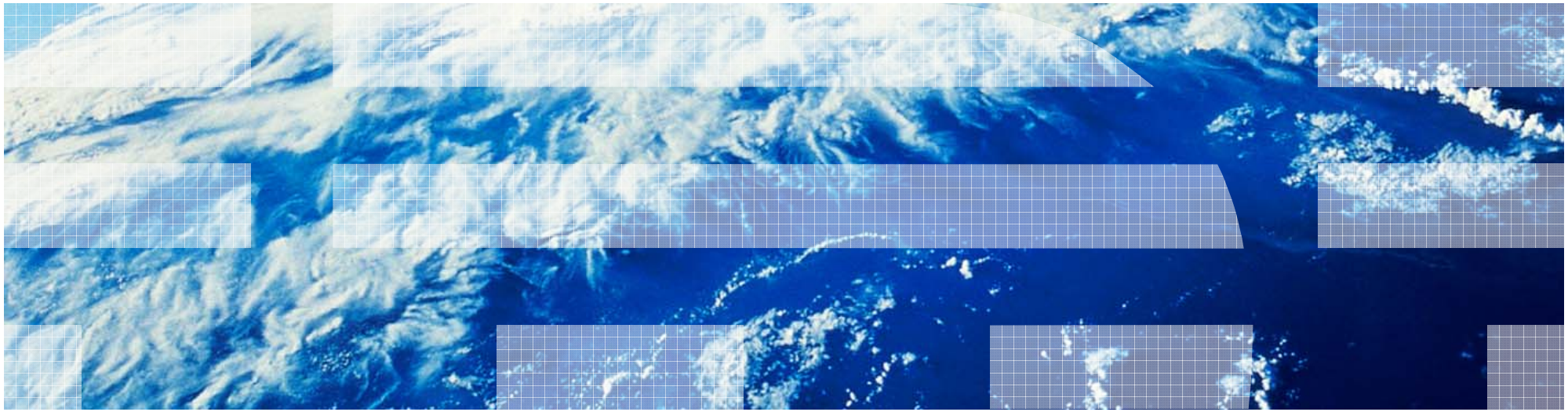
IBM

プログラミング言語 X10

河内谷 清久仁 <kawatiya@jp.ibm.com>
日本アイ･ビー･エム（株）東京基礎研究所
http://www.trl.ibm.com/people/kawatiya/



この資料はX10 2.0.6に基づいています

PPLサマースクール2010：マルチコア時代の新言語

- **タイトル: プログラミング言語X10**
- **講師**: 河内谷清久仁（日本アイ・ビー・エム(株) 東京基礎研究所）

- **概要**: X10は，IBM Researchが開発している新しい並列分散プログラミング言語で，米DARPAのHPCS（High Productivity Computing Systems）プログラムに基づくIBMのPERCS（Productive Easy-to-use Reliable Computer Systems）プロジェクトの一部である．PERCSプロジェクトは，先進的なチップ技術，アーキテクチャ，OS，コンパイラ，プログラミング言語，ツールなどを統合し，ハードウェアとソフトウェアを総合的にデザインすることで，並列アプリケーションの生産性を向上させることを目標としており，そのために，X10は新しいプログラミングモデルとEclipseに統合された開発ツール群を提供する．

  X10は型安全で近代的なオブジェクト指向言語で，マルチコアSMPチップやGPGPUなどが相互接続されたヘテロな並列分散環境に対するスケーラブルなプログラミングを可能としている．X10はいわゆるPGAS（Partitioned Global Address Space）を採用している言語族の一員であり，複数の「places」にまたがった配列などのデータ構造を扱える．さらに，オブジェクトがどのplaceに存在するかを考慮したプログラミングのための「at」や「ateach」，軽量な並列アクティビティを生成，終了するための「async」や「finish」，同期的実行のための「clock」や「atomic」などの構文を提供している．

  本講演では，X10の設計思想や位置づけに加え，これらの特徴的な構文についても使用例をまじえつつ解説を行う．なお，X10の開発はオープンソースプロジェクトとして行われており，その成果はhttp://x10-lang.org/からアクセスできる．メーリングリストなどを通じたサポートも行われている．

- **講師略歴**: 日本IBM東京基礎研究所シニア・リサーチャー．Javaを中心とするプログラミング言語の研究・開発に従事してきた．X10言語については主にJavaバックエンド部分の実装と性能向上を担当している．

カバーする内容

- このセミナーでは，並列・分散環境向けプログラミング言語「X10」について，言語仕様とプログラム例を中心に解説します
  - ➔ **目標：これを聞けば，X10で分散プログラミングができる（気分になる）！**

- 内容
  1. **X10の概要**
     - 目的，特徴，背景，歴史，開発体制，実行環境
     - プログラミングモデル，プログラム例
  2. **X10の言語仕様**
     - 基本的文法
     - データ型（class, struct, function, ...）
     - 特徴的な機構
     - 並列・分散プログラミング
  3. **X10を試してみる**
     - 環境構築，Webページ，メーリングリスト

特にここを，プログラム例をベースに解説します

1. X10の概要

X10（エックステン）とは？

HPCS: http://www.highproductivity.org/
PERCS: http://www.research.ibm.com/hptools/
Blue Waters: http://www.ncsa.illinois.edu/BlueWaters/

- **IBM Researchが開発している，新しい並列分散プログラミング言語**
  - 並列アプリケーションの生産性を向上させることが目的の一つ

- **米DARPAのHPCSプログラム**
  - High Productivity Computing Systems
  - 2002～2010年で，ペタフロップスクラスのスーパーコンピューターを開発する
  - 現在（フェーズ3）は，IBMとクレイの2社が研究・開発を継続中

DARPA: Defense Advanced Research Projects Agency 国防高等研究計画局

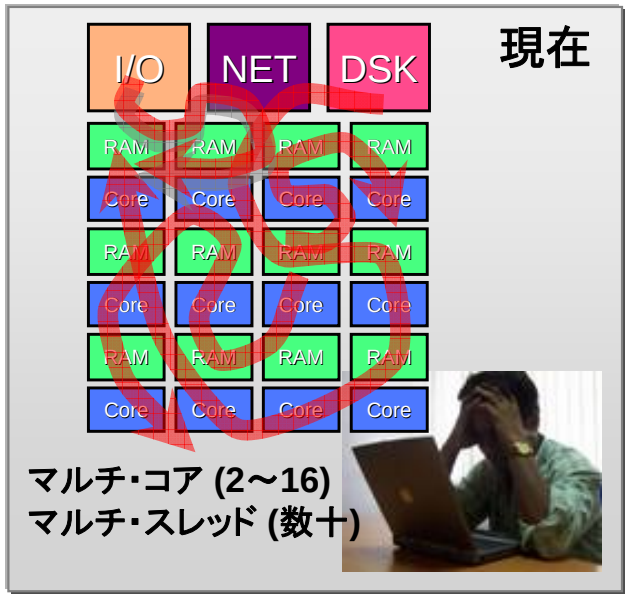
- **IBMのPERCSプロジェクト**
  - Productive Easy-to-use Reliable Computer Systems
  - ハードとソフトを総合的にデザインし，並列アプリケーションの生産性を向上させる
  - 先進的なチップ技術，アーキテクチャ，OS，コンパイラ，**プログラミング言語**，ツールなどを統合
  - PERCSシステム ＝ POWER7 ＋ AIX ＋ GPFS ＋ **X10** ＋ ・・・
  - Blue Watersシステム（PERCS技術を使用したペタスケールマシン，2011年完成）

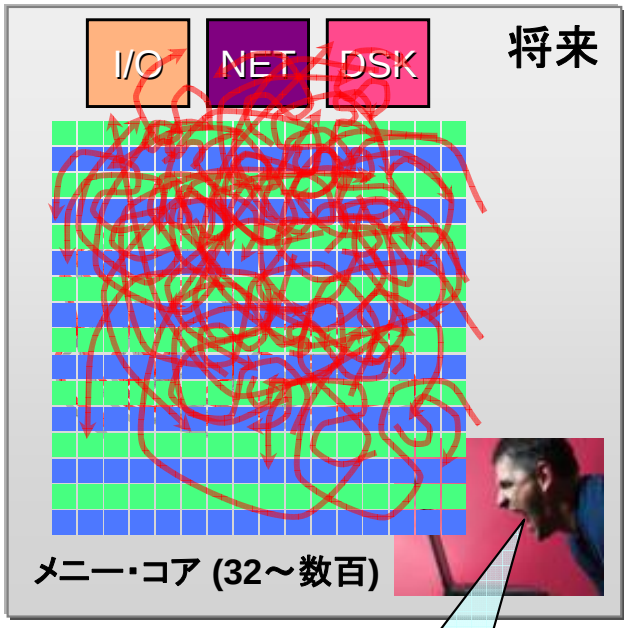
GPFS: General Parallel File System

X10の目的

Figure from “IBM Global Technology Outlook 2010”

- 並列アプリケーションの生産性を向上させる のが目的，そのために，
  - マルチコア時代のための新しいプログラミングモデル と，
  - Eclipseに統合された開発ツール群を提供





背景

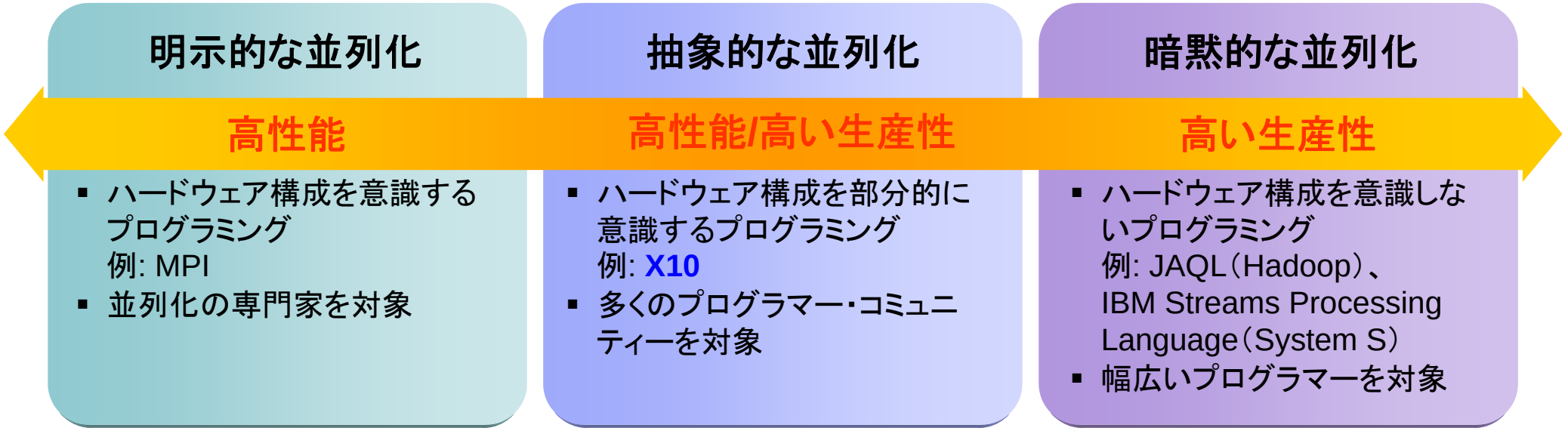
- 1スレッドの性能はもはやそう上がらない →マルチコアやハイブリッドシステムの登場
  - ソフトウェアの性能向上には，その並列性を上げることが必須
- 並列性の高いソフトウェアの開発では，生産性の高いプログラミング・モデルが重要
  - プログラマにどのように並列性を見せるか？

「こんなん，どうやって
プログラムするねん！」

X10の設計思想

Figure from “IBM Global Technology Outlook 2010”

- 並列・分散環境を隠蔽するのではなく，**抽象化してプログラマに見せる**ことで，生産性を保ちつつ高性能を達成する



➔**APGAS** (Asynchronous Partitioned Global Address Space) **モデル**

- 単一アドレス空間だが，複数の「Places」に分割されている
- 各Placeでは複数の「Activities」（≒軽量スレッド）を実行可能
- データはどれかのPlaceに属し，そこからしかアクセスできない
  - 他のPlaceからは参照しか見えない

X10：マルチコア時代の新言語

1. What is the design principle of X10?

➔並列・分散環境を隠蔽するのではなく，抽象化してプログラマに見せることで，生産性を保ちつつ高性能を達成する

- 言語の基本コンストラクトとして，並列・分散実行を指定できる
- データの分散を意識したプログラミング，デッドロックの回避

2. What does X10 identify problems on multi-core?

➔並列アプリケーション開発の生産性

- マルチコアの一般化により，並列処理の専門家だけがプログラムする時代ではなくなりつつある
- 高性能な並列プログラムを開発しやすいプログラミングモデルが必要

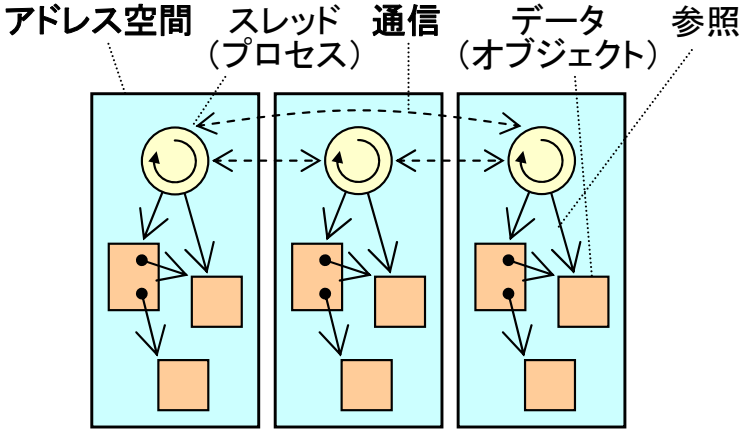
3. How does X10 solve these problems?

➔APGASモデルに基づく，並列・分散を（抽象的に）意識したプログラミング

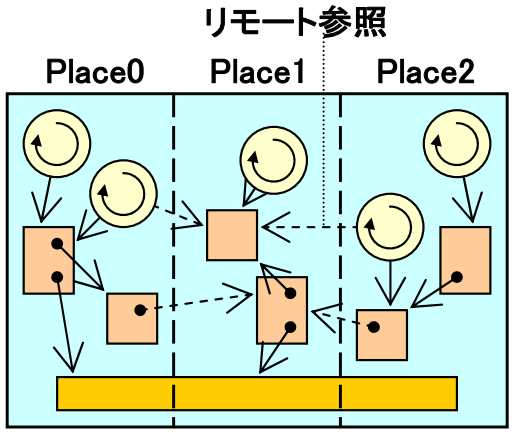
- さらに，Eclipseベースの開発ツールキットの提供

PGASモデル

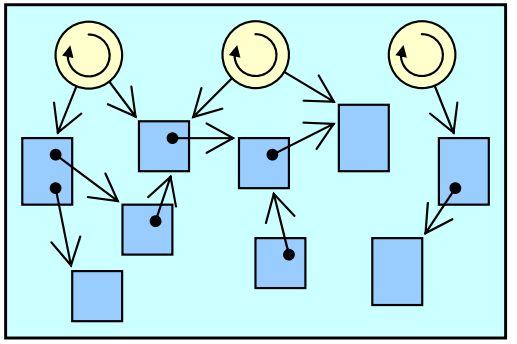
UPC: Unified Parallel C
CAF: Co-Array Fortran



Message Passing
MPIなど



PGAS
UPC, CAF, X10 など



Shared Memory
OpenMPなど

Partitioned Global Address Space (PGAS)

