に，setの後ろにフィールド名を付加した名前を付けることが多い．

constructorは，すでに見たように，インスタンス生成時の初期設定を定義している．constructorの名前はクラス名と同じでなければならない．

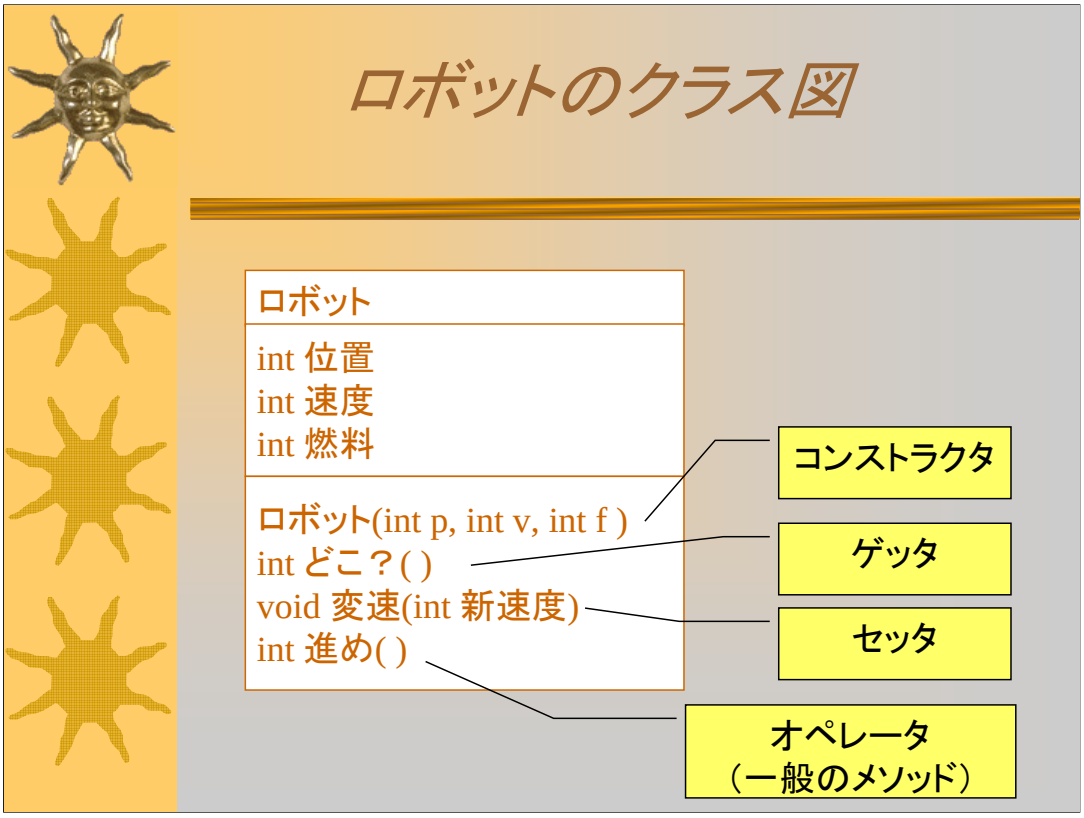


この例では，Position の値を10にセットするために，setPositionというsetterを使用している．



いつもクラスを定義したJavaのソースコードを見るのは大変なので，設計内容の概略をこのような**クラス図**として図式化することが多い．

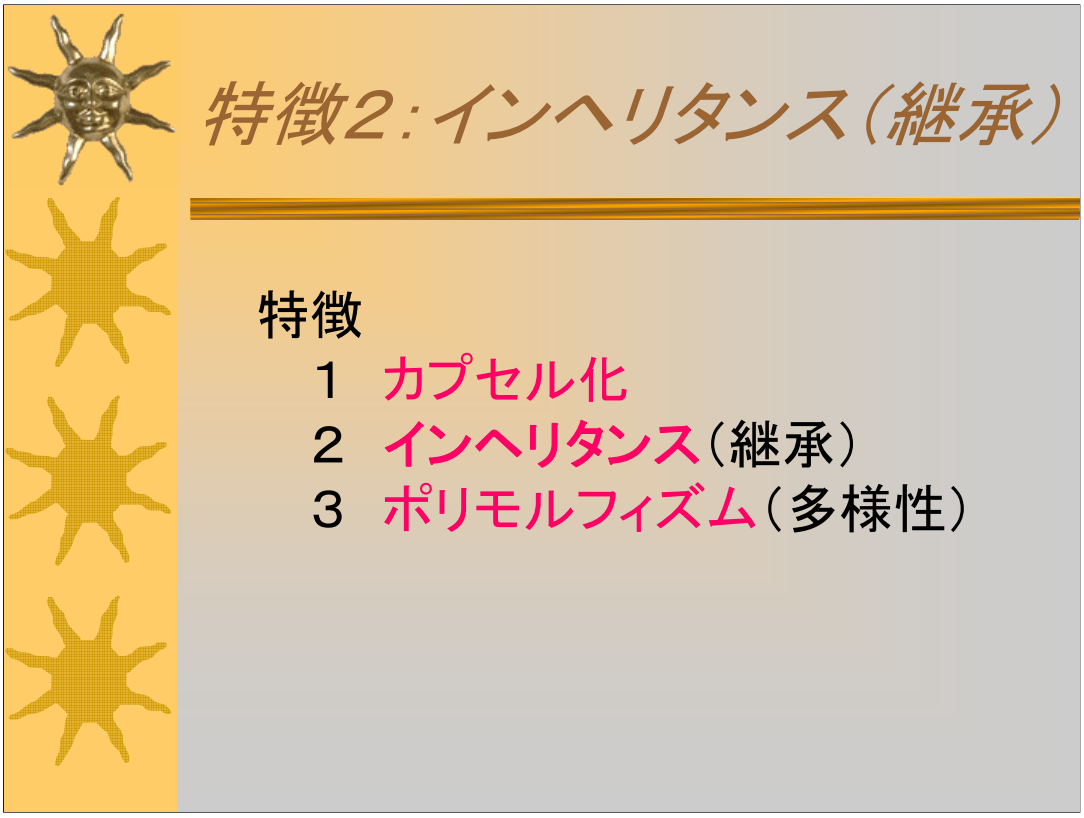
このような情報は，クラスを利用して応用プログラムを作成しようとするプログラマーに対して，利用法の「**インタフェース**」を提供しているので，**API** (Application Programmer's Interface)と呼ばれることがある．

