Base Language
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
naïve n-body computation in Chapel
n-body in Chapel (where n == 5)

- A serial computation
- From the Computer Language Benchmarks Game
  - Chapel implementation in release under examples/benchmarks/shootout/nbody.chpl
- Computes the influence of 5 bodies on one another
  - The Sun, Jupiter, Saturn, Uranus, Neptune
- Executes for a user-specifiable number of timesteps

Image source: [http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png](http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png)
5-body in Chapel: Declarations

```chapel
const pi = 3.141592653589793,
solarMass = 4 * pi**2,
daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

...
5-body in Chapel: Declarations

```chapel
const pi = 3.141592653589793,
solarMass = 4 * pi**2,
daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}
```

Variable declarations
Configuration Variable
Record declaration
Tuple type
const pi = 3.141592653589793, solarMass = 4 * pi**2, daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

...
Variables, Constants, and Parameters

● **Basic syntax**

```
declaration:
    var   identifier [: type] [= init-expr];
    const identifier [: type] [= init-expr];
    param identifier [: type] [= init-expr];
```

● **Meaning**

- **var/const**: execution-time variable/constant
- **param**: compile-time constant
- No **init-expr** ⇒ initial value is the type’s default
- No **type** ⇒ type is taken from **init-expr**

● **Examples**

```
const pi: real = 3.14159;
var count: int; // initialized to 0
param debug = true; // inferred to be bool
```
**Chapel’s Static Type Inference**

```
const pi = 3.14, // pi is a real
    coord = 1.2 + 3.4i, // coord is a complex...
    coord2 = pi*coord, // ...as is coord2
    name = "brad", // name is a string
    verbose = false; // verbose is boolean

proc addem(x, y) { // addem() has generic arguments
    return x + y; // and an inferred return type
}

var sum = addem(1, pi), // sum is a real
    fullname = addem(name, "ford"); // fullname is a string

writeln((sum, fullname));
```

(4.14, bradford)
5-body in Chapel: Declarations

```chapel
const pi = 3.141592653589793,
solarMass = 4 * pi**2,
daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}
```

Variable declarations

Configuration Variable

Record declaration

Tuple type
5-body in Chapel: Declarations

```chapel
const pi = 3.141592653589793,
    solarMass = 4 * pi**2,
    daysPerYear = 365.24;

config const numsteps = 10000;

record body {
    var pos: 3*real;
    var v: 3*real;
    var mass: real;
}
```

Configuration Variable

$ ./nbody --numsteps=100
Configs

```plaintext
param intSize = 32;
type elementType = real(32);
const epsilon = 0.01:elementType;
var start = 1:int(intSize);
```
Configs

```chpl
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);
```

```
$ chpl myProgram.chpl -sintSize=64 -selementType=real
$ ./a.out --start=2 --epsilon=0.00001
```
const pi = 3.141592653589793,
solarMass = 4 * pi**2,
daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

...
Records and Classes

● Chapel’s struct/object types
  ● Contain variable definitions (fields)
  ● Contain procedure & iterator definitions (methods)
  ● Records: value-based (e.g., assignment copies fields)
  ● Classes: reference-based (e.g., assignment aliases object)
  ● Record : Class :: C++ struct : Java class

● Example

```plaintext
record circle { 
  var radius: real;
  proc area() { 
    return pi*radius**2;
  }
}
```
```plaintext
var c1, c2: circle;
c1 = new circle(radius=1.0);
c2 = c1; // copies c1
c1.radius = 5.0;
writeln(c2.radius); // 1.0
// records deleted by compiler
```
**Records and Classes**

- Chapel’s struct/object types
  - Contain variable definitions (fields)
  - Contain procedure & iterator definitions (methods)
  - Records: value-based (e.g., assignment copies fields)
  - Classes: reference-based (e.g., assignment aliases object)
  - Record : Class :: C++ struct : Java class

- Example

```plaintext
class circle {
    var radius: real;
    proc area() {
        return pi*radius**2;
    }
}
```