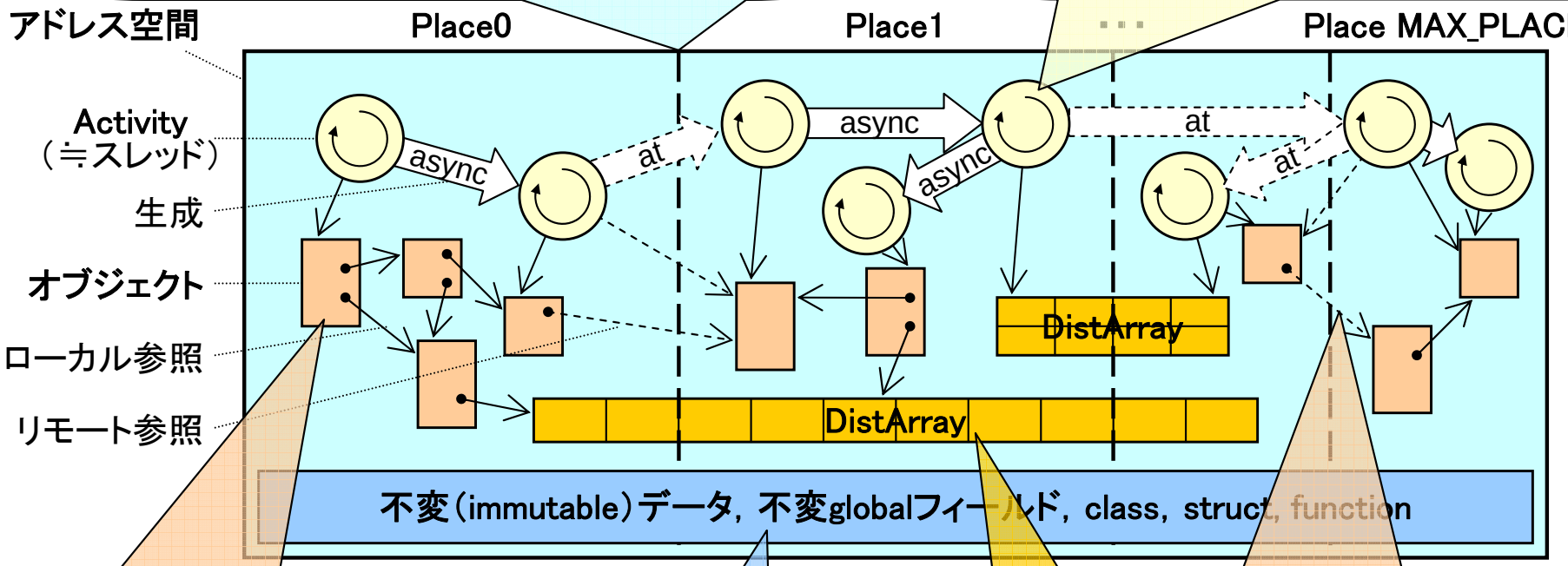
- 単一アドレス空間だが，複数の「Places」に分割されている
  - Placeが「局所性」を抽象化している，ヘテロでもかまわない（PPE，SPE，GPU，・・・）
- データはどれかのPlaceに属し，移動は起こらない
  - 複数Placeにまたがるデータ構造も作成可能（各要素が特定のPlaceに紐付けされている）
- データは同じPlaceからしかアクセスできない
  - データを他のPlaceから「参照」することは可能

- **X10 ≒ PGASの概念をJava上で実現したもの**
  - さらに，非同期実行のサポートなどを強化 → “Asynchronous PGAS (**APGAS**)”

X10の実行モデル

PGAS：アドレス空間は**複数のPlaceに静的に分割されている**（不均一でもかまわない）
- オブジェクトとActivityを保持

Activityは，Place内で逐次（sequential）動作する，非同期な実行主体（≒軽量スレッド）
- Place内に動的に生成可能（**async**文）
  - ローカルActivityとは同期・排他制御可能
- リモートPlaceにも移動可能（**at**文）
  - リモートActivityは同期不可（終了は待てる）



オブジェクトは特定のPlaceに所属
- **同じPlaceのActivityからのみアクセス可能**

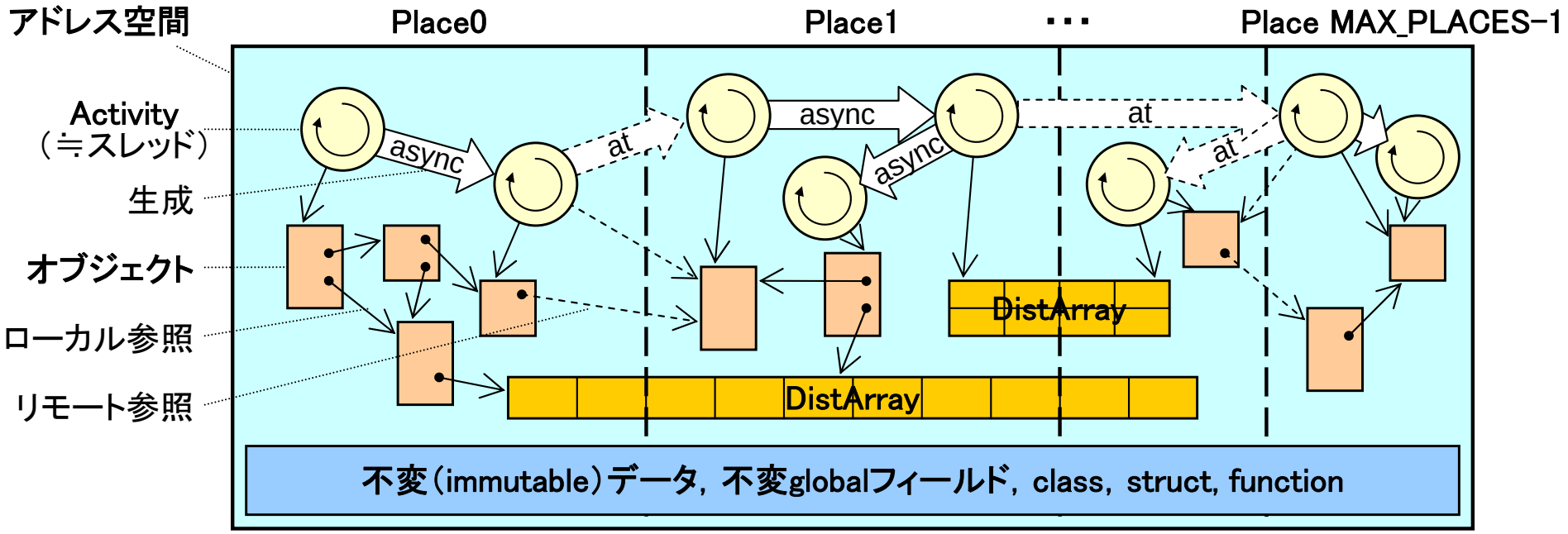
ただし，**不変**（immutable）**データ**はどのPlaceからもアクセス可能
- 暗黙的に複製される

リモートオブジェクトは「**参照**」可能
- 中身にアクセスするには，そのPlaceにActivityを生成する必要

複数Placeにまたがるデータ構造（DistArray）も作成可能
- 各要素は特定のPlaceに所属

X10の実行モデル＝APGAS

- **2つの基本アイデア ＝ Places and Asynchrony**



- X10の特徴的な構文・データ構造

細粒度並列処理
- **async** S
- **future** S

分散処理
- **at** (place) S

実行の同期
- **finish** S
- Clock

排他制御
- **atomic** S
- **when** (cond) S

分散データ構造
- Point, Region, Dist
- DistArray

Hello World in X10

- X10プログラム

```
public class HelloWholeWorld {
    public static def main(args:Rail[String]):Void {
        ateach (p in Dist.makeUnique()) {
            Console.OUT.println("Hello World from place "+here.id);
} } }
```

各PlaceにActivityを生成する文

- コンパイルと実行

```
$ x10c HelloWholeWorld.x10
$ ls
HelloWholeWorld.x10      HelloWholeWorld$1$1.class
HelloWholeWorld.java     HelloWholeWorld$Main.class
HelloWholeWorld.class    HelloWholeWorld$Main$1.class
HelloWholeWorld$1.class  HelloWholeWorld$Main$2.class

$ x10 HelloWholeWorld
Hello World from place 3
Hello World from place 0
Hello World from place 1
Hello World from place 2
```

X10プログラムはJava（またはC++）に変換され，コンパイルされる

各Placeで非同期に実行される

C++ back-endを使う場合の例
$ x10c++ -o HelloWholeWorld HelloWholeWorld.x10
$ runx10 HelloWholeWorld もしくは，
$ mpirun -n 4 HelloWholeWorld

X10 at a Glance

- **型安全で近代的なオブジェクト指向言語**
  - マルチコアSMPチップやGPGPUなどが相互接続されたヘテロな並列分散環境に対するスケーラブルなプログラミングを可能とする

- 基本的にはJava風
  - オブジェクト指向，静的な型付け
  - 単一継承（インタフェースは複数可）
  - 実行開始はmain(Rail[String])から
  - Javaに近い制御構文
    - for, while, if, switch, …
- Javaとの違い
  - 変数宣言などはScalaに近い
    - val, var, def, [T], ...
  - より広範なgenericsサポート
  - structや関数などの新しいデータ型
  - 強力な型推論，constrained type
  - そして，並列・分散処理の機能
  - ・・・

```
public class Example[T] {
    val data:T;
    def this(d:T) { data = d; }
    def get():T { return data; }

    public static def main(args:Rail[String]) {
        Console.OUT.println("Hello X10 world");

        // add from 1 to arg
        val end = (args.length > 0) ? Int.parse(args(0)) : 10;
        var sum:Int = 0;
        for (var i:Int = 1; i <= end; i++) sum += i;
        Console.OUT.printf("Sum of 1-%d: %d¥n", end, sum);

        // object creation with generics
        val o = new Example[Double](1.2);
        Console.OUT.println(o.get());

        // various data types
        val func = (i:Int, j:Int)=>i*j;
        Console.OUT.println(func(3, 4));

        // parallel/distributed execution
        finish for (p in Place.places())
            async at (p) Console.OUT.println("Place"+here.id);
    }
}
```

実行結果

```
Hello X10 world
Sum of 1-10: 55
1.2
12
Place3
Place0
Place1
Place2
```

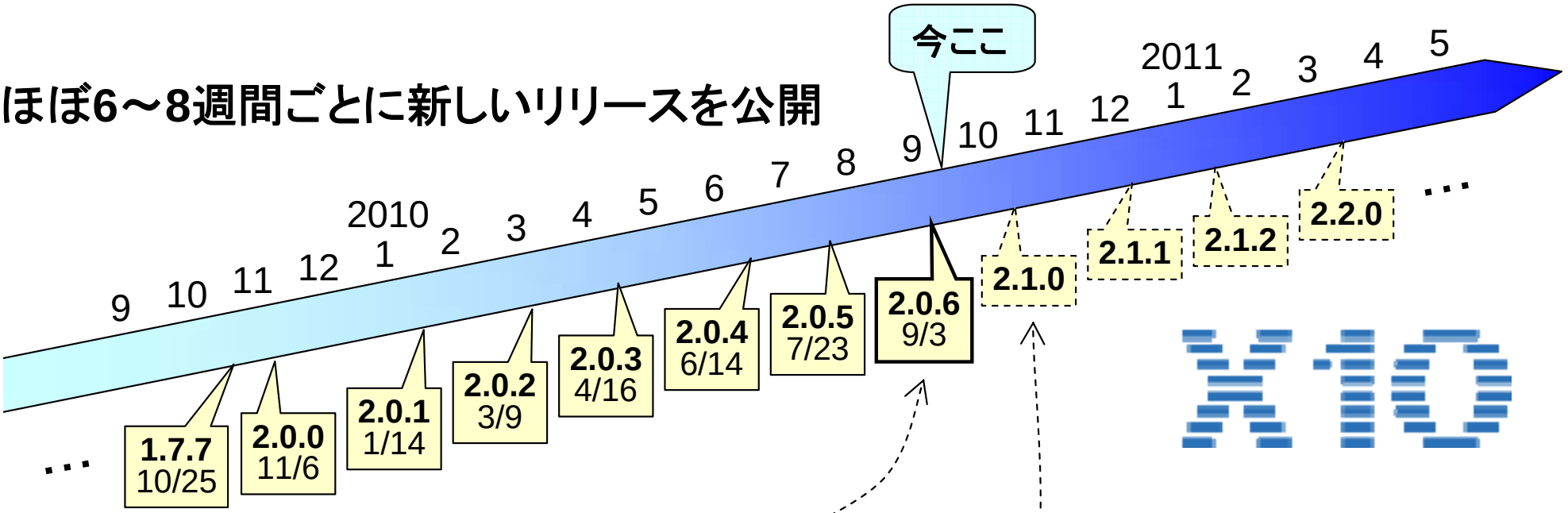
（私見では）**X10≒Javaの制御構文＋Scala風の宣言と関数＋struct＋独自の型システム・配列・スレッド・同期・分散処理機能**

X10の開発状況

Release news: http://x10-lang.org/News
Roadmap: http://x10-lang.org/Project+Roadmap

- X10プロジェクトは2004年に開始
  - ペタスケールマシン用の並列・分散プログラムをどうやって開発するか？
  - 生産性の向上，ヘテロな環境への対応

- **ほぼ6～8週間ごとに新しいリリースを公開**



  - 現時点の最新版は**2.0.6**（9/3公開）←この資料はこの版に基づく
  - 次のリリースは2.1（2010年10月予定）
    - 言語仕様は一部拡張・変更されるかも

X10の開発体制

EPL: http://www.eclipse.org/legal/epl-v10.html

- **X10はオープンソースプロジェクト**
  - プロジェクトページ：http://x10-lang.org/
    - 言語仕様のほか，チュートリアルやプログラム例など多数 →
  - ライセンスはEclipse Public License

- **ソースコード管理**
  - SourceForge (svn) https://x10.svn.sourceforge.net/svnroot/x10/
  - JIRA (bug tracking) http://jira.codehaus.org/browse/XTENLANG

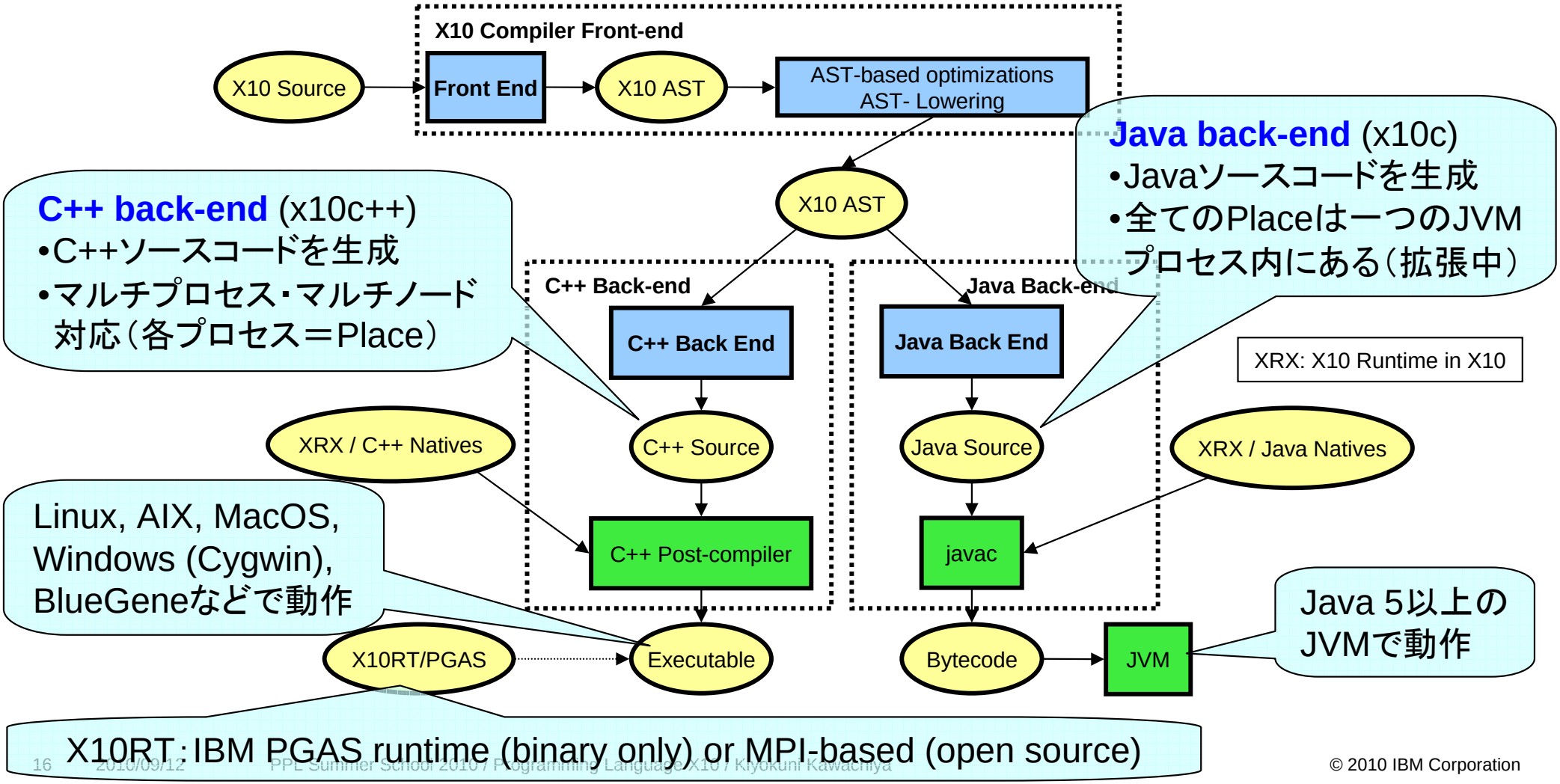
- **X10のコミュニティ**
  - メーリングリスト　http://sourceforge.net/mail/?group_id=181722
    - ユーザー向け　<x10-users@lists.sourceforge.net>
    - 開発者向け　<x10-core@lists.sourceforge.net>
  - X10 Innovation Awards (18 awards in 2010)
    - 詳細：http://x10-lang.org/X10+Innovation+Awards
  - X10 Day on 2010/04/16
    - 資料：http://x10-lang.org/X10+Day
  - Perhaps some event in SPLASH 2010 (2010/10)

Contributions are Welcome!