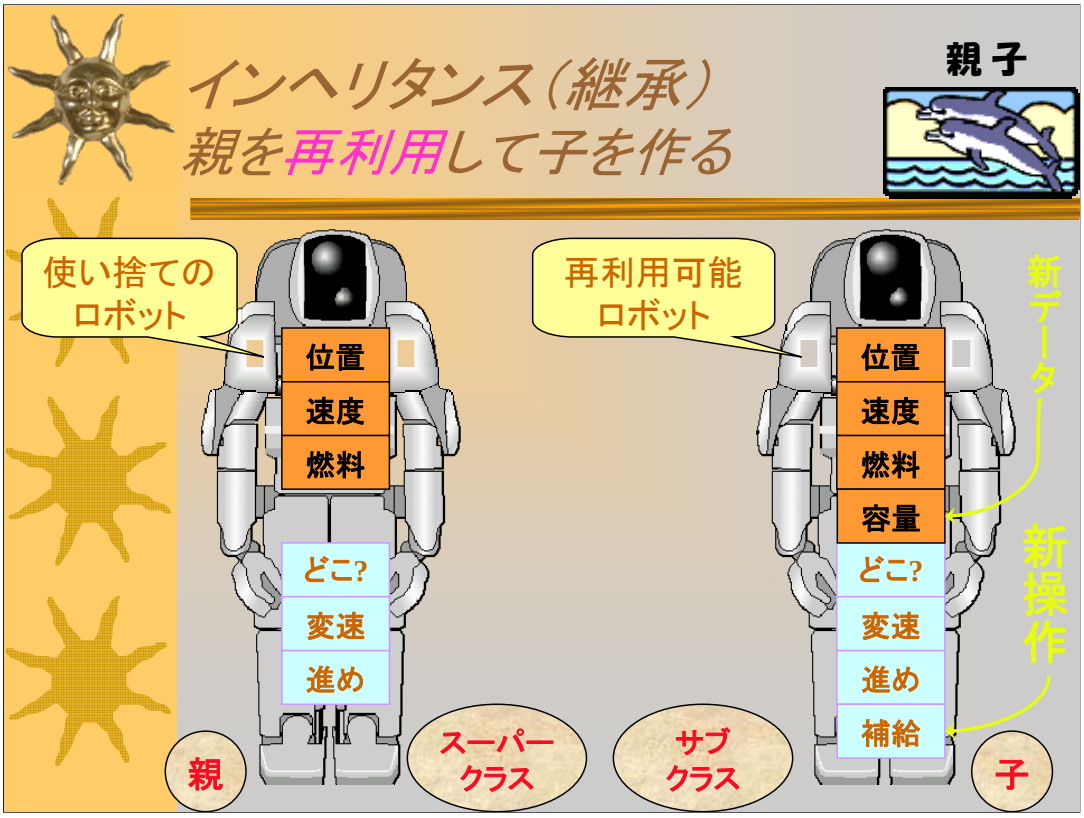


これはこれまで作ってきた「ロボット」クラスのクラス図である.

「どこ？」メソッドはgetter,「変速」メソッドは setter になっている.

「進め」メソッドは getter, setter のような基本的なメソッドではなく, このロボット特有の応用的なメソッドである.


特徴2：インヘリタンス（継承）
特徴
1　カプセル化
2　インヘリタンス（継承）
3　ポリモルフィズム（多様性）



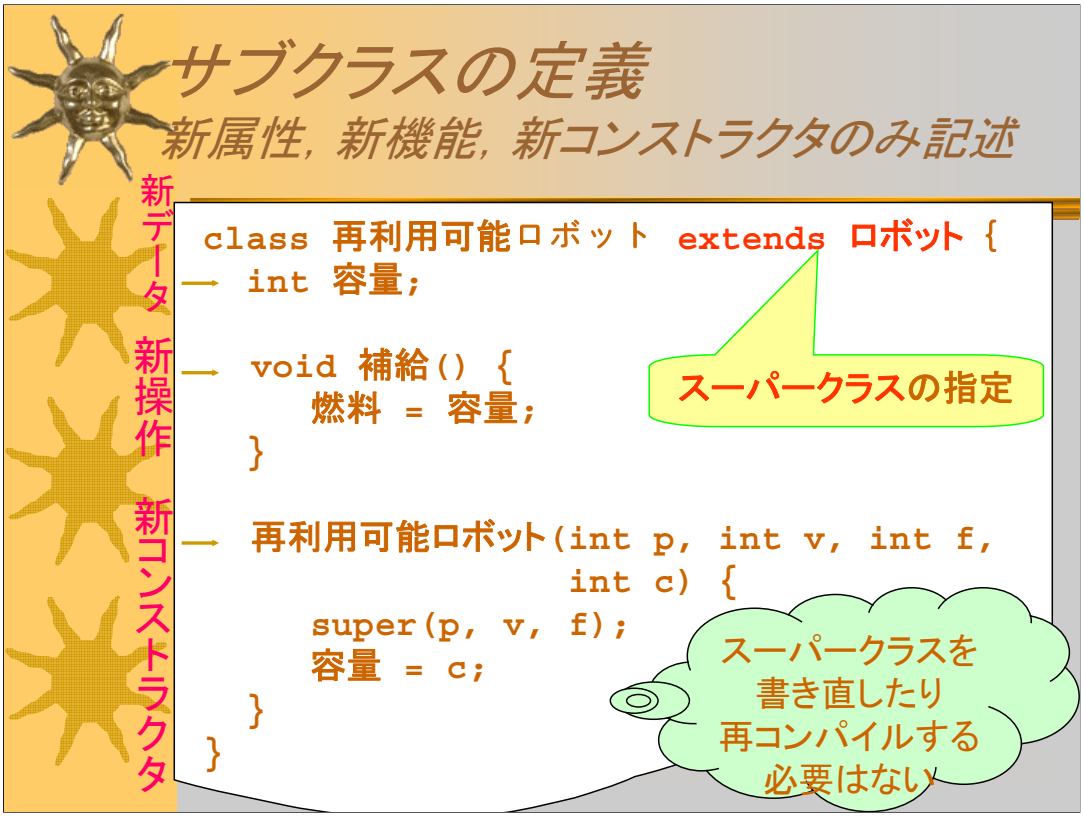
オブジェクト指向の特徴は，すでに作ったコードを再利用して機能拡張をしていく仕組みが整っていることである．それが**インヘリタンス**（**継承**）という機能である．