```plaintext
var c1, c2: circle;
c1 = new circle(radius=1.0);
c2 = c1;  // aliases c1’s circle
c1.radius = 5.0;
writeln(c2.radius);  // 5.0
delete c1;  // users delete classes
```
## Classes vs. Records

### Classes
- **heap-allocated**
  - Pointers to fields
  - Requires ‘delete’
- **'ref' semantics**
  - crucial when object identity matters
- **support dynamic dispatch**
- **support inheritance**
- **similar to Java classes**

### Records
- **allocated in-place**
  - Fields in contiguous memory
  - Memory managed
- **'value' semantics**
  - compiler may introduce copies
- **no dynamic dispatch**
- **no inheritance (yet)**
- **similar to C++ structs (sans pointers)**
5-body in Chapel: Declarations

\[
\text{const } \pi = 3.141592653589793, \\
solarMass = 4 \times \pi^2, \\
daysPerYear = 365.24;
\]

\[
\text{config const } \text{numsteps} = 10000;
\]

\[
\text{record body} \\{
\text{var } \text{pos: } 3\times\text{real}; \\
\text{var } \text{v: } 3\times\text{real}; \\
\text{var } \text{mass: } \text{real};
\}
\]
Tuples

● Use
  ● support lightweight grouping of values
    ● e.g., passing/returning procedure arguments
    ● multidimensional array indices
    ● short vectors
  ● support heterogeneous data types

● Examples

```plaintext
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```
const pi = 3.141592653589793,
solarMass = 4 * pi**2,
daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

...
5-body in Chapel: the Bodies

```chapel
var bodies =
  [ /* sun */
    new body(mass = solarMass),

    /* jupiter */
    new body(pos = ( 4.84143144246472090e+00,
                    -1.16032004402742839e+00,
                    -1.03622044471123109e-01),

                    v = ( 1.66007664274403694e-03 * daysPerYear,
                         7.69901118419740425e-03 * daysPerYear,
                         -6.90460016972063023e-05 * daysPerYear),

                    mass = 9.54791938424326609e-04 * solarMass),

    /* saturn */
    new body(...),

    /* uranus */
    new body(...),

    /* neptune */
    new body(...)
  ]
```
5-body in Chapel: the Bodies

```chapel
var bodies =
[  /* sun */
   new body(mass = solarMass),

   /* jupiter */
   new body(pos = (4.84143144246472090e+00, -1.16032004402742839e+00, -1.03622044471123109e-01),
       v = (1.66007664274403694e-03 * daysPerYear, 7.69901118419740425e-03 * daysPerYear, -6.90460016972063023e-05 * daysPerYear),
       mass = 9.54791938424326609e-04 * solarMass),

   /* saturn */
   new body(...),

   /* uranus */
   new body(...),

   /* neptune */
   new body(...) ]
```

5-body in Chapel: the Bodies

```chapel
var bodies =
    [ /* sun */
      new body(mass = solarMass),
    
    /* jupiter */
      new body(pos = ( 4.8414314424647209e+00,
                      -1.16032004402742839e+00,
                      -1.03622044471123109e-01),
      v = ( 1.66007664274403694e-03 * daysPerYear,
           7.69901118419740425e-03 * daysPerYear,
           -6.90460016972063023e-05 * daysPerYear),
      mass =  9.54791938424326609e-04 * solarMass),
    
    /* saturn */
      new body(...),

    /* uranus */
      new body(...),

    /* neptune */
      new body(...) ]
```
5-body in Chapel: the Bodies

```chapel
var bodies =
[   /* sun */
    new body(mass = solarMass),

   /* jupiter */
    new body(pos = ( 4.84143144246472090e+00,
                    -1.16032004402742839e+00,
                    -1.03622044471123109e-01),
        v = ( 1.66007664274403694e-03 * daysPerYear,
             7.69901118419740425e-03 * daysPerYear,
             -6.90460016972063023e-05 * daysPerYear),
        mass = 9.54791938424326609e-04 * solarMass),

    /* saturn */
    new body(...),

    /* uranus */
    new body(...),

    /* neptune */
    new body(...) ]
```
5-body in Chapel: the Bodies

```chapel
var bodies =
[ /* sun */
    new body(mass = solarMass),

    /* jupiter */
    new body(pos = ( 4.84143144246472090e+00,
    -1.16032004402742839e+00,
    -1.03622044471123109e-01),
    v = ( 1.66007664274403694e-03 * daysPerYear,
    7.69901118419740425e-03 * daysPerYear,
    -6.90460016972063023e-05 * daysPerYear),
    mass = 9.54791938424326609e-04 * solarMass),

    /* saturn */
    new body(...),

    /* uranus */
    new body(...),

    /* neptune */
    new body(...) ]
```