X10の実装

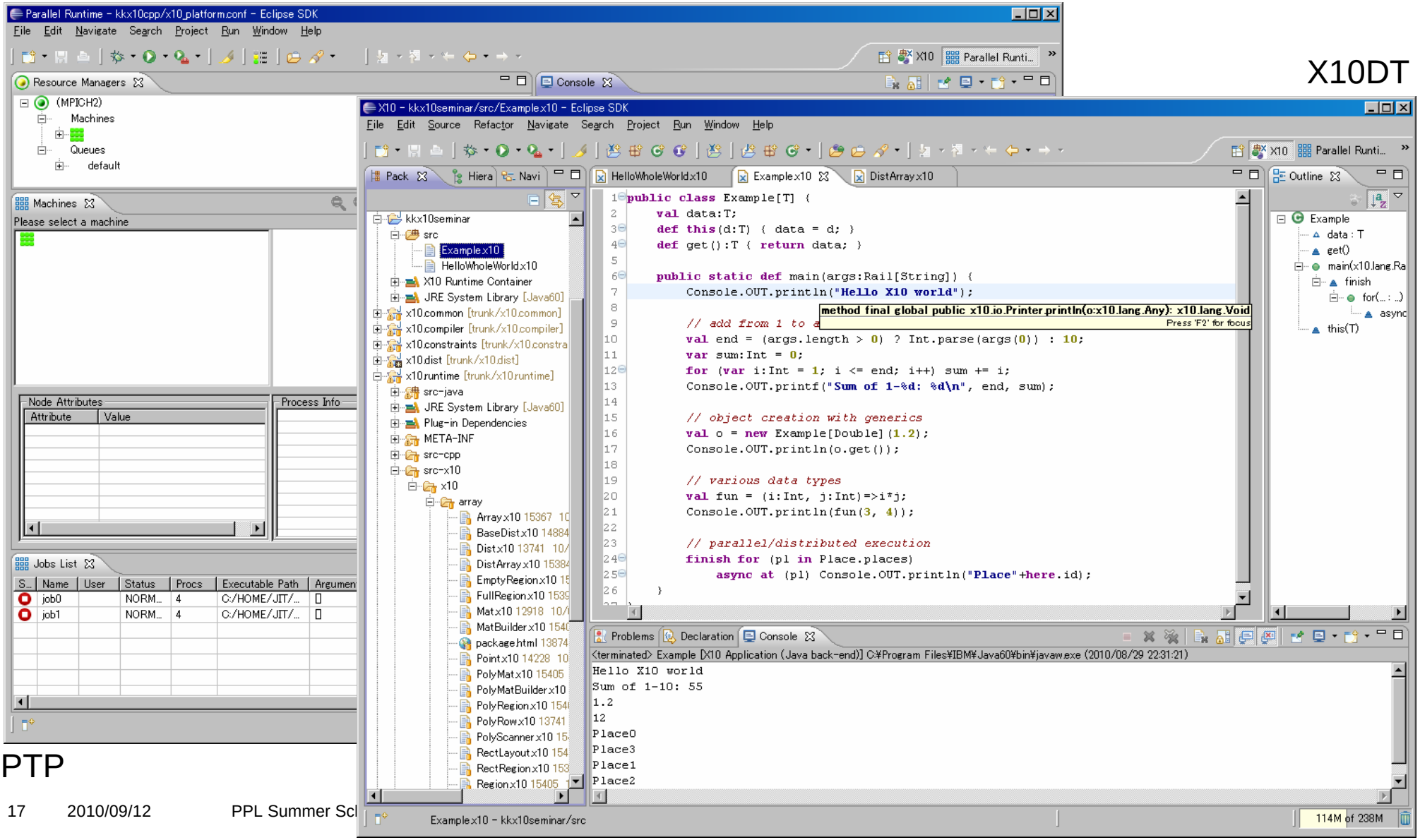
Figure from X10 tutorials

- X10コンパイラはJavaで実装されている
- X10プログラムは，C++またはJavaのソースコードとしてコンパイルされる



X10DT

- Eclipseベースの統合開発環境



2. X10 2.0.6 言語仕様

説明の前に

x10-206.pdf §0.0

- この節の内容は，**X10 2.0.6**（現時点での最新版）に基づいています
  - 言語仕様 – **x10-206.pdf**，約200ページ
    - http://dist.codehaus.org/x10/documentation/languagespec/x10-206.pdf
  - 基本ライブラリ仕様（Javadoc形式）
    - http://dist.codehaus.org/x10/xdoc/

- **使用上の注意**
  - Javaの知識があった方がわかりやすいです
  - X10 2.1以降では，一部の仕様が拡張・変更される可能性があります
    - オブジェクトのローカル化とGlobalRefの導入など
  - この節の説明には，講演者の独自解釈が一部入っています
    - 万一，仕様や実際の挙動と異なる場合はそちらが正しいです
  - 言語仕様書のどの部分に記載があるかは，右上に表示しています

- 掲載したコード例は全て，X10 2.0.6上で単体でコンパイル＆実行可能です
  → X10seminar_kawatiya_src.zipに入ってます

```
class HelloWorld {
    public static def main(args:Rail[String]) {
        Console.OUT.println("Hello World");
} }
```

X10 at a Glance（再掲）

- 基本的にはJava風
  - 実行開始は指定したクラスのmain(Rail[String])から
  - Javaに近い制御構文
    - for, while, if, switch, …
- Javaとの違い
  - val, varによる変数宣言
    - 型は「:」の後ろに書く
  - defによるメソッド宣言
  - より広範なgenericsサポート
  - structや関数などのデータ型
  - そして，並列・分散処理の機能
  - ・・・

```
public class Example[T] {
    val data:T;
    def this(d:T) { data = d; }
    def get():T { return data; }

    public static def main(args:Rail[String]) {
        Console.OUT.println("Hello X10 world");

        // add from 1 to arg
        val end = (args.length > 0) ? Int.parse(args(0)) : 10;
        var sum:Int = 0;
        for (var i:Int = 1; i <= end; i++) sum += i;
        Console.OUT.printf("Sum of 1-%d: %d¥n", end, sum);

        // object creation with generics
        val o = new Example[Double](1.2);
        Console.OUT.println(o.get());

        // various data types
        val func = (i:Int, j:Int)=>i*j;
        Console.OUT.println(func(3, 4));

        // parallel/distributed execution
        finish for (pl in Place.places)
            async at (pl) Console.OUT.println("Place"+here.id);
    }
}
```

実行結果

```
Hello X10 world
Sum of 1-10: 55
1.2
12
Place3
Place0
Place1
Place2
```

Keywords

- X10の予約語はJavaとかなり重なっているが，
  - 並列・分散実行のためのキーワード等が追加されている
  - X10では，プリミティブ型名（Int, Doubleなど）は予約語ではない
    - そういう型がstructとして定義されている（ただし実装はnative）

X10の予約語

(any)	as	async
at	ateach	atomic
await	clocked	(current)
def	(extern)	finish
foreach	future	global
(has)	here	in
next	nonblocking	
operator	or	pinned
property	proto	safe
self	sequential	
shared	(to)	type
val	(value)	var
when		

abstract	break	case
catch	class	(const)
continue	default	do
else	extends	final
finally	for	(goto)
if	implements	
import	instanceof	
interface	native	new
package	private	protected
public	return	static
super	switch	this
throw	throws	try
while		

Javaの予約語

assert	boolean
byte	char
double	enum
float	int
long	short
strictfp	synchronized
transient	void
volatile	

*括弧付きの語は未使用（と思われるもの）

Primitive Types, and Names

x10-206.pdf § 9.4, § 6

- Javaと同様の基本型に加え，符号なし整数が用意されている
  - Boolean, Char, Byte, Short, Int, Long, Float, Double, Void, String, UByte, UShort, UInt, ULong
  - 上記のうちVoidとString以外はstructとして定義されている（実装はnative）

- 基本型はJavaのように，int, boolean, void, ...とも書ける
  - これは，x10/lang/_**.x10**でそのように**type**宣言されているため

```
public static type int = Int;
public static type Int(b:Int) = Int{self==b};
public static type unsigned = UInt;
public static type int32 = int;
public static type int64 = long;

public static type Point(r: Int) = Point{self.rank==r};
public static type Region(r:Int) = Region{self.rank==r};
public static type Dist(r:Int) = Dist{self.rank==r};
public static type Array[T](r:Int) = Array[T]{self.rank==r};
public static type Rail[T](n:Int) = Rail[T]{self.length==n};

public static type Console = x10.io.Console;
                  :
```

- 名前づけのコンベンション
  - 型名：MyClass
  - 型パラメータ：T, U, ...
  - フィールド名：fieldName
  - メソッド名：methodName
  - 静的フィールド：STATIC_VAL

Literals

x10-206.pdf § 3, § 10.2, § 11.26, § 16

- X10のリテラルはJavaとほぼ同じ，主な違いは，
  - unsigned型用のリテラルがある．1.2はfloatでなくdoubleのリテラルである
  - X10独自の型のリテラル表現

true, false	Boolean型のリテラル
null	null値，オブジェクト型の変数に代入可能
123, -321, 0123（8進数), 0xBED(16進数)	Int型のリテラル
1234567890l, 0x123456789ABCL	Long型のリテラル
123u, 0123u, 0xBEAU	UInt型のリテラル
123ul, 0124567012ul, 0xDecafC0ffeefUL	ULong型のリテラル
1f, 6.023E+32f, 6.626068E-34F	Float型のリテラル
0.0, 0e100, 229792458d, 314159265e-8	Double型のリテラル
'c', '¥n', '¥t', '¥¥', '¥'", '¥101'	Char型のリテラル
"hi!", "", "Hello¥n"	String型のリテラル
	Byte型やUByte型のリテラルはない
(i:Int, j:Int):Int=>i*j	関数リテラル
[1, 3, 7]	ValRail[Int]型のリテラル，Point型にも変換可
1..10	Region型のリテラル
[1..3, 5..7]	ValRail[Region]型，Region型に変換可

Operators

x10-206.pdf § 3, § 11

- X10の演算子はJavaとほぼ同じ
  - 一部はユーザー定義も可能（後述）

```
==    !=    <     >     <=    >=    &&    ||
+     -     *     /     %     &     |     ^     <<    >>    >>>
+=    -=    *=    /=    %=    &=    |=    ^=    <<=   >>=   >>>=
=     ? :     ++    --    !     ~
=>    ->    <:    :>    @     ..
```

- X10固有のもの
  - =>　　関数リテラルやファンクション型の定義に使う
  - ->　　特定PlaceへのDistを作るのに使う
  - <: :>　クラスの包含関係を示すのに使う（Constraintなどで使用）
  - @　　アノテーションに使う
  - ..　　Regionを作るのに使う

Statements and Expressions

x10-206.pdf § 11, § 12, § 4.4

- Javaと同様の式や制御構文が使用可能
  - if, else, switch, case, default, for, while, do, break, continue, throw, try, catch, finally, return, this, new, instanceof

- Javaとの違い
  - 配列アクセスは **a(i, j) = b(i, j) + 1**
    - 内部的にはsetとapplyというメソッドが呼ばれている
  - イテレータループは
    for (v in iterableData) ～
  - キャストは **v as TypeName**
  - コレクションの要素かどうか調べる式*
    v in collectionData
  - 新しい型名を作る **type**
  - 制約の指定で自身を表す **self**
  - 並列・分散処理の機能（後述）

```
public class StatementExample {
   static type Int2DArray = Array[Int]{self.rank==2};

   public static def main(args:Rail[String]) {
      for (s in args) Console.OUT.println(s);

      val R = [1..2,3..5] as Region;
      val pt = [2,4] as Point;
      //Console.OUT.println(pt in R); // -> true
      Console.OUT.println(R.contains(pt)); // -> true

      val a = new Int2DArray(R);
      for ((i,j) in R) a(i,j)=i*j;
      Console.OUT.println(a(2,4));   // -> 8
   }
}
```

* X10 2.0.6では（なぜか）エラーになる，collectionData.contains(v)で代替可能

Comments, and Packages

x10-206.pdf § 3, § 6

- X10のコメントはJavaと同様
  - /* ～ */ か // ～ で指定する，入れ子にはできない

```
/* comment */
// comment
/* comment /* */
/*
 * multi-line comment
 */
/**
 * Javadoc comment
 */
```

- Javaと同様のpackage機構が使える
  - x10.lang.*とx10.array.*は，自動的にimportされる
  - x10.util.*や（Console以外の）x10.io.*は，プログラムが自分でimportする

X10のデータ型

- X10は静的に型付けされた言語で，3つのデータ型を持つ
  - **オブジェクト**（class），**struct**，**関数**（function）

- **オブジェクト（class）** – Javaのオブジェクトとほぼ同じ
  - クラスからnewにより生成される，参照経由でアクセスされる
  - 生成されたPlaceから移動できない（リモートPlaceから「参照」は可能）
- **struct** – メタ情報を持たないオブジェクト
  - struct型から生成される，インライン可能
  - 値は変更できない（immutable），Place間で自由にコピー可能
  - 参照渡しはできない，代入などの操作では値がコピーされる
  - structは「長いビット数のスカラー値」だと考えればよい
    - IntやDoubleなどのプリミティブ型も，structとして定義されている
- **関数（function）** – 他の言語のクロージャやラムダ式に相当
  - 一級データなので，変数に代入したり，引数として受け渡しも可能
  - 値は変更できない（immutable），Place間で自由にコピー可能

Class Types (Objects)

x10-206.pdf § 4.1.1, § 8

- X10のクラスはJavaとほぼ同じ
  - x10.lang.Objectからの単一継承（extends）による階層構造をなす
  - newで実体化，参照が作られる
  - abstract classも使用可能

Javaとの主な違い

- 型パラメータは[ ]で指定
  - 使用時は省略できない
- 静的フィールドは変更不能（immutable）なもののみ
  - 変更可能フィールドを大域的に持ちたい場合は，singletonパターンを使い，atでアクセス
- コンストラクタはクラス名ではなく，thisで宣言

```
public class MyCell[T] {
    public static val class_name = "my cell";
    var empty:Boolean;
    var contents:T;
    public def this() { emptyOut(); }
    public def this(t:T) { putIn(t); }
    public def putIn(t:T) { contents = t; empty = false; }
    public def emptyOut() { empty = true; }
    public def isEmpty() = empty;
    public def getOut():T throws Exception {
        if (empty) throw new Exception("Empty!");
        return contents;
    }

    public static def main(args:Rail[String])
                                        throws Exception {
        val c = new MyCell[Double]() ;
        c.putIn(1.2);
        Console.OUT.println(c.getOut()); // -> 1.2
    }
}
```

Structs

- X10のstruct ≒ ユーザー定義可能なプリミティブ型
  - 「長いビット数のスカラー値」だと考えればよい*
    - IntやDoubleなどのプリミティブ型も，structとして定義されている
  - 参照はとれない，常に値渡し
  - 「==」は内容が同じかどうかで判定
- 値は変更できない（immutable）
  - フィールドは暗黙にglobal val
  - Place間で自由にコピー可能
- 継承（extends）はできないが，インタフェースや関数の実装（implements）は可能
- 型パラメータも使用可能
- struct値を作るには，new無しでコンストラクタを呼ぶ

```
public class StructExample {
    static struct MyPair[T,U] {
        public val first:T, second:U;
        public def this(f:T,s:U) { first = f; second = s; }
        public global safe def toString():String {
            return "<" + first + ", " + second + ">";
        }
    }

    public static def main(args:Rail[String]) {
        val s = MyPair[Int,Double](3, 1.2);
        Console.OUT.println(s); // -> <3, 1.2>
    }
}
```

* ただし，Java back-endでは現在，ユーザー定義のstructはJavaのクラスとして実装されている

Function Types

x10-206.pdf § 4.6, § 10

- X10は関数（クロージャ）を一級データとして扱える
  - 変数に代入したり，引数として受け渡しできる
- 関数の型やリテラルは「=>」を使って記述する
  - 例：val f:(Int,Int)=>Int
    = (a:Int,b:Int):Int=>a+b;
- 関数は( )を使って呼び出せる
  - 例：上の定義で，f(2,3) → 5
- 関数は「実装（implements）」可能
  - 呼び出すと，引数の型に対応するapplyメソッドが実行される
- メソッドやオペレータは関数値として取り出せる（メソッドセレクタ）
  - 例：Math.pow.(Double,Double)
  - 例：Int.+ Int.–(Int)

```
public class FuncExample[T] {
    val func:(T)=>T;
    def this(f:(T)=>T) { func = f; }
    def doIt(a:T) = func(a);

    public static def main(args:Rail[String]) {
        val f = (d:Double)=>d*1.05;
        val e = new FuncExample[Double](f);
        Console.OUT.println(e.doIt(10));   // -> 10.5

        val sq:(Int)=>Int
            = (n:Int) => {
                var s:Int=0;
                val abs_n = n<0 ? -n : n;
                for ((i) in 1..abs_n) s += abs_n;
                s
            };
        Console.OUT.println(sq(10));       // -> 100
    }
}
```