この例では，これまで作成したロボット（使い捨て）を拡張して，再利用可能ロボットを作ろうとしている．つまり，燃料が切れたら，燃料補給できる機能を追加する．

新しいフィールドとして燃料タンクの「容量」を追加する．

新しいフィールドとして「補給」メソッドを追加する．

このようなクラス間の関係を，図のように，親/子と呼んだり，**スーパークラス/サブクラス**と呼んだりする．

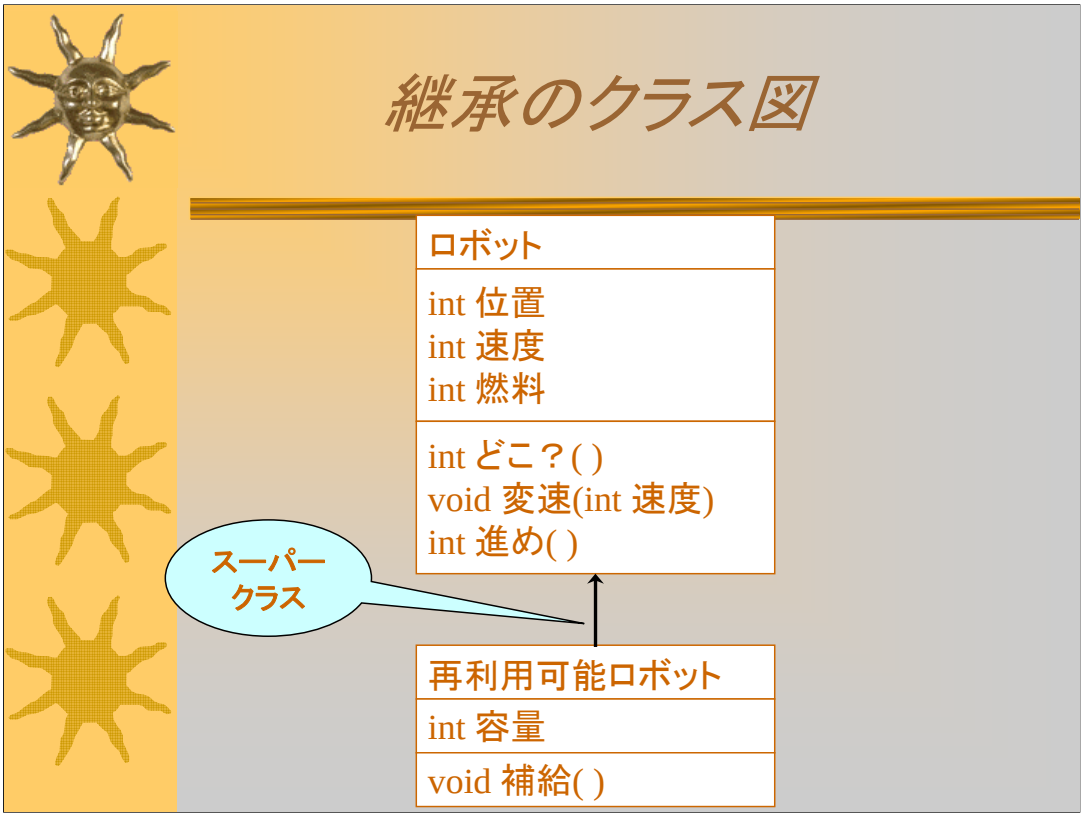


Javaではこの図のように

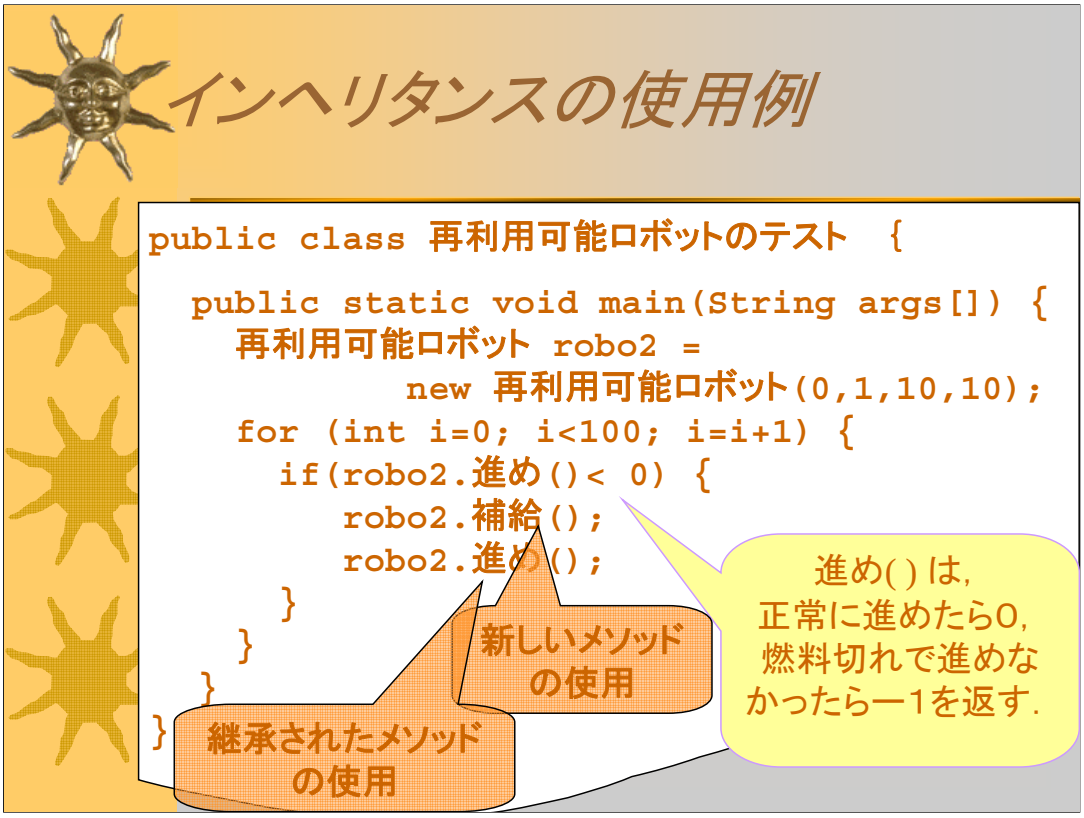