Function Examples

- クロージャ
  - フィールド変数を束縛した関数を返す例

```
public class ClosureExample {
    var counter:Int = 0;        // environment to be captured
    def getFunc(c:Int) {
        counter = c;            // initial value
        val closure = (i:Int)=>{counter += i};
        return closure;
    }

    public static def main(args:Rail[String]) {
        val func = new ClosureExample().getFunc(1000);
        Console.OUT.println(func(1));        // -> 1001
        Console.OUT.println(func(10));       // -> 1011
    }
}
```

- 部分適用（もどき）
  - 第２引数だけを固定した関数を返す例
  - メソッドセレクタの使用例

```
public class CurryingExample {
    static def curry[T](f:(T,T)=>T, b:T) = (x:T)=>f(x,b);

    public static def main(args:Rail[String]){
        val power = Math.pow.(Double,Double);
        Console.OUT.println(power(2, 5));    // -> 32.0
        val square = curry[Double](power, 2);
        Console.OUT.println(square(5));      // -> 25.0
    }
}
```

Interface Types

x10-206.pdf § 4.1.2, § 7

- X10はJavaとほぼ同様のインタフェース型を利用可能
  - 直接の実体化はできないが，変数の型や型パラメータには指定できる
- クラスやstructは，複数のインタフェースを実装（implements）可能
- インタフェースは，複数のインタフェースを継承（extends）可能

```
public class InterfaceExample {
    interface Named { def name():String; }
    interface Mobile { def move(howFar:Int):Void; }
    interface NamedPoint extends Named, Mobile { }

    static class KimThePoint implements Named, Mobile {
        var pos:Int = 0;
        public def name() = "Kim (" + pos + ")";
        public def move(dPos:Int) { pos += dPos; }
    }

    public static def main(args:Rail[String]) {
        val m:Mobile = new KimThePoint();
        m.move(12); m.move(-3);
        Console.OUT.println((m as Named).name());
        // -> Kim (9)
    }
}
```

Interface, class, struct, function

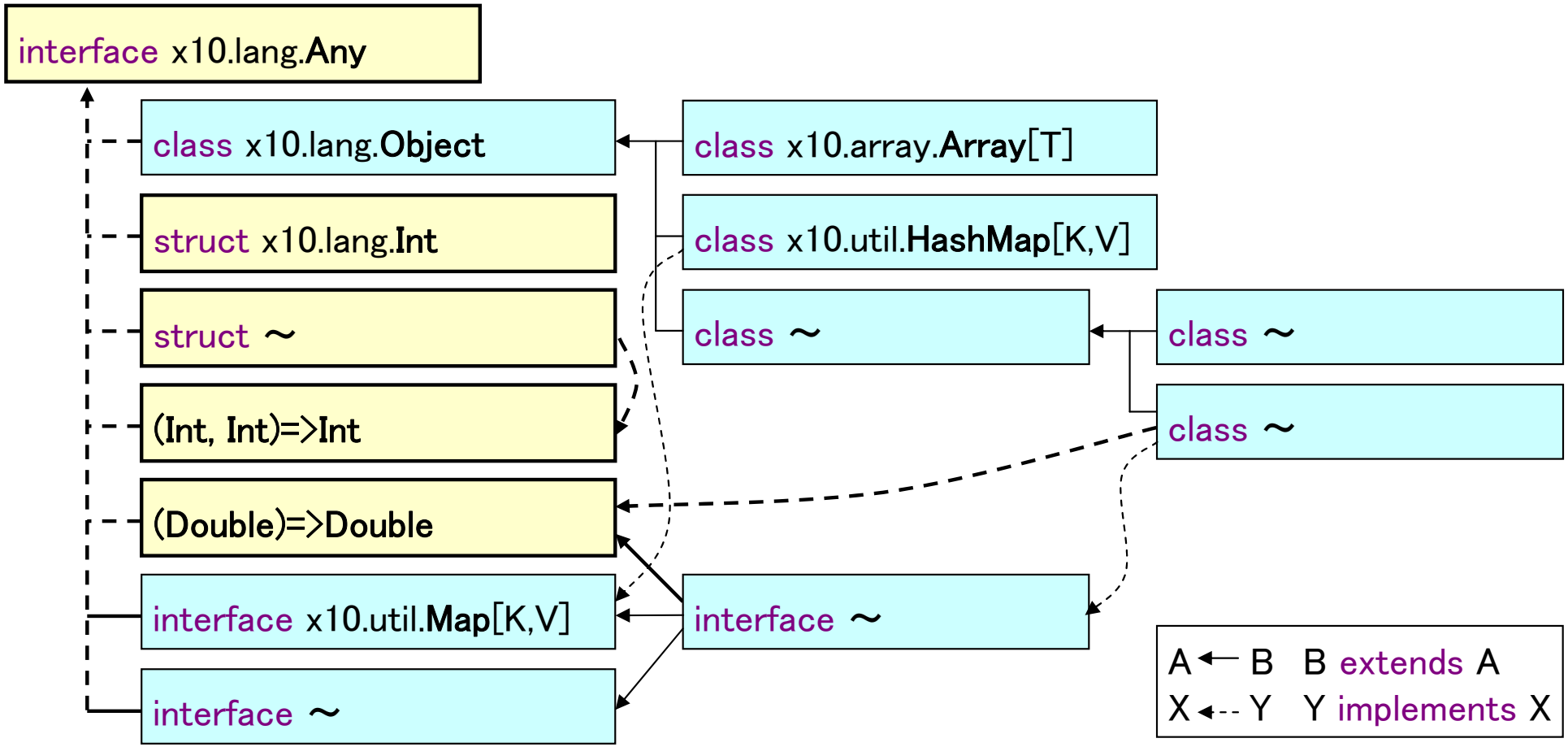
x10-206.pdf § 4.1.2, § 7, § 9.2

- インタフェースもスタティック（静的）フィールドを持てる
  - 暗黙的にglobal public static val扱いになる
  - どのPlaceからも参照可能
- structもインタフェースを実装可能
  - インタフェース型の変数に代入する時は（型が分かるように）auto-boxingされる
- 関数も「実装（implements）」可能
  - 呼び出すと，引数の型に対応するapplyメソッドが実行される

```
public class ImplementExample {
    interface KnowsPi {
        PI = 3.14159265358; // <- global public static val
    }
    static struct CircleArea implements KnowsPi, ()=>Double {
        val radius:Double;
        def this(r:Double) { radius = r; }
        public def apply() = PI * radius * radius;
    }

    public static def main(args:Rail[String]) {
        val c10 = CircleArea(10);
        val c10area = c10();
        Console.OUT.println(c10area); // -> 314.159265358
    }
}
```

Type Hierarchy

- classは階層構造をとる．単一継承，全てObjectのサブクラス
- 全ての型は，interface x10.lang.Anyのサブタイプ
- classやstructは複数のinterfaceやfunctionを実装できる



Any and Object

x10-206.pdf § 4.10

- interface **x10.lang.Any**
  - X10のすべてのデータ（オブジェクト，struct，関数）はこれを実装
  - toString()やtypeName()を宣言している

- class **x10.lang.Object**
  - すべてのオブジェクトはこれを継承（Javaと同様）
  - homeというプロパティを持つ*
    - オブジェクトが存在するPlaceを示す

```
package x10.lang;

public interface Any {
    property def home():Place;
    property def at(Place):Boolean;
    property def at(Object):Boolean;
    global safe def toString():String;
    global safe def typeName():String;
    global safe def equals(Any):Boolean;
    global safe def hashCode():Int;
}

public class Object(home:Place) implements Any {
    public native def this(); // default constructor
    public property def home() = home;
    public property def at(p:Place) = home==p;
    public property def at(r:Object) = home==r.home;
    public global safe native def toString():String;
    public global safe native final def typeName():String;
    public global safe def equals(x:Any) = this==x;
    public global safe native def hashCode():Int;
}
```


* X10 2.1では変更の可能性あり

Type Parameters

x10-206.pdf § 4.2, § 4.11.2

- class, struct, interfaceは複数の**型パラメータ**を持てる
  - 型パラメータは[ ]で指定
  - 関数の宣言は型パラメータを持てない（今のところ）
- メソッドも型パラメータを持てる
- X10のgenericsはC++のテンプレートに近い（i.e. type erasureではない）
  - インスタンスを使用する際は必ず具体的な型の指定が必要
- 指定する型はinterfaceやstructや関数でもよい
  - ただし，Voidは型ではないので指定できない

```
public class GenericsExample {
    static class Cell[T] implements ()=>T, (T)=>Void {
        var x:T;
        def this(x:T) { this.x = x; }
        def get():T = x;
        def set(x:T) = { this.x = x; }
        public def apply() = x;
        public def apply(x:T) = set(x);
    }

    static def cellGet[U](c:Cell[U]):U = c.get();

    public static def main(args:Rail[String]) {
        val c = new Cell[Double](1.2);
        Console.OUT.println(c());              // -> 1.2
        c(3.4); // invokes apply(3.4)
        val v = GenericsExample.cellGet[Double](c);
        Console.OUT.println(v);                // -> 3.4
        val d = new Cell[(Double,Double)=>Double](null);
        d.set(Math.pow.(Double,Double));
        Console.OUT.println(d.get()(5, 2));    // -> 25.0
    }
}
```

Variances

x10-206.pdf § 4.2.1, § 4.6

- Super :> Sub（SubはSuperのサブタイプ）の時，C[Super]とC[Sub]の関係は？

- class C[T] { ... } と宣言すると，invariant（不変）になる
  - C[Super] と C[Sub] に親子関係は無い
- class C[+T] { ... } と宣言すると，covariant（共変）になる
  - C[Super] :> C[Sub]　→ val cSup:C[Super] = new C[Sub](); が可能
- class C[–T] { ... } と宣言すると，contravariant（反変）になる
  - C[Sub] :> C[Super]　→ val cSub:C[Sub] = new C[Super](); が可能

関数タイプのvariance

- 返り値の型についてはcovariant，引数の型についてはcontravariant
  - 例えば，(String)=>Object :> (Object)=>Object :> (Object)=>String
    :> (String)=>String :>

C[+T] or C[–T] ?

- Covariantの例
  - Tは返り値の型として使用可能
  - GetOnly[Object] :> GetOnly[String]

x10-206.pdf § 4.2.1

```
public class GetOnly[+T] {
    val x:T;
    def this(x:T) { this.x = x; }
    def get():T = x;

    public static def main(args:Rail[String]) {
        val gStr:GetOnly[String] = new GetOnly[String]("hello");
        val gObj:GetOnly[Object] = gStr; // OK since covariant
        val obj:Object = gObj.get();
        Console.OUT.println(obj); // -> hello
    }
}
```

- Contravariantの例
  - Tは引数の型として使用可能
  - SetOnly[String] :> SetOnly[Object]

```
public class SetOnly[-T] {
    var x:T;
    def this(x:T) { this.x = x; }
    def set(x:T) = { this.x = x; }
    def summary():String = x.typeName();

    public static def main(args:Rail[String]) {
        val sObj:SetOnly[Object] = new SetOnly[Object](new Object());
        Console.OUT.println(sObj.summary()); // -> x10.lang.Object
        val sStr:SetOnly[String] = sObj; // OK since contravariant
        sStr.set("hello");
        Console.OUT.println(sStr.summary()); // -> x10.lang.String
    }
}
```

Properties

x10-206.pdf § 4.1.3, § 8.3

- class, struct, interfaceはプロパティを持てる
  - class C**(p,q)** { ... } のように指定する
  - プロパティは，global public valなインスタンスフィールド
    - どのPlaceからもアクセス可能
  - データの生成時にまとめて設定され，変更はできない
    - property文でセットする（コンストラクタごと省略も可）
  - 同名のgetterメソッドが自動的に定義される
- オブジェクトは，homeプロパティを持つ*
  - オブジェクトが存在するPlaceを示す
  - 配列などはrankプロパティも持つ（次元を表す）
- プロパティは，制約の記述に使える

```
public class Position(x:Int,y:Int) {
    def this(i:Int,j:Int) { property(i, j); }

    public static def main(args:Rail[String]) {
        val pos = new Position(10,20);
        Console.OUT.println(pos.x());       // -> 10
        at (here.next()) {
            Console.OUT.println(pos.y);     // -> 20
            Console.OUT.println(pos.home==here);
        }                                   // -> false
        var posOnX:Position{y==0};
      //posOnX = pos; // compile error
        posOnX = new Position(10,0);        // OK
    }
}
```

 * X10 2.1では変更の可能性あり

Property Methods

x10-206.pdf § 8.4.2

- property修飾子のついたメソッド
  - 制約指定の式で使える
  - プロパティにしかアクセスできない
  - コンパイル時に計算できるものでないといけない
  - 暗黙のうちに，final global

```
public class MyArray(onePlace:Place,rect:Boolean,zeroBased:Boolean) {
  //property def rail():Boolean = rect && onePlace==here && zeroBased;
    property rail:Boolean = rect && onePlace==here && zeroBased;
                                                          :
    def example() {
        val a:MyArray{rail==true}= ...
    }
```

- 引数がない場合，defと()を省略して宣言し，()なしで使用できる
  - ほとんどフィールド定義に見えるが，オブジェクト内の場所は占有しない

Constrained Types

x10-206.pdf § 4.5, § 8.9

- X10では，型やメソッドに「制約」をつけられる
- TypeName**{式}**で指定する（使える式には今のところかなり制限がある）
  - 例：String{self != null}は，nullでないStringを表す型
  - 例：42の最も厳密な型は，Int{self==42}

- 配列の次元（rank）を制限する時などに便利
- 制約は基本的にコンパイル時に（型推論により）チェックされる
  - 動的チェックも可能

```
public class ConstraintExample {
    static def addPoints(p:Point, q:Point{rank==p.rank}):Point{rank==p.rank} {
        return Point.make(p.rank, (i:Int)=>p(i)+q(i)); // returns a new Point
    }
    public static def main(args:Rail[String]) {
        val p0 = [1,2,3] as Point;      // inferred type is Point{rank==3}
        val p1 = [4,5,6] as Point;      // inferred type is Point{rank==3}
        val p2 = addPoints(p0, p1);     // OK, result is Point{rank==3}
        Console.OUT.println(p2);        // -> (5,7,9)
        val p3 = [7,8] as Point;        // inferred type is Point{rank==2}
      //val p4 = addPoints(p2, p3);     // compile error!
        val p5:Point = [9,10];          // type is Point (without constraints)
      //val p6 = addPoints(p2, p5);     // runtme error (ClassCastException)
    }
}
```

More Constraints

x10-206.pdf § 4.5, § 8.9, § 13.2

- 型パラメータに制約をつける例
  - <:はsubtypeを示すオペレータ
- Placeに制約をつける例
  - ClassName!は，ClassName{self.home==here}の省略記法*
    - ローカルなオブジェクトしか代入できないことを示す

```
public class ThrowableBox[T](ball:T){T <: Throwable} {
    def throwMe() throws T { throw ball; }

    public static def main(args:Rail[String]) {
        val t0 = new ThrowableBox[Exception](new Exception("hey")); // OK
      //val t1 = new ThrowableBox[String]("hello");                 // compile error
        try { t0.throwMe(); } catch (e:Throwable) { Console.OUT.println(e.getMessage()); } // -> hey

        val o0:Object! = new Object();
        val o1:Object  = at (here.next()) new Object();  // OK
      //val o2:Object! = o1;                             // compile error
    }
}
```

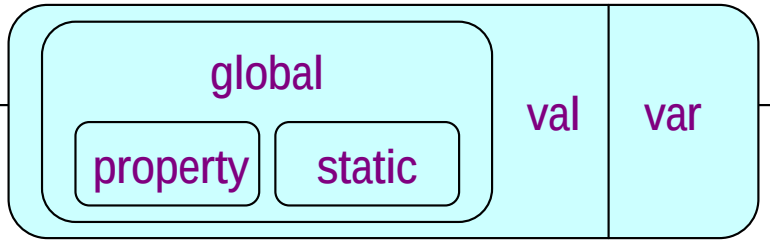
 * X10 2.1では変更の可能性あり

Type Inference

x10-206.pdf § 4.11, § 4.9

- X10コンパイラは，変数や返り値の型を推論（infer）する
- 推測できる場合は型の指定を省略可能
  - 特に，valによる変数宣言では，省略する方が推奨される
    - X10コンパイラが適切な型を制約つきで推論する
    - val宣言に型を書いておきたい場合，<:を使うとよい
- 制約は基本的に，コンパイル時に型推論によりチェックされる
  - 静的に判定できないものは，実行時チェックのコードが出る（コンパイル時に警告が出る）
  - コンパイルエラーにしたい場合，「x10c -STATIC_CALLS ～」でコンパイルする

```
public class InferenceExample {
    static def f(a:Int) = a + 1; // inferred return type is Int
    static def g(a:Int) {   // inferred return type is Any
        if (a==0) return 0; else return "non-zero";
    }

    public static def main(args:Rail[String]) { // inferred Void
        var theAnswer:Int{self==42}; // can only contain 42

        val a = 42;          // inferred type is Int{self==42}
        theAnswer = a;       // OK
        val b = 99;          // inferred type is Int{self==99}
      //theAnswer = b;       // compile error
        val c:Int = 42;
        theAnswer = c;       // dynamically checked, and OK
        val d:Int = 99;
      //theAnswer = d;       // runtime error
        val e <: Int = 42;
        theAnswer = e;       // OK
        Console.OUT.println(theAnswer); // -> 42
    }
}
```



Fields

- オブジェクトはフィールドを持てる（インスタンスフィールド）
  - 変更可能フィールド（varで宣言）と，
  - 変更不能フィールド（valで宣言）がある
- varフィールドはオブジェクトを生成したPlace（=home）からのみアクセス可能
- valフィールドを「global」と宣言すると，どのPlaceからもアクセス可能になる*
  - コピーされるので，使いすぎに注意
- propertyは自動的にglobal valになる

- structもフィールドを持てる
  - valのみ，自動的にglobal扱い
- classやinterfaceはstaticフィールドを持てる
  - valのみ，自動的にglobal扱い
  - constはstatic valの省略記法

```
public class FieldExample[T](propertyValue:T) {
  var mutableField:Int = 1;
  val immutableField = 2;
  global val globalField = 3;
  static val staticField = 4;

  public static def main(args:Rail[String]) {
    val o = new FieldExample[Int](5);
    o.mutableField = 10;
   //o.immutableField = 20; // compile error
    val r = at (here.next()) {
              var t:Int = 0;
              t += at (o.home) o.mutableField;
              t += at (o.home) o.immutableField;
              t += o.globalField;
              t += FieldExample.staticField;
              t += o.propertyValue;
              t };
    Console.OUT.println(r); // -> 24
  }
}
```


* X10 2.1では変更の可能性あり

Variables

x10-206.pdf § 5, § 10.2.1, § 14.10

- ローカル変数も，varまたはvalで宣言する
  - varは変更可能（mutable）な変数
  - valは変更不能（immutable）な値
    - 他のActivityやPlaceからも参照可能
    - その場で値を設定している場合は型指定を省略できる（推論される）
  - valもvarも付けないとvalになる
- メソッドの仮引数の宣言もほぼ同様
  - ただし型指定は省略できない
  - var宣言すると書き込めるが，参照渡しになるわけではない
    - 引数は全て値渡し（Javaと同様）
- var変数は，sharedを付けると他のActivityやクロージャからもアクセスできる（予定）
  - ただし，X10 2.0.6では未実装

```
public class VariableExample {
    public static def main(args:Rail[String]) {
        example(1, 11);
    }
    static def example(a:Int, var x:Int) {
      //a = 10;          // compile error
        x = 111;         // OK
        val b:Int;
        b = 2;           // OK
      //b = 20;          // compile error
        c:Int = 3;       // val c:Int = 3;
      //c = 30;          // compile error
        var y:Int = 22;
        y = 222;         // OK
        shared var z:Int = 33;
        finish async {
            Console.OUT.println(a+b+c); // -> 6
          //z = 333;     // will be supported
        }
        Console.OUT.println(z);
    }
}
```

Methods

- classにはメソッドを宣言できる
  - インスタンスメソッドとスタティックメソッドが利用可能（Javaと同様）
  - メソッドはdefで宣言する
  - 本体部分は「= 式」の形でも書ける
- メソッドには様々な限定子が指定可能
  - globalメソッドは他のPlaceからも呼び出し可能*，ただしglobalフィールドにしか直接アクセスできない
- 制約（Method Guard）も指定可能
  - 満たされる場合だけ呼び出せる
  - Constraint erasureとして実装
- メソッドはメソッドセレクタ記法により関数として取り出せる
- structにもメソッドを宣言可能
  - ただし，継承がないので仮想呼び出しは無い

x10-206.pdf § 8.4, § 10.3

```
public class MethodExample {
    static val staticData = "static";
    static def getStaticData() = staticData;
    global val globalData:String;
    def this(g:String) { globalData = g; }
    global def getGlobalData() = globalData;
    var mutableData:String = "";
    global def getMutableData() = at (home) mutableData;
    def setMutableData(x:Object){x!=null}
      { mutableData = x.toString(); }

    public static def main(args:Rail[String]) {
        Console.OUT.println(
                MethodExample.getStaticData());     // -> static
        val o = at (here.next()) new MethodExample("test");
        Console.OUT.println(o.getGlobalData());   // -> test
        at (o) o.setMutableData("hello");
        Console.OUT.println(o.getMutableData()); // -> hello
        val func = o.getMutableData.();
        at (o) o.setMutableData("goodbye");
        Console.OUT.println(func());              // -> goodbye
    }
}
```

 * X10 2.1では変更の可能性あり

Method Qualifiers

x10-206.pdf § 8.4.4, § 8.4.2, § 14.11.1

- メソッドには様々な限定子(qualifier)が指定可能(一部未実装)

Qualifier	意味	メソッド内の制限事項など
global*	他のPlaceからも呼べる	globalフィールドしか直接アクセスできない (atを明示的に使ってアクセスするのは可)
pinned	ローカルPlaceで処理が完結する(通信がない)	atが使えない
nonblocking	内部でブロックしない	whenが使えない
sequential	内部でActivityを生成しない	asyncが使えない
safe	atomicブロック内から呼べる	pinned nonblocking sequentlalの省略記法
atomic	Place内で不可分に実行される	safeなメソッドしか呼べない
property	制約の記述に使える	propertyフィールドしかアクセスできない コンパイル時に計算可能でないといけない

Javaの public, private, protected, package, final, abstract, native なども指定できる(はず)

 * X10 2.1では変更の可能性あり

User-Defined Operators

x10-206.pdf § 8.8, § 10.4

- +などのオペレータも
  ユーザ定義できる
- defの代わりにoperatorで宣言
  - 定義したいオペレータが
    来る所にそれを書けばよい
- staticなオペレータも定義可能
- （暗黙の）型変換も定義可能
  - これを使えば，
    operator (n:Int)+this 等を
    いちいち定義しなくて済む
- オペレータを関数として取り出すには，MyPoly.+のようにする

```
public class MyPoly(a:Int,b:Int,c:Int) { // denotes a*x^2 + b*x + c
    global public safe def toString() = a+"*x^2 + " + b+"*x + " + c;
    def apply(n:Int) = a*n*n + b*n + c;

    operator this+(p:MyPoly) = new MyPoly(a+p.a, b+p.b, c+p.c);
    static operator (p:MyPoly)*(q:MyPoly) {
        if (p.a!=0 || q.a!=0) Console.OUT.println("unsupported");
        return new MyPoly(p.b*q.b, p.b*q.c+p.c*q.b, p.c*q.c);
    }
    operator -this = new MyPoly(-a,-b,-c);
    static operator (n:Int):MyPoly = new MyPoly(0,0,n); // coercion

    public static def main(args:Rail[String]) {
        val X = new MyPoly(0,1,0);      // 0*X^2 + X + 0
        val t = -X + 3;                 // -X+3
        val u = 2*X + 1;                // 2X+1
        val v = t * u;                  // (-X+3)(2X+1)
        Console.OUT.println(v);         // -> -2*x^2 + 5*x + 3
        Console.OUT.println(v(2));      // -> 5
        val func = MyPoly.+;
        Console.OUT.println(func(t,u)); // -> 0*x^2 + 1*x + 4
} }
```

Subscripting / Indexed Assignment

x10-206.pdf § 8.8.5, § 11.7

- setとapplyにより，配列アクセス風の処理を実現できる
  - 例：a(i,j) = a(i,j) + 1; → a.**set**(a.**apply**(i,j) + 1, i, j); が実行される
  - 実際，X10の配列はこの仕組みで提供される
  - setやapplyはユーザが定義可能
    - ただし，setには必ず対応するapplyが必要（逆は不要）

- a(i,j)は，a.**apply**(i,j)の省略記法（既出）

- a(i,j)=bは，a.**set**(b,i,j)の省略記法

```
public class MyVec { // denotes a vector (v0,v1)
    var v0:Int, v1:Int;
    def this(v0:Int,v1:Int) { this.v0=v0; this.v1=v1; }
    global public safe def toString() = "(" + v0 + "," + v1 + ")";

    def apply() = toString();
    def apply(i:Int) = (i==0) ? v0 : v1;
    def set(v:Int,i:Int) { if (i==0) v0=v; else v1=v; }

    public static def main(args:Rail[String]) {
        val vec = new MyVec(10,11);
        vec(0) = vec(1) + 10; // == vec.set(vec.apply(1)+10, 0)
        Console.OUT.println(vec()); // -> (21,11)
    }
}
```

Rail and ValRail

x10-206.pdf § 11.26

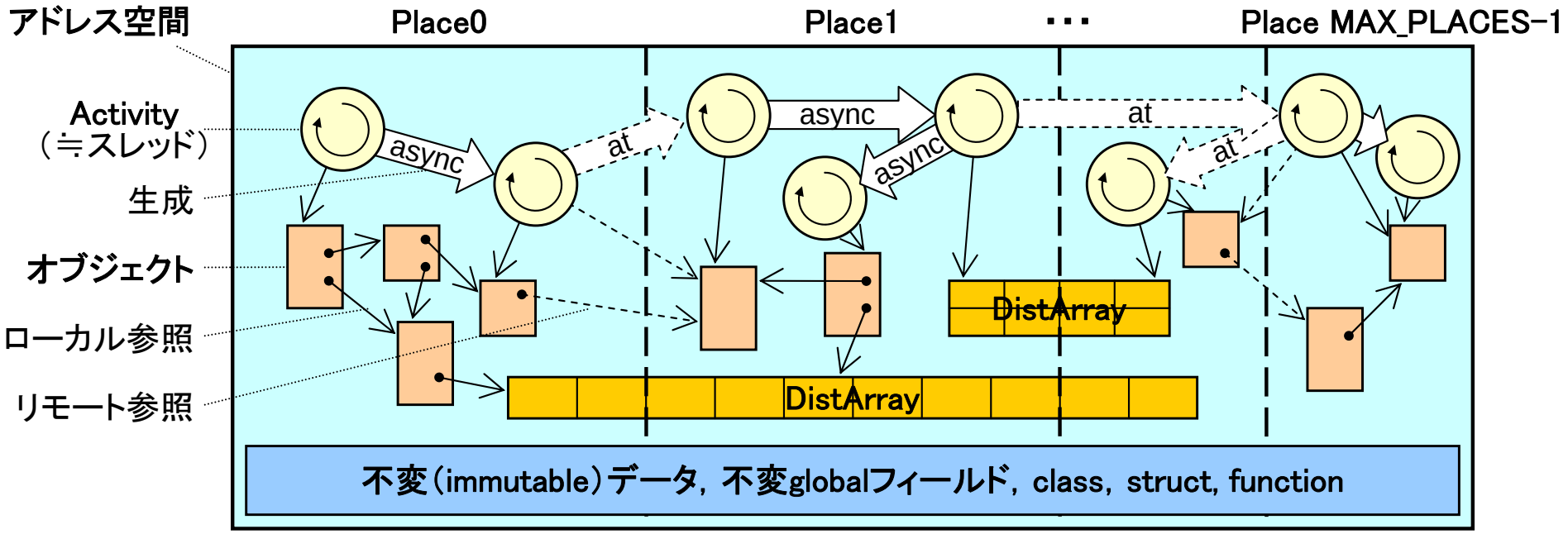
Rail: http://dist.codehaus.org/x10/xdoc/2.0.6/x10/lang/Rail.html

- x10.lang.**Rail**[T]はゼロスタートのローカル1次元配列
  - newではなく，val r = Rail.**make**[Int](length, ...);のように作る（今のところ）
    - initializer関数も指定可能
  - アクセスは**r(i)**
  - サイズはr.length
    - property，変更はできない
  - mainの引数はRail[String]!
- x10.lang.**ValRail**[+T]は要素の中身が変更できないRail
  - [1,3,7]はValRailのリテラル
    - ValRail[Int]{length==3}!型
  - ValRailはglobalなデータ，他のPlaceからも値を参照できる

```
public class RailExample {
    public static def main(args:Rail[String]!) {
        val r = Rail.make[Int](5,(i:Int)=>i);  // Rail[Int]{length==5}!
        for (var i:Int=0; i < r.length; i++) r(i)=r(i)*2;
        Console.OUT.println(r(3));             // -> 6

        val a = [1,3,7];            // ValRail[Int]{length==3}!
        for (n in a) Console.OUT.println(n); // -> 1, 3, 7

        val b = [1,a,"please"]; // ValRail[Any]{length==3}!
        at (here.next()) Console.OUT.println(b(2)); // -> please
    }
}
```

X10による並列・分散処理（再掲）

- 複数の**Places**で複数の**Activities**を実行することで，並列・分散処理を行う



- X10の特徴的な構文・データ構造

細粒度並列処理	分散処理	実行の同期	排他制御	分散データ構造
• **async** S	• **at** (place) S	• **finish** S	• **atomic** S	• Point, Region, Dist
• **future** S		• Clock	• **when** (cond) S	• DistArray

Places

- Placeはプロセッサ（もしくは1台のマシン）に対応すると考えればよい
  - 変更可能（mutable）なデータは特定のPlaceに所属している（参照は可能）
- Placeの構成と数（Place.MAX_PLACES）は実行中には変化しない
  - Placeには，0からの続き番号（id）が振られている
- x10.lang.**Place**はビルトインのstruct
  - プロパティ「id」に番号が入る
- **here**は「現在のPlace」を示す
  - here.next()は「隣のPlace」を示す（idは一周する）
- ClassName**!**は，ClassName {self.home==here}の省略記法*
  - そのPlaceのオブジェクトしか代入できないことを示す
- async文で，同じPlace内に新しいActivityを生成できる
- at文で他のPlaceに移動できる

```
package x10.lang;
public final struct Place(id:Int)  {
   public const MAX_PLACES = 4; // in case of Java back-end
   public const places
      = ValRail.make[Place](MAX_PLACES, ((id:Int)=>Place(id)));
   public const FIRST_PLACE:Place(0) = places(0) as Place(0);

   public def this(id:Int):Place(id) { property(id); }
   public static def place(id: Int):Place(id) = Place(id);
   public def next():Place = next(1);
                  :
}
// in x10/lang/_.x10
public static type Place(id:Int) = Place{self.id==id};
```

 * X10 2.1では変更の可能性あり

Place Changing

x10-206.pdf § 14.2, § 14.9, § 14.3

- **at (p) S**
  - Activityがpで指定されたPlaceに移動して，Sを実行する
    - 新しいActivityができるわけではない（少なくとも意味的には）
    - pにはPlaceのほかObjectも指定可*
  - Sが終了すると次の文へ進む
    - Sがasyncで生成したActivitiesは終了していなくてもよい
  - S内から外側ブロックのval変数にアクセス可能
  - S内から値を返せる（Sは式でもよい）
- **async (p) S**
  - async at (p) Sの省略記法
  - pにActivityを生成し，Sを非同期実行

```
public class AtExample {
    static class C { // a typical mutable object
        var f:Int = 0; // accessibe only at home
        def m(s:String)=Console.OUT.println(s+f);
    }

    static def copyRemoteFields(a:C, b:C) {
        at (b) b.f = at (a) a.f;
    }
    static def invokeRemote(obj:C, arg:String) {
        at (obj) obj.m(arg);
    }
    public static def main(args:Rail[String]) {
        val a = new C(); a.f = 10;  // at Place0
        val x = at (here.next()) {
                    val b = new C();    // at Place1
                    copyRemoteFields(a, b);
                    return b; // can return a value
                };
        invokeRemote(x, "x is ");  // -> x is 10
    }
}
```