extends 親クラス名

と書くことによって，子クラスを定義できる．

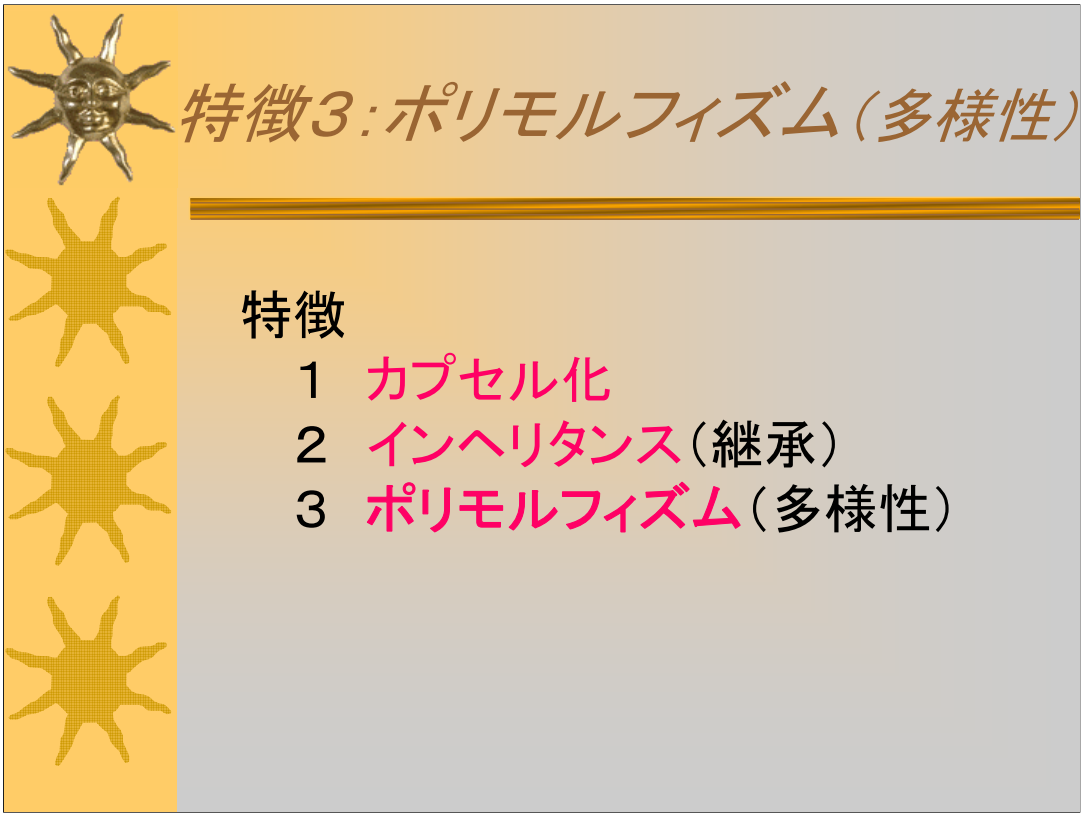
子クラスには，新しく追加するフィールドやメソッドだけを記述する．

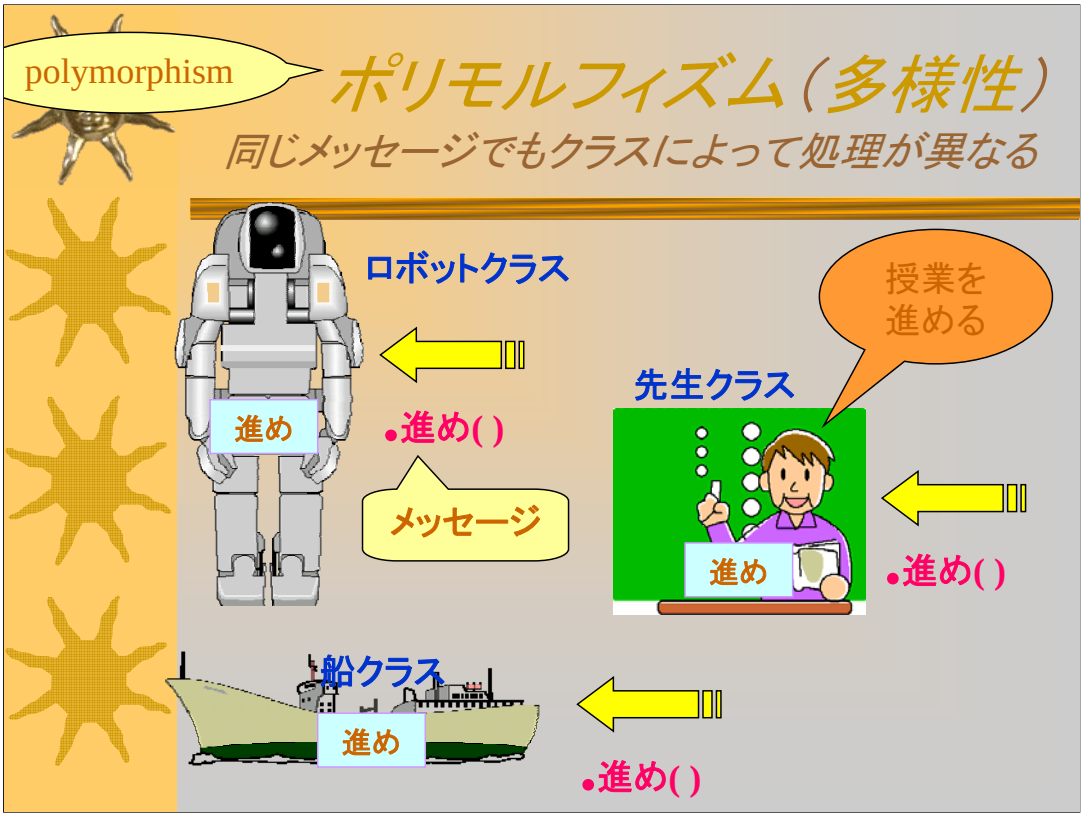


クラス図を書くときには，子クラスから親クラスに矢印を付けておく．



この例では,「進め」ボタンを100回連打する. ただし, 途中で燃料切れになったときは, 燃料を補給して, ボタンを押し直している.


特徴3：ポリモルフィズム（多様性）