 * X10 2.1では変更の可能性あり

Activities

x10-206.pdf § 14.3, § 14.4

- **async S**
  - 同じPlaceに子Activityを生成し，Sを実行する
    - asyncを実行した親Activityは，すぐに次の文の実行へ進む
  - S内から外側ブロックのval（およびshared var）変数にアクセス可能
  - 注：async文はハンドルのようなものは返さない．また，S内から値を返すと，（現在の実装では）処理完了までブロックするので返さない方がよい
- **finish S**
  - SとS内で生成された全てのActivity（孫も含む）が終了するまで待つ
  - S内で起こった例外をトラップしてthrowする（rooted exception model）
    - 複数起きたらMultipleExceptions

```
public class Fib { // compute a Fibonacci number
    var r:Int; // in-out parameter
    def this(i:Int) { r = i; }
    def run() {
        if (r < 2) return;
        val f1 = new Fib(r-1), f2 = new Fib(r-2);
        finish {
            async f1.run();// compute Fib(r-1) in parallel
                  f2.run(); // compute Fib(r-2)
        }
        r = f1.r + f2.r; // Fib(r) = Fib(r-1) + Fib(r-2)
    }
    public static def main(args:Rail[String]) {
        val n = (args.length>0)? Int.parse(args(0)) : 10;
        Console.OUT.println("Compute Fib("+n+")");
        val f = new Fib(n); f.run();
        Console.OUT.println(f.r); // -> 55
    }
}
```

Rooted Exception Model

x10-206.pdf § 14.1

- finish Sは，SとSの子孫Activitiesで起きた例外をトラップしてthrowする
  - つまり，finishを囲うtry～catchで，非同期に実行したコードの例外もつかまえて対処可能
    - Activitiesが木構造の親子関係を持ち，finishで全ての子孫の終了を確定できるので，これが可能となる **Rooted** Exception Model
  - 複数のActivitiesで例外が起きた場合は，x10.lang.MultipleExceptionsになる

```
public class AsyncException {
    static class R { var v:Int=0; } // to receive a result
    static def asyncCalc(i1:Int, i2:Int) {
        Console.OUT.print(i1 + "," + i2 + " ==> ");
        val r1=new R(), r2=new R();
        try {
            finish {
                async r1.v = 100 / i1;
                async r2.v = 100 / i2;
            }
        } catch (e:Exception) {
            Console.OUT.print(e.typeName()+" ");
        }
        Console.OUT.println(r1.v + "," + r2.v);
    }
    public static def main(args:Rail[String]) {
        asyncCalc(1, 2); // -> 100,50
        asyncCalc(0, 2); // -> x10.lang.ArithmeticException 0,50
        asyncCalc(0, 0); // -> x10.lang.MultipleExceptions 0,0
    }
}
```

Initial Activity

x10-206.pdf § 14.5

- X10プログラムの開始と終了
  - Place 0の「initial activity」が指定クラスのmainを実行することで開始
  - Activitiesはinitial activityからの木構造を成す
  - それらが全部終了するとプログラムが終了する
    - mainの外側に，右の仮想的なfinish文があると考えればよい

```
public class C {
    public static def main(args:Rail[String]!):Void { ... }
                    :
}
```

```
finish async (Place.FIRST_PLACE) { // initial activity
    // prepare args, then
    C.main(args);
}
```

- Activityの終了とは?
  - ローカルな終了 – そのActivity自体の文が全部終わる
    - at S は，Sがローカル終了すれば先に進む
  - グローバルな終了 – さらに，生成したActivitiesが（孫も含め）全部終わる
    - finish S は，Sがグローバル終了するまでブロックする

Miscellaneous Syntax

- **foreach (x in collection) S**
  - for (x in collection) async S の省略記法
  - コレクションの各要素xに対する計算が並列に行われる
- **ateach (pt in dist) S**
  - foreach (pt in dist.region) at (dist(pt)) S の省略記法
  - Dist dist内各Point ptに対応するPlaceで，Sが並列に実行される
  - 実際には，同じPlaceへの移動はまとめて行われる（べき）
- **f = future E**
  - 式Eの計算が非同期に行われる（Activityが暗黙的に生成される）
  - Future[T]が返される（Tは式Eの型）
  - f.force()を呼ぶと，計算が完了する（Eがグローバル終了）までブロック
  - 計算結果は，f()で取り出せる
- **@Uncounted async S**
  - finishで待たれない，デーモン的に動作するActivityが作れる

x10-206.pdf § 14.6, § 14.7, § 14.8

```
import x10.compiler.Uncounted;
public class MiscExample {
    static def longCompute(n:Int) {       // add 1 to n
        var s:Long=0; for ((i) in 1..n) s += i;
        Console.OUT.println("finished for n="+n);
        return s;
    }
    public static def main(args:Rail[String]) {
        for ((i) in 1..5) Console.OUT.println(i); // -> 1,2,3,4,5
        finish foreach ((i) in 1..5)
            Console.OUT.println(i);           // -> 2,1,3,5,4 (etc.)
        finish ateach (pt in Dist.makeUnique())
            Console.OUT.println(here.id);  // -> 3,0,1,2 (etc.)
        val future_a = future longCompute(100000);
        val b = "another long computation here";
        Console.OUT.println(future_a.typeName()); // -> Future[Long]
        future_a.force();                     // block until the termination
        val a = future_a();                   // get the result
        Console.OUT.println(a);               // -> 5000050000
        finish {
            @Uncounted async {
                val s = longCompute(200000);
                Console.OUT.println(s);       // -> 20000100000 (after "end")
            }
        }
        Console.OUT.println("end");
    }
}
```

Collecting Finish

x10-206.pdf § 14.4.1

- **finish (Reducible[T]) S**
  - S内から**offer t**で渡されるT型の値を，reducer.apply(T,T)で集計した結果を返す
  - 一つのActivityが複数のofferをしてもよい（全部集計される）
  - Sの部分はasyncやatで並列計算してもよい

```
public class CollectingFinish {
   static struct IntAdd implements Reducible[Int] {
      public global safe def zero()=0;
      public global safe def apply(a:Int,b:Int)=a+b;
   }
   public static def main(args:Rail[String]) {
      val v = finish (IntAdd()) {
         for ((i) in 1..10) async { offer i; offer i*100; }
      };
      Console.OUT.println(v); // -> 5555
} }
```

```
public class MRex { // from x10-206.pdf 14.4.1
   static struct Accum(n:Int, sum:Int, sumSq:Int) {
      def this(n:Int, sum:Int, sumSq:Int) { property(n, sum, sumSq); }
   }
   static struct StatReducer implements Reducible[Accum] {
      public global safe def zero() = Accum(0, 0, 0);
      public global safe def apply(a:Accum, b:Accum)
         = Accum(a.n+b.n, a.sum+b.sum, a.sumSq+b.sumSq);
   }

   static def accumulate(data:ValRail[Int]):Accum {
      val v = finish (StatReducer()) {
         for (x in data) offer Accum(1, x, x*x);
      };
      Console.OUT.println(v); // for debug
      return v;
   }
   public static def main(args:Rail[String]) {
      val data =[5, 6, 8, 4, 7, 4, 2, 3, 6, 6, 3, 5, 4, 5, 6, 5, 4, 5];
      val a:Accum = accumulate(data); // accumulate n, sum, sumSq
      val mean = (a.sum as Double) / a.n;
      val variance = (a.sumSq as Double) / a.n - mean*mean;
      val sdev = Math.sqrt(variance);
      Console.OUT.printf("mean=%f sdev=%f¥n", mean, sdev);
                        // -> mean=4.888889 sdev=1.448712
} }
```

Atomic Blocks

x10-206.pdf § 14.11

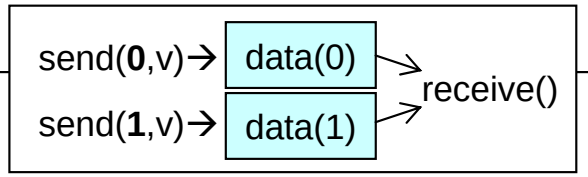
- 同一Place内のActivityの排他制御機構
  - 注：X10では，異なるPlace間の排他制御はできない（そもそも書き込めるデータはPlace間で共有できない）
  - 注：Javaのような，オブジェクトを指定した排他制御はできない
- **atomic S**
  - SはPlace内で不可分に実行される
  - S内ではブロックしうる文は使えない
    - at, async, finish, future, force(), when, nextなどは使えない
    - 呼べるのはsafeなメソッドだけ
    - atomicを入れ子にすることは可能
- **atomic def m() ～**
  - そのメソッド自体をatomicブロックで囲ったのと同じ効果

```
public class AtomicExample[T] {
  var target:T; def this(init:T) { target = init; }
  atomic def CAS(o:T, n:T):Boolean {
    if (target == o) { target = n; return true; }
    return false;
  }

  public static def main(args:Rail[String]) {
    val o = new AtomicExample[Int](0);
    finish foreach ((i) in 1..5)
      Console.OUT.println(o.CAS(0, i)); // -> only 1 true

    val v = Rail.make[Int](1); v(0)=0;
    finish foreach ((i) in 1..1000) {
      val tmp = i * i;
      atomic { v(0) += tmp; }
    }
    Console.OUT.println(v(0)); // -> 333833500
  }
}
```

- 他に，x10.util.concurrent.atomic.AtomicIntegerなどのクラスも使用可能

IBM

Conditional Atomic Blocks



x10-206.pdf § 14.11.2

- 指定された条件（のどれか）が成立するまでブロックする構文

- **when (cond) S**
  - condが成立するまでブロックし，成立したらSが不可分（atomic）に実行される
  - condやSにはatomic文のSと同じ制約がある（つまりwhen文は入れ子にできない）

- **when (cond1) S1 or (cond2) S2 or ...**
  - どれかのcondが成立するまでブロックし，成立した条件のSが不可分に実行される
  - when (cond1 || cond2) {
    if (cond1) S1; else if (cond2) S2;
    } と同じ

- **await(cond)**
  - condが成立するまでブロックし，成立したら次の文へ進む
  - when (cond) { ; } と同じ

```
public class TwoChannel[T] {
   val data = Rail.make[T](2); // 1 slot for each chan
   val filled = Rail.make[Boolean](2,(Int)=>false);
   def send(ch:Int, v:T) {
      when (!filled(ch)) { data(ch)=v; filled(ch)=true; }
   }
   def receive():T {
      var v:T;
      when (filled(0)) { v=data(0); filled(0)=false; }
      or (filled(1)) { v=data(1); filled(1)=false; }
      return v;
   }

   public static def main(args:Rail[String]) {
      val chan = new TwoChannel[Int]();
      finish {
         async for ((i) in 1..5) chan.send(0, i);
         async for ((i) in 11..15) chan.send(1, i);
         async for ((i) in 1..10) // receiver
            Console.OUT.println(chan.receive());
                  // -> 1,11,12,2,13,3,4,5,14,15 (etc.)
} } }
```

Clocks

- **クロック**は，Place内のバリア同期のための機構
  - 複数のActivitiesを「登録」できる
  - 登録された全Activitiesがnextを呼ぶまでブロック
- **c = Clock.make()**
  - クロックcを生成，自分（Activity）はcに「登録」される
- **async clocked(c) S**
  - クロックcに登録されたActivityを生成する
  - 自分がcに登録されていないとClockUseException
  - 注：よそからもらってきたクロックに自分を追加登録することはできない
  - クロックは複数指定することも可能
- **next**
  - 自分がその回（フェーズ）の実行を終えたことを通知
  - クロックに登録されている全Activitiesがnextを実行するまでブロックする
  - Acitvityが複数のクロックに登録されている場合，登録されている全てのクロックに対して行われる

```
public class ClockExample {
  static atomic def say(s:String)
                    = Console.OUT.println(s);
  public static def main(args:Rail[String]) {
    finish async { // this async is necessary
      val c = Clock.make();
      async clocked(c) { // clocked activity A
        say("A-1   "); next;
        say("   A-2"); next;
        say("A-3   ");
      }
      async clocked(c) { // clocked activity B
        Activity.sleep(1000); // sleep 1sec
        say("B-1   "); next;
        say("   B-2"); next;
        say("B-3   ");
      }
      async clocked(c) { // clocked activity C
       for ((i) in 1..3) {
        val spc = (i%2==0) ? "   " : "";
        say(spc + "C-" + i); next;
      }}
  } } }
```

実行例

```
A−1
C−1
B−1
    C−2
    B−2
    A−2
B−3
C−3
A−3
```

Clocks, Advanced

- Clockオブジェクトに対する操作
  - c.next()
    - 特定のクロックcについてだけnext処理
  - c.resume()
    - その回の実行完了を(早めに)通知
    - ブロックしない，nextはこの後で別途実行する必要がある
    - 他のActivitiesが早めに「次の回」に進める
  - c.drop()
    - 自分(Activity)のクロックcへの登録を解除

- Clock使用上の注意点
  - クロックを作ったActivityも，そのクロックに「登録」される(そうしないと，引き続くasync clocked(c)が実行できない)
  - つまり，そのActivityがnextを呼ぶか，終了もしくはc.drop()を呼んでクロックから登録解除されないと，クロックが先に進めなくなる！
  - そのため，右の例でfinishの後のasyncが無いと，A-1,B-1,C-1を表示したところで止まってしまう
    - asyncを使わず，finishの終了直前にnext;next;かc.drop();を入れることでも回避可能

```
public class ClockExample {
  static atomic def say(s:String)
                    = Console.OUT.println(s);
  public static def main(args:Rail[String]) {
    finish async { // this async is necessary
      val c = Clock.make();
      async clocked(c) { // clocked activity A
        say("A-1   "); next;
        say("   A-2"); next;
        say("A-3   ");
      }
      async clocked(c) { // clocked activity B
        Activity.sleep(1000); // sleep 1sec
        say("B-1   "); next;
        say("   B-2"); next;
        say("B-3   ");
      }
      async clocked(c) { // clocked activity C
       for ((i) in 1..3) {
        val spc = (i%2==0) ? "   " : "";
        say(spc + "C-" + i); next;
      }}
  }}}
```

実行例
```
A-1
C-1
B-1
    C-2
    B-2
    A-2
B-3
C-3
A-3
```

Deadlock Freedom

x10-206.pdf § 15.2, § 2.8.2

- 以下の条件を満たすX10プログラムはデッドロックしない（できない）

1. at, async, finish, atomicは自由に使ってよい
   - ちなみに，X10のasyncが「ハンドル」を返さないのは，デッドロックするプログラムを本質的に書けなくするため
2. whenやawaitは使わない
   - futureもたぶんダメ
3. クロックは使ってよいが，以下の条件を満たすこと
   - c.next()を使わない（next文はOK）
   - finish Sにおいて，S内のクロック付きasyncは全てクロック無しasyncの制御下にある
     - finishより外のクロックを使わないためのルール

不適切なクロック使用でデッドロックする例

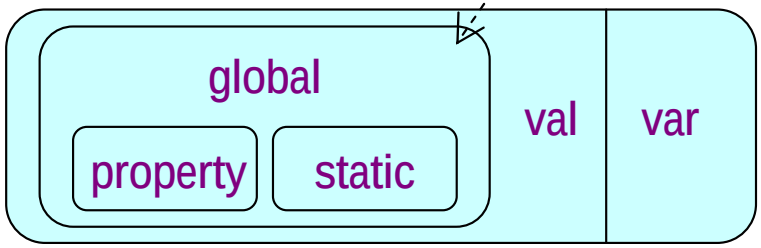
```
val c = Clock.make();
async clocked(c) {                  // clocked activity A
   finish async clocked(c) { // clocked activity B <==
      next;   // B-next, blocks until A-next
   }
   next;      // A-next, never executed
}
```

X10 2.0 Distributed Object Model*

x10-206.pdf §8.1.2, §8.7

- 基本は,「**変更可能(mutable)なものは特定のPlaceに所属**(ただし参照は渡せる)」
  - 逆に, 変更不能(immutable)なものはどのPlaceからも見える(ようにできる)

- **オブジェクト**
  - 各オブジェクトは, 生成されたPlace(=home)に所属する. homeは変更できない
    - at (p) Sにより, S内(別Place)で用いるための「リモート参照」が作られる
  - オブジェクトはグローバルな同一性を持つ,「==」で比較可能(リモート参照も)
    - リモート参照がhomeに戻ってくると同じ値のローカル参照に戻る
  - 変更可能(var)フィールドは, homeでのみアクセス可能
  - 変更不能(val)フィールドは, global宣言されていると他のPlaceからも読める
  - インスタンスメソッドは, global宣言されているとリモート参照からも呼べる
    - ただし, globalフィールドにしか直接アクセスできない(at (o.home)経由は可)
- **Structs**
  - フィールドは全て変更不能(val), 暗黙のうちにglobal
  - StructはPlace間で自由に複製できる
  - 「==」の判定は内容に対して行われる
- **関数**
  - これも, Place間で自由に複製できる
- **クラス**
  - スタティックフィールドは変更不能(val), 暗黙のうちにglobal
  - クラスの初期化はPlace 0で行われ, 他のPlaceにコピーされる



フィールド修飾子の包含関係(再掲)

 * X10 2.1では変更の可能性あり

Distributed Arrays

x10-206.pdf § 16

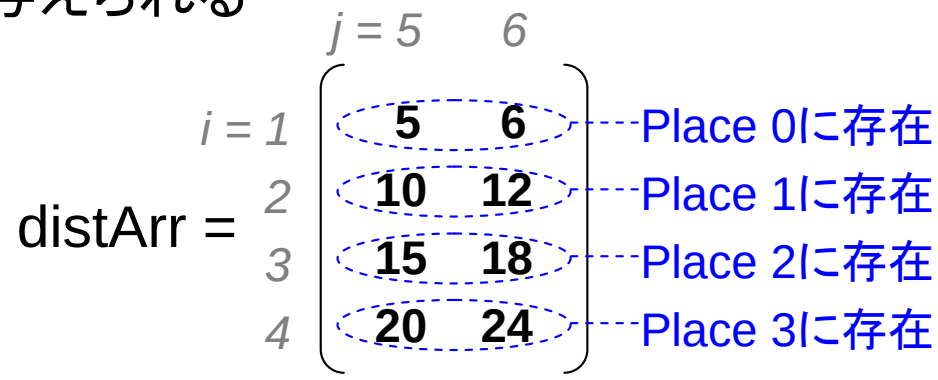
DistArray: http://dist.codehaus.org/x10/xdoc/2.0.6/x10/array/DistArray.html

- X10では，要素が各Placeに散らばった配列（**DistArray**）を作成可能
  - val arr = DistArray.make[T](Dist, ...);で作成し，arr(pt)などでアクセス
  - 各要素は**Point**（n次元の点）でインデクシングされる
  - 配列の定義域は**Region**（次元が同じPointの集合）で与えられる
  - 各要素がどのPlaceに存在するかは，**Dist**（Region内の各PointからPlaceへの写像）で与えられる

```
public class DistArrayExample {
  public static def main(args:Rail[String]) {
    val R = [1..4, 5..6] as Region; Console.OUT.println(R);
    val D = Dist.makeBlock(R);      Console.OUT.println(D);
    val distArr = DistArray.make[Int](D,((i,j):Point)=>i*j);
    for (pt in R) at (D(pt)) {
      Console.OUT.println(
        "distArr" + pt + "=" + distArr(pt) + " at " + here);
} } }
```

実行結果

```
[1..4,5..6]
Dist([1..1,5..6]->0,[2..2,5..6]->1,
     [3..3,5..6]->2,[4..4,5..6]->3)
distArr(1,5)=5 at (Place 0)
distArr(1,6)=6 at (Place 0)
distArr(2,5)=10 at (Place 1)
distArr(2,6)=12 at (Place 1)
distArr(3,5)=15 at (Place 2)
distArr(3,6)=18 at (Place 2)
distArr(4,5)=20 at (Place 3)
distArr(4,6)=24 at (Place 3)
```



- ほぼ同じ枠組みで，ローカル配列（Array）も作成可能

Point and Region

x10-206.pdf §16.1, §5.3, §16.2

- x10.array.**Pointは<u>n次元上の1点</u>を表す**. n次元配列の「添え字」として使用できる
  - 例：val pt = [1,3,7] as Point; → 3次元上のPoint (1,3,7)
    - pt(1)などで対応する要素を取り出せる（上の例だと3）
  - Pointの要素はIntに固定で変更不能，次元はrankプロパティでわかる
  - Pointのdestructuring access
    - val (i,j,k) = [1,3,7] as Point; → i=1, j=3, k=7
    - val p(i,j) = Point.make(2,3); → p=(2,3), i=2, j=3
    - for ((i) in 1..10) ～ も実はこれの一種，なので括弧は省略できない

- x10.array.**Regionは次元（rank）が同じ<u>Pointの集合</u>**. 配列の「有効な添え字」を表す
  - 例：val R1 = **1..5**; → 1次元のPoint (1)(2)(3)(4)(5)からなるRegion{rank==1}
  - 例：val R2 = [1..2, 3..5] as Region;
    → 2次元のPoint (1,3)(1,4)(1,5)(2,3)(2,4)(2,5)からなるRegion{rank==2}
  - 注：Regionに実際にPointが入っているわけではなく，集合の「形」を覚えている
  - Region同士の演算もいろいろ定義されている（省略）

- x10.array.**Array**[T]**はローカルな配列**. 指定されたRegionが定義域，各要素はT型
  - 例：val arr = new Array[Int](1..5); arr(3) = ...
  - X10の配列はゼロから始まらなくてもよいし，要素に「すき間」があってもよい

Dist and DistArray

- x10.array.**Dist**はRegion内の各PointからPlaceへの写像
  - 分散配列の各要素がどのPlaceに存在するかを表す
  - 「D | place」や「D | region」で，要素を制限できる
- x10.array.**DistArray**[T]は要素が分散して存在する配列
  - 定義域と要素の存在場所は作成時にDistで指定される
    - 後から変更はできない
  - newではなく，DistArray.**make**[Int](D, ...)で作る（今のところ）
    - initializer関数も指定可
  - Distと同様に「|」で要素の制限が可能

```
public class DistExample {
  static def Dprint(name:String,D
    Console.OUT.print(name+":"
    for (pt in D) Console.OUT.pri
    Console.OUT.println();
  }
  public static def main(args:Rail[String]) {
    Console.OUT.println(Place.places);
    val R1 = 1..6, place1 = here.next();
    val D1 = Dist.makeUnique();                Dprint("D1",D1);
    val D2 = R1->here;                         Dprint("D2",D2);
    val D3 = Dist.makeConstant(R1,place1);     Dprint("D3",D3);
    val D4 = Dist.makeBlock(R1);               Dprint("D4",D4);
    val D5 = D4 | place1;                      Dprint("D5",D5);
    val D6 = D4 | 3..5;                        Dprint("D6",D6);
    val D7 = Dist.makeUnique(D6.places());     Dprint("D7",D7);

    val a1 = new Array[Int](6); // simple array allocation
    val a2 = new Array[Int](R1,((i):Point)=>i);
    val d1 = DistArray.make[Int](D4,(p:Point(1))=>at(a2) a2(p));
    val d2 = d1 | 3..5;
    for (p in d2) Console.OUT.print(at (d2.dist(p)) d2(p));
    Console.OUT.println();
} }
```

実行結果

```
[(Place 0), (Place 1), (Place 2), (Place 3)]
D1: (0)->0 (1)->1 (2)->2 (3)->3
D2: (1)->0 (2)->0 (3)->0 (4)->0 (5)->0 (6)->0
D3: (1)->1 (2)->1 (3)->1 (4)->1 (5)->1 (6)->1
D4: (1)->0 (2)->0 (3)->1 (4)->1 (5)->2 (6)->3
D5: (3)->1 (4)->1
D6: (3)->1 (4)->1 (5)->2
D7: (0)->1 (1)->2
345
```

x10-206.pdf § 16.4, § 16.5, § 16.6

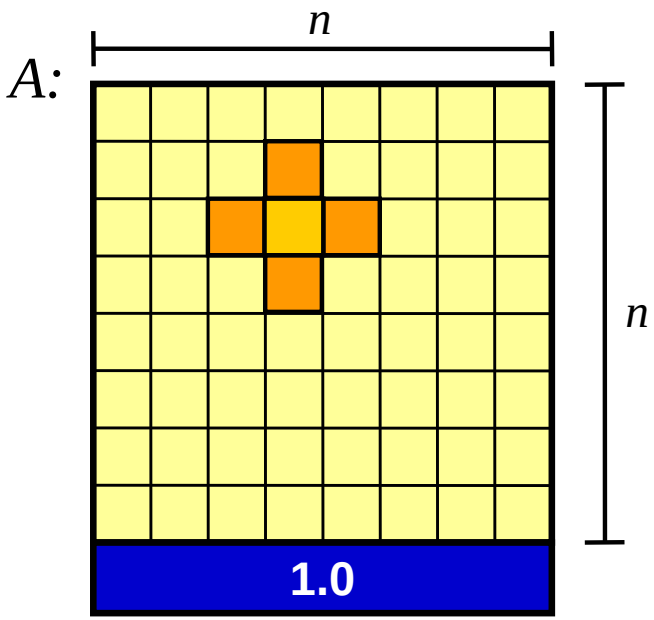
Example Distributed Program

- **$1^2 + 2^2 + \cdots + 1000^2$** の計算を複数Placeで分担して行うプログラム
- 1～1000の値をDistArray arrに用意
- 各Placeでの計算結果を入れるRail tmpを用意
- 各Placeで並行して以下の処理を行う
  - 自Placeにある要素vについて，v*vを求め，合計する
- 各Placeでの計算が終わったら，tmpの要素を合計し，結果を出力
- 実は，同様の処理を行うための便利メソッドが用意されている
  - map(func:(T)=>T):DistArray[T]
  - reduce(func:(T,T)=>T, unit:T):T

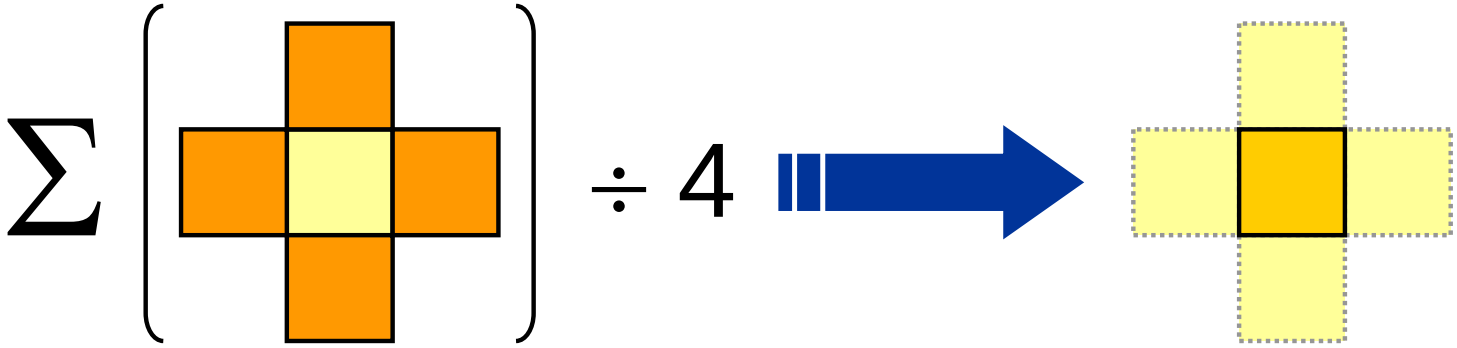
```
public class MyDistCalc {
    public static def main(args:Rail[String]) {
        val R = 1..1000; val D = Dist.makeBlock(R);
        val arr = DistArray.make[Int](D, ((i):Point)=>i);

        val places = arr.dist.places();  // ValRail[Place]
        val tmp = Rail.make[Int](places.length);
        finish foreach ((i) in 0..places.length-1) { //  =async
            tmp(i) = at (places(i)) {
                val a = arr | here;
                var s:Int = 0; for (p in a) s += a(p)*a(p);
                s  // return value of at
            };
        }
        var result:Int = 0; for (i in tmp) result += i;
        Console.OUT.println(result);    // -> 333833500

        // We can actually use DistArray.map and reduce
        val r = arr.map((i:Int)=>i*i).reduce(Int.+, 0);
        Console.OUT.println(r);         // -> 333833500
    }
}
```

Distributed 2-D Heat Transfer

from GroveX10 UT June 2010.ppt



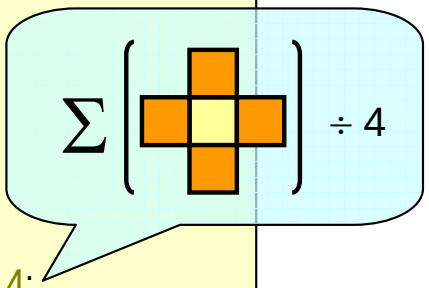
repeat until max change < ε



Distributed 2-D Heat Transfer

modified from GroveX10 UT June 2010.ppt

```
public class MyHeatTransfer { // from "GroveX10 UT June 2010.ppt", modified by kawatiya@jp.ibm.com
    static val N=5, epsilon=0.01;
    static val BigD = Dist.makeBlock([0..N+1, 0..N+1], 0);   // Dist for the bigger region including the border
    static val D = BigD | ([1..N, 1..N] as Region);          // Dist for the NxN region, which is part of BigD
    static val A = DistArray.make[Double](BigD, ((i,j):Point)=>(i==0 ? 1.0 : 0.0)); // main DistArray
    static val Temp = DistArray.make[Double](BigD);

    static def dump(X:DistArray[Double](2), last:Int) { Console.OUT.println("--------");
        for (p(i,j) in X) Console.OUT.printf("%3.2f%c", at (X.dist(p)) X(p), j!=last ? ' ' : '¥n');
    }
    static def stencil_1((x,y):Point(2)) { // heat transfer for a point
        return ( (at(A.dist(x-1,y)) A(x-1,y)) + (at(A.dist(x+1,y)) A(x+1,y)) + A(x,y-1) + A(x,y+1) ) / 4;
    }
    public static def main(args:Rail[String]) {
        Console.OUT.println("N=" + N);
        val D_Base = Dist.makeUnique(D.places());
        var delta:Double = 1.0;
        do {    dump(A, N+1); // to check intermediate states
            finish ateach (z in D_Base) // at each place of D, do parallel
                for (p:Point(2) in D | here) Temp(p) = stencil_1(p);
            delta = A.map(Temp, D.region, (x:Double,y:Double)=>Math.abs(x-y))
                        .reduce(Math.max.(Double,Double), 0.0); // get max delta
            finish ateach (p in D) A(p) = Temp(p); // copy the array in parallel
        } while (delta >= epsilon);  dump(A, N+1);
}   }
```



Result

```
N=5
--------
1.00 1.00 1.00 1.00 1.00 1.00 1.00
0.00 0.45 0.60 0.64 0.60 0.45 0.00
0.00 0.22 0.33 0.37 0.33 0.22 0.00
0.00 0.10 0.17 0.19 0.17 0.10 0.00
0.00 0.05 0.08 0.09 0.08 0.05 0.00
0.00 0.02 0.03 0.03 0.03 0.02 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

Native Code Integration

- @Nativeというアノテーションを使って，X10を実装している言語（JavaまたはC++）のコードを呼び出すことができる*
  - @Nativeはメソッドまたはブロックに指定可能で，X10コンパイル時に変換される

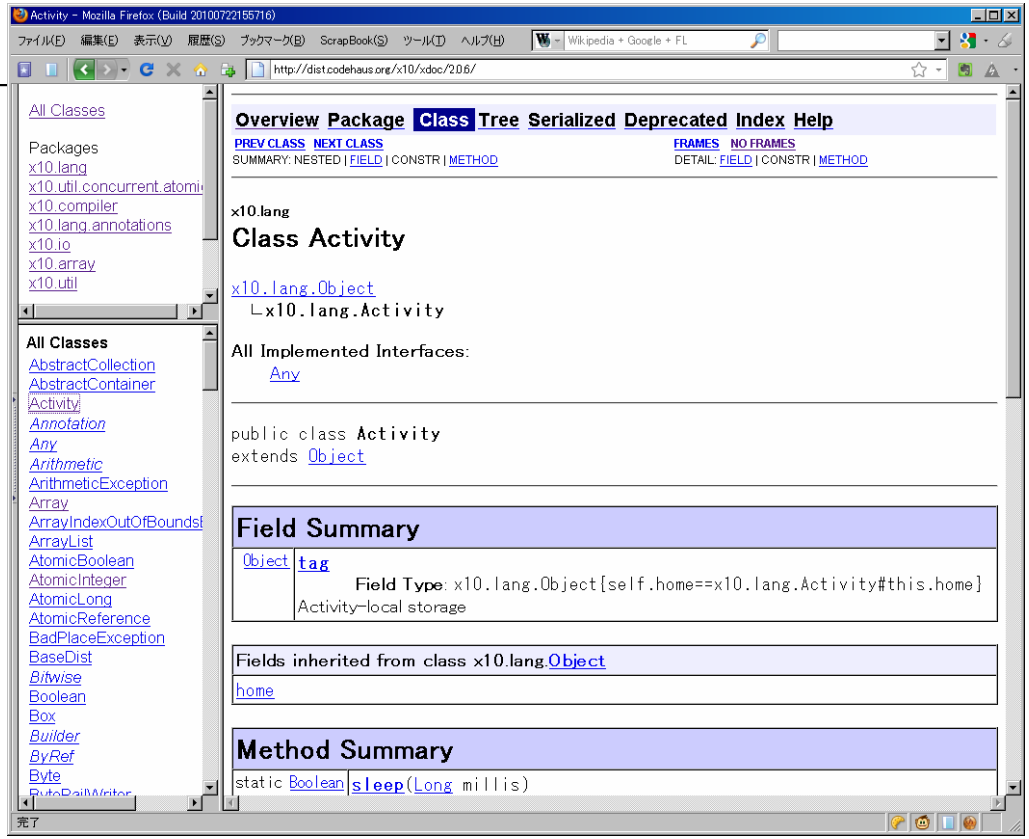
```
import x10.compiler.Native; // to use @Native annotation
public class NativeExample {
   @Native("c++", "printf(¥"%d+%d=%d¥¥n¥", (#1), (#2), (#1)+(#2)) /*C++*/")
   @Native("java", "System.out.println((#1) + ¥"+¥" + (#2) + ¥"=¥" + ((#1)+(#2))) /*Java*/")
   static native def nativeAdd(x:Int, y:Int):Void; // call to this method is replaced

   public static def main(args:Rail[String]) {
      val a=1, b=2;
      nativeAdd(a, b); // -> 1+2=3

      val s:Int;
      @Native("c++", "s = 1; /*C++*/")
      @Native("java", "s = 2; /*Java*/")
      { s = 0; /*Unknown*/ } // this block is replaced
      val backEnd = (s==1)?"C++" : (s==2)?"Java" : "Unknown";
      Console.OUT.println("I am on " + backEnd); // -> I am on Java  (etc.)
} }
```

x10c (Java back-end)が生成したコードの一部