特徴
1　カプセル化
2　インヘリタンス（継承）
3　ポリモルフィズム（多様性）

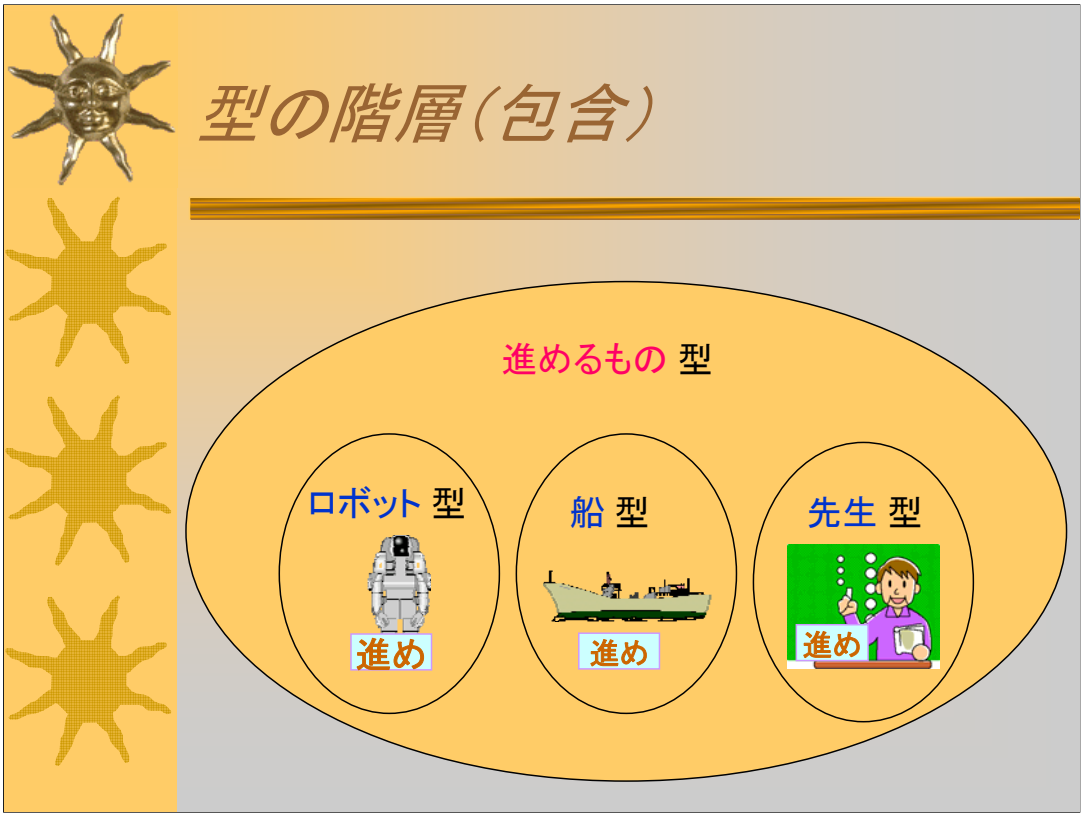


ここに図示してある3つのクラスに，いずれも「進め」メソッドがあるとしよう．3つのメソッドの内容（プログラムコード）は全く異なるものである．ロボットが「進む」とは，足を動かすプログラムコードを実行することだし，船が「進む」とは，スクリューを回転させるコードを実行することである．先生が「進む」とは，脱線していた授業を進ませることである．したがって，本来なら異なるメソッド名を付けるのがスジである．

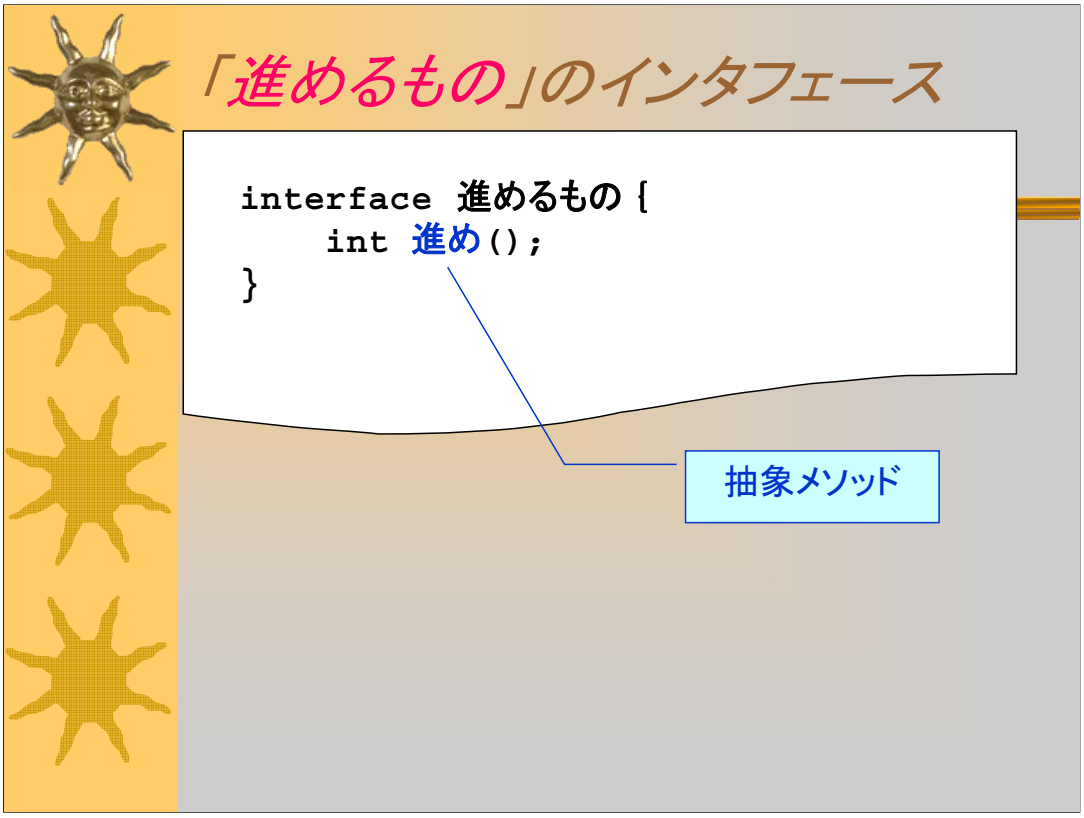
しかし，人間にとって「進め」という言葉がメソッド名として適切なら，その言葉を共通して使えるようにした方が便利で，言葉数の節約になり，人間にとって心理的に覚えやすい．

プログラミングの世界でもその考え方を採り入れたのが「**ポリモルフィズム**」という機能である．（このスライドでは「**多様性**」という和訳を採用したが，「**多態性**」と呼ばれるときもある．）

オブジェクト思考の言葉で述べると，「**同じメッセージ（メソッド）でも，受け手のオブジェクトのクラスによって，処理が異なる**」という機能である．



そのような複数のクラスをまとめて，やや抽象的な意味での「クラス」（あるいは「データ型」）にすることもできる．この例ではそれを「進めるもの」という名前のクラスにしている．