```
final int a = 1;
final int b = 2;
System.out.println((((int)(a))) + "+" + (((int)(b))) + "="
+ ((((int)(a)))+(((int)(b))))) /*Java*/;

final int s;
s = 2; /*Java*/
   :
```

 * X10 2.1では変更の可能性あり

X10 Standard Library



- RailやArrayに加え，Math, HashMapなどの基本的な機能が用意されている
  - http://dist.codehaus.org/x10/xdoc/ で仕様を確認可能
    - Javadoc形式なので少し見づらい，ソースを見る方が早いかも・・・

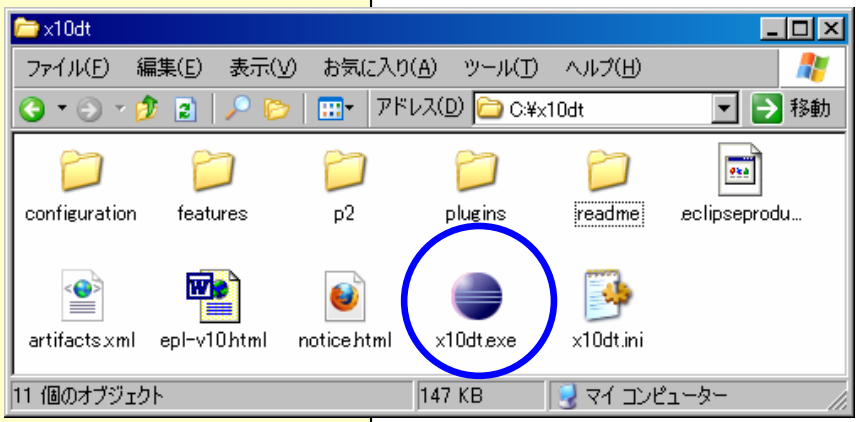
- **用意されている主なクラス（またはstruct）**
  - x10.lang.Any, Object, Int, String,
    Rail[T], ValRail[+T], Place, Activity, Future[T], Clock,
    System, Math, Complex, Cell[T], PlaceLocalHandle[T], ...
  - x10.array.Point, Region, Dist, DistArray[T], Array[T], ...
  - x10.io.Console, File, Reader, Writer, Printer, ...
  - x10.util.HashMap[K,V], Timer, Random, Box[+T], Pair[T,U], GrowableRail[T], ...
  - x10.util.concurrent.atomic.AtomicInteger, AtomicReference[T], Fences, ...
  - x10.lang.*とx10.array.*は，自動的にimportされる

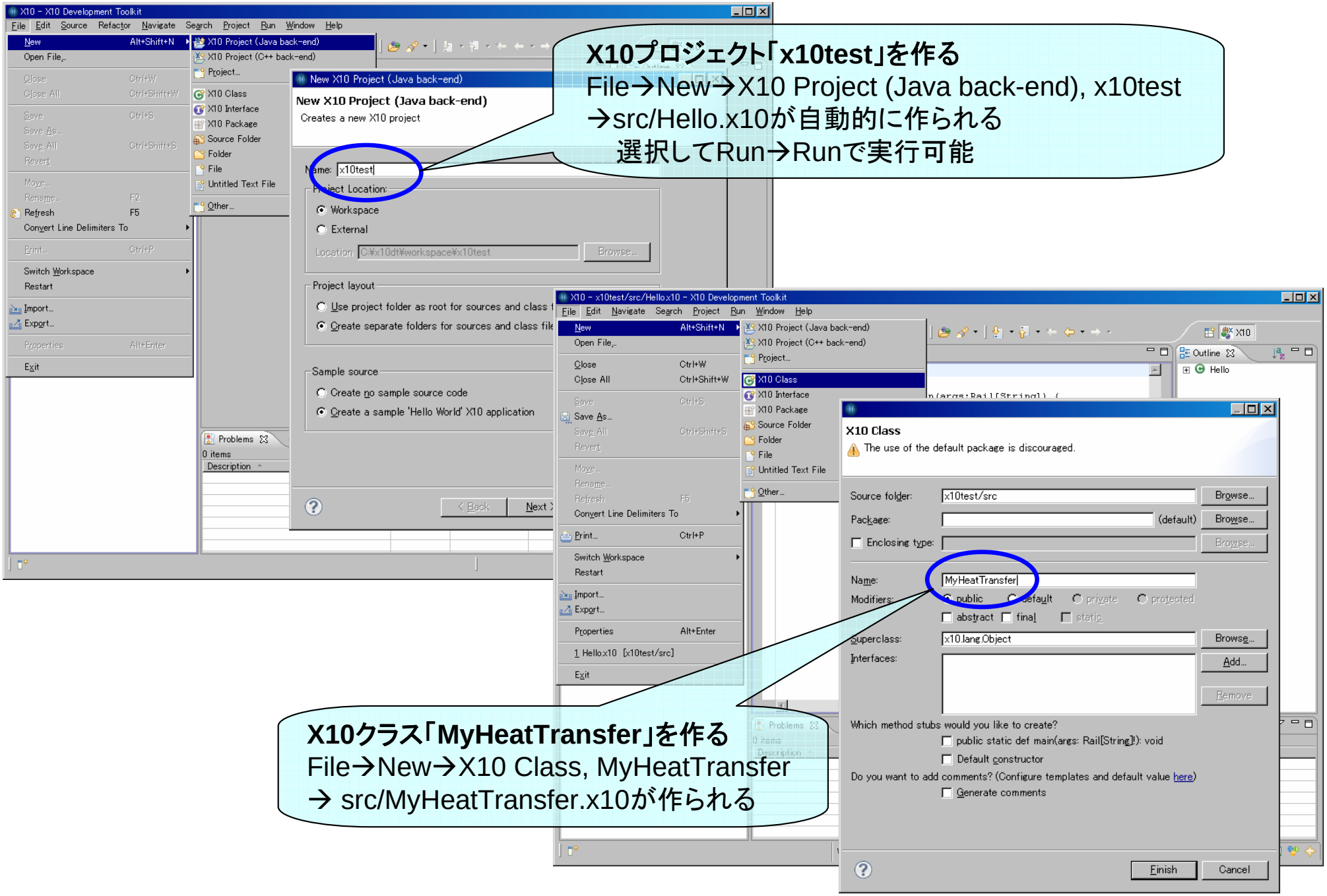
3. X10を試してみる

X10を試してみる

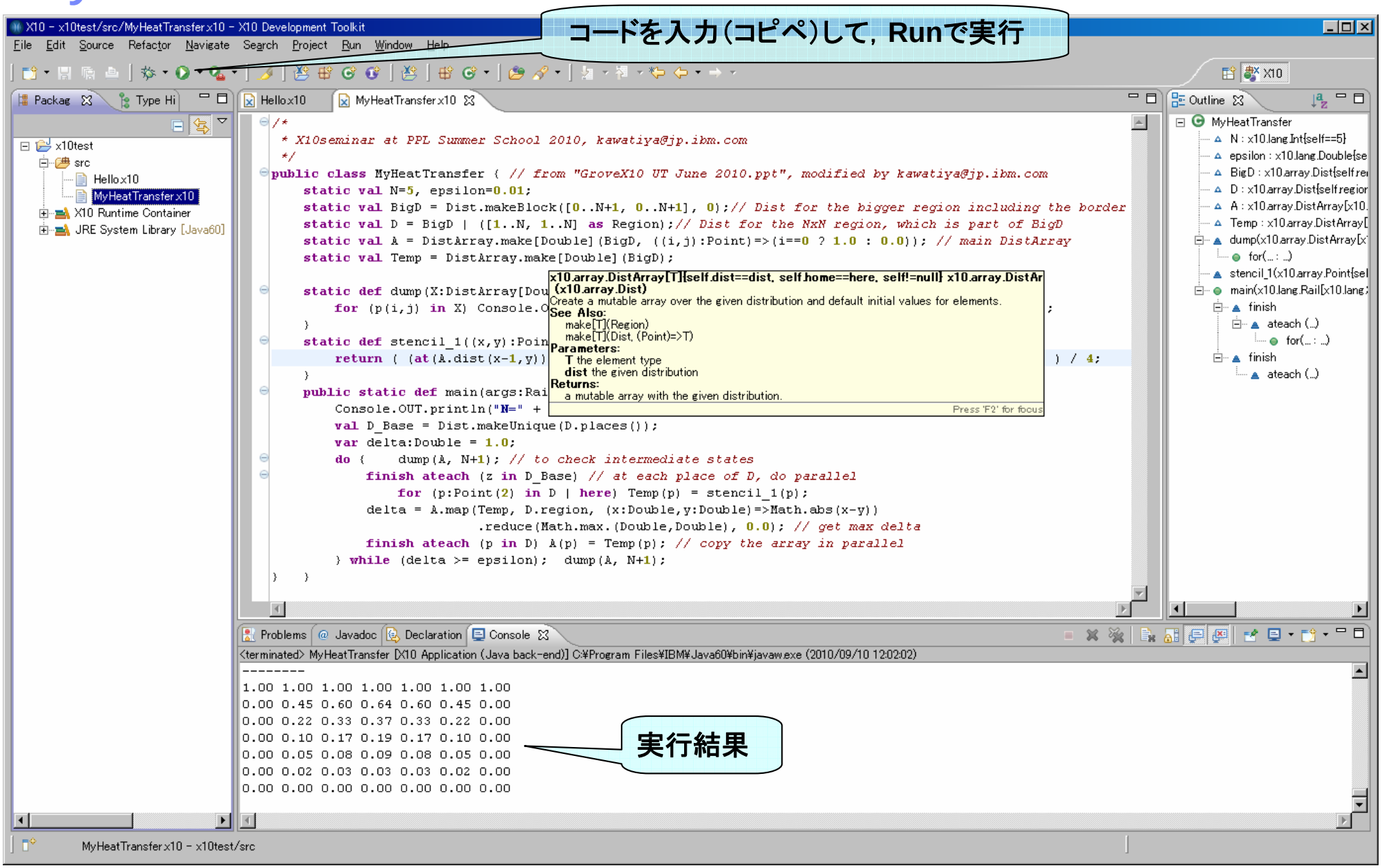
- Eclipse上にX10DTをインストールするのが最も簡単
  - http://x10-lang.org/X10DT+2.0+Installation
  - バージョン2.0.6からは，Eclipse（3.6.0 Helios）も含んだzipが用意されている
    - Linux x86, Linux x86_64, MacOS x86, Win32+cygwin用がある
- X10DTの解凍と起動例（注：起動にはJava環境が必要）

```
$ ls -l  x10dt-2.0.6-win32.win32.x86.zip
-rwx------ 1 kawatiya  169649357  x10dt-2.0.6-win32.win32.x86.zip
$ md5sum x10dt-2.0.6-win32.win32.x86.zip
916502282d594a32df0708913c719ad6  x10dt-2.0.6-win32.win32.x86.zip
$ cd c:/
$ unzip x10dt-2.0.6-win32.win32.x86.zip
   creating: x10dt/
          :
  inflating: x10dt/readme/readme_eclipse.html
  inflating: x10dt/x10dt.exe
  inflating: x10dt/x10dt.ini
$ cd x10dt
$ ls
./                artifacts.xml     features/     plugins/     x10dt.ini
../               configuration/    notice.html   readme/
.eclipseproduct   epl-v10.html      p2/           x10dt.exe
$ ./x10dt.exe
```



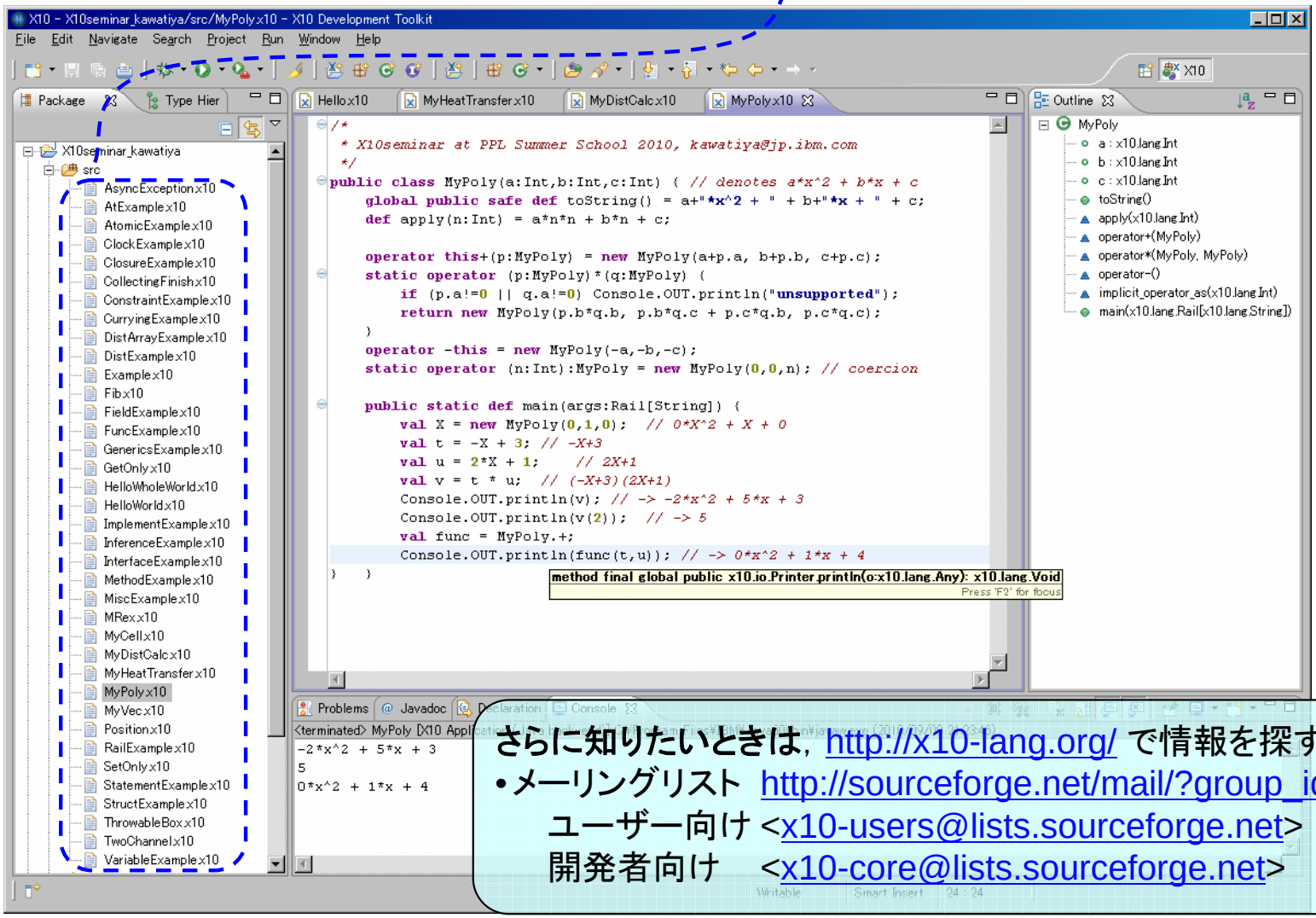


MyHeatTransfer on X10DT



X10をさらに試す

- この資料で使ったサンプルコード→ **X10seminar_kawatiya_src.zip**
  - 解凍して，ソースファイルをX10DTのsrcにドラッグすると取り込めます



さらに知りたいときは，http://x10-lang.org/ で情報を探すか，MLで質問

- メーリングリスト http://sourceforge.net/mail/?group_id=181722
  - ユーザー向け <x10-users@lists.sourceforge.net>
  - 開発者向け <x10-core@lists.sourceforge.net>

ご清聴ありがとうございました