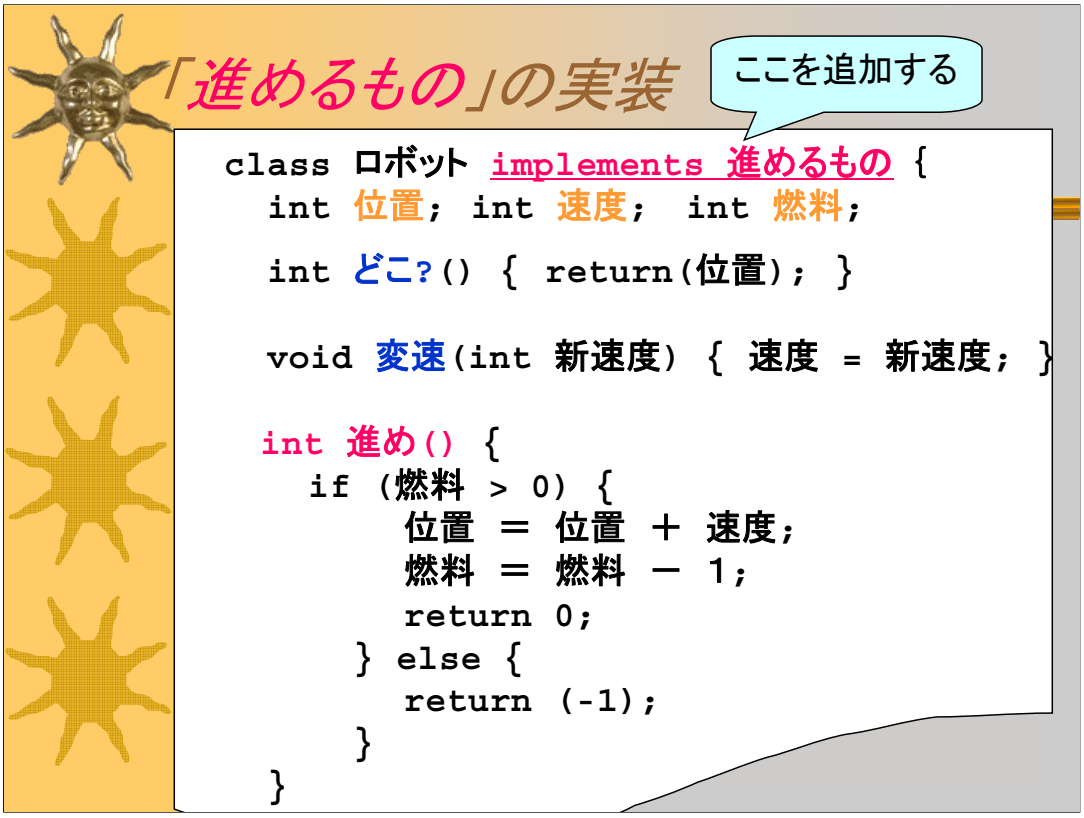


Javaでこの考え方を実現する方法の1つが,「**インタフェース**」というものである.

このJavaコードは,「進めるもの」クラスは,

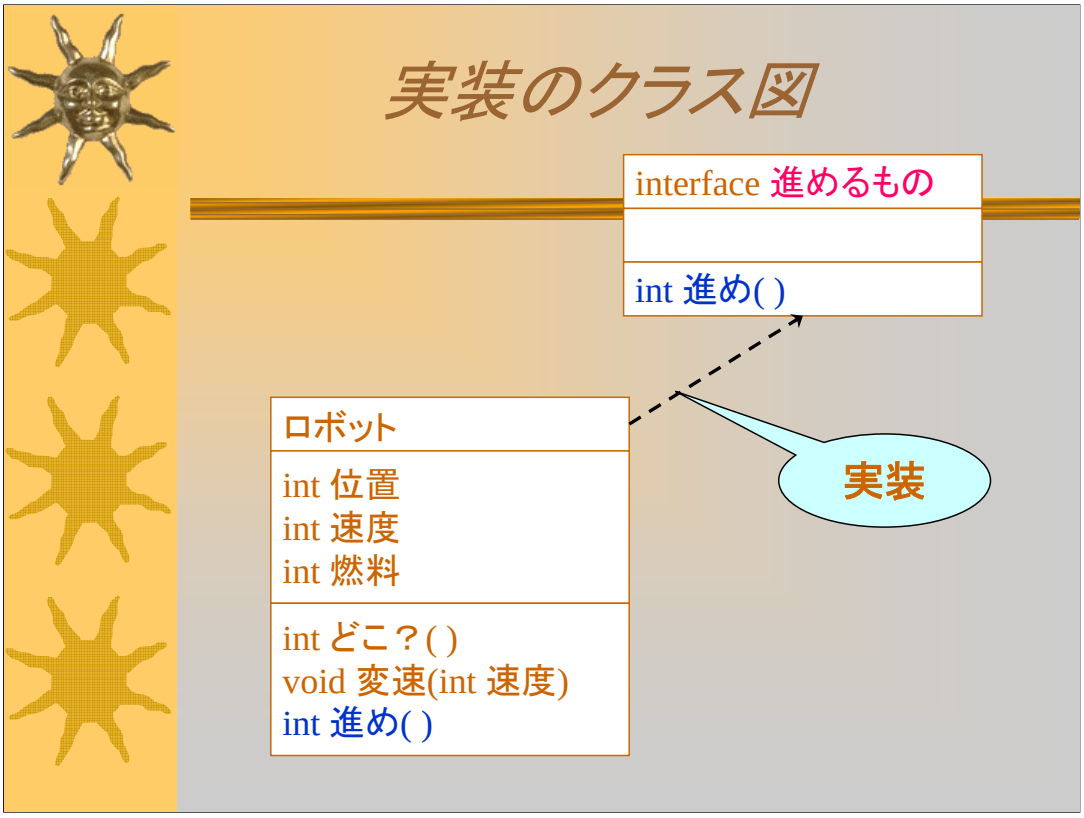
int 進め();

というメソッドを共通に持っていることを宣言している.

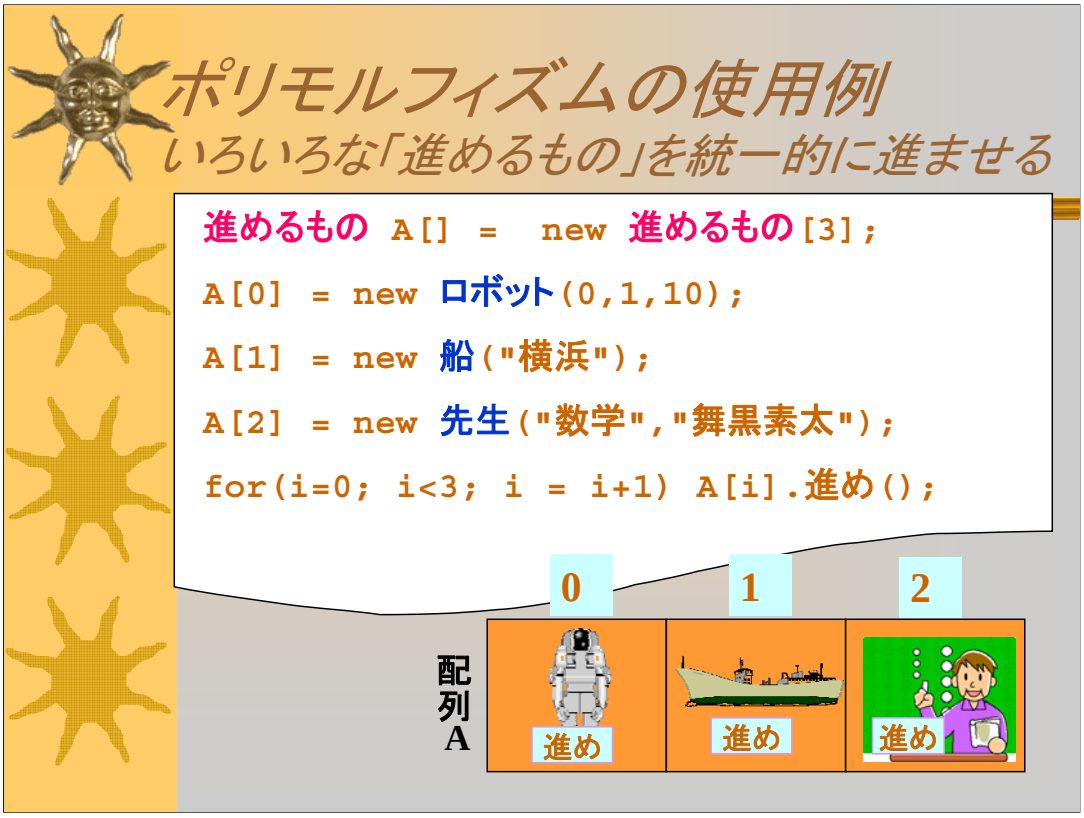


ロボットクラスが，進めるものクラスでもあることを，**implements** というキーワードを用いてこのスライドのように記述する.

このような場合，「ロボット」クラスは「進めるもの」インタフェースを「**実装している**」という.



ロボットが「進めるもの」を実装していることを，このようなクラス図で表す．



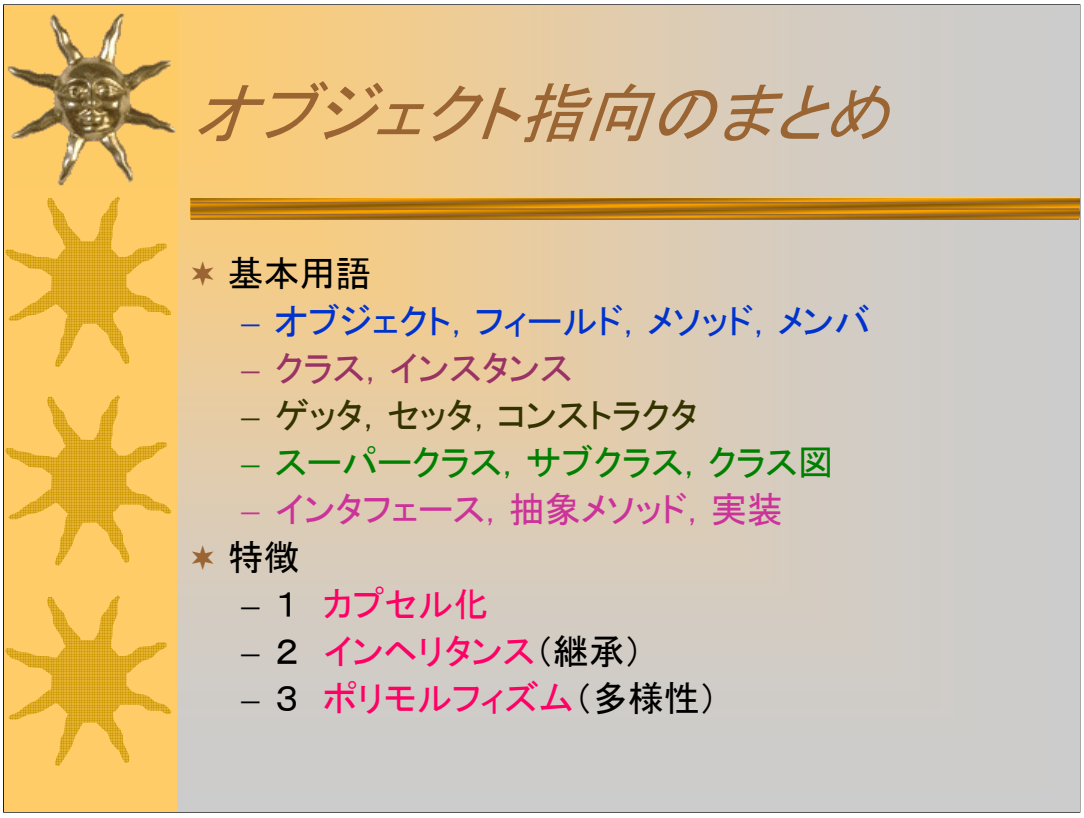
この例では，第1行目で，「進めるもの」を3つまで記憶できる配列 A を用意している．整数の配列を用意するときに，

int A[ ] = new int[3];

のように書くのと同じ理屈で，int のところは「進めるもの」というデータ型としただけである．

それぞれ，ロボット，船，先生のインスタンスを代入し，ループの中で，「進め」という同じ名前のメソッドによって，（異なる意味と異なるメカニズムのもとで）それぞれを進ませている．すなわち，ループの1周目にはロボットが足を動かして進み，2周目には船がスクリューを回転させて進み，3周目には先生が授業を進める．このような処理が，この1行で記述できるのは驚きである．

このようなプログラミングテクニックは Java のプログラムコードでは実にたくさん見かけるものであるので，ぜひ覚えておこう．



このスライドでまとめた用語のおよその意味が，それぞれ数秒で想起されるように復習してほしい．