Creating a Record

Tuples

Array

Copyright 2016 Cray Inc.
Array Types

● Syntax

array-type:
  [ domain-expr ] elt-type
array-value:
  [elt1, elt2, elt3, ... eltn]

● Meaning:
  ● array-type: stores an element of elt-type for each index
  ● array-value: represent the array with these values

● Examples

  \texttt{var A: [1..3] int,} \quad \text{\texttt{// A stores 0, 0, 0}}
  \texttt{B = [5, 3, 9],} \quad \text{\texttt{// B stores 5, 3, 9}}
  \texttt{C: [1..m, 1..n] real,} \quad \text{\texttt{// 2D m by n array of reals}}
  \texttt{D: [1..m][1..n] real;} \quad \text{\texttt{// array of arrays of reals}}

\textit{Much more on arrays in data parallelism section later…}
5-body in Chapel: the Bodies

```chapel
var bodies =
[ /* sun */
  new body(mass = solarMass),

  /* jupiter */
  new body(pos = ( 4.84143144246472090e+00,
                  -1.16032004402742839e+00,
                  -1.03622044471123109e-01),
       v = ( 1.66007664274403694e-03 * daysPerYear,
            7.69901118419740425e-03 * daysPerYear,
            -6.90460016972063023e-05 * daysPerYear),
       mass = 9.54791938424326609e-04 * solarMass),

  /* saturn */
  new body(...),

  /* uranus */
  new body(...),

  /* neptune */
  new body(...) ]
```

Creating a Record

Tuples

Array
5-body in Chapel: main()

```chapel
proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
    writef("%.9r\n", energy());
}
```

...
5-body in Chapel: main()

```chapel
proc main() {
    initSun();

    writef("%.9r\n", energy());
    for 1..numsteps do
        advance(0.01);
        writef("%.9r\n", energy());
    }
}
```

- **Procedure Definition**
- **Procedure Call**
- **Formatted I/O** *(not covered here)*
- **Looping over a Range**
proc main() {
    initSun();
    writeln("%.9r\n", energy());
    for 1..numsteps do
        advance(0.01);
        writeln("%.9r\n", energy());
}

...
5-body in Chapel: main()

```chapel
proc main() {
    initSun();

    writeln("%.9r\n", energy());
    for 1..numsteps do
        advance(0.01);
        writeln("%.9r\n", energy());
}
```

Range Value
Range Values

- **Syntax**

  \[
  \text{range-expr:} \\
  \quad [\text{low}] .. [\text{high}]
  \]

- **Semantics**

  - Regular sequence of integers
    - \( \text{low} \leq \text{high} \): \( \text{low}, \text{low}+1, \text{low}+2, \ldots, \text{high} \)
    - \( \text{low} > \text{high} \): degenerate (an empty range)
    - \( \text{low} \) or \( \text{high} \) unspecified: unbounded in that direction

- **Examples**

  1..6  // 1, 2, 3, 4, 5, 6
  6..1  // empty
  3..   // 3, 4, 5, 6, 7, ...
Range Operators

```prolog
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
  for i in r do
    write(i, " ");
    writeln();
}
```

1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 ... n-1
... proc main() {
  initSun();
  writeln("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
    writeln("%.9r\n", energy());
}
...
For Loops

- **Syntax:**

```plaintext
for-loop:
    for [index-expr in] iterable-expr { stmt-list }
```

- **Meaning:**
  - Executes loop body serially, once per loop iteration
  - Declares new variables for identifiers in `index-expr`
    - type and const-ness determined by `iteratable-expr`
    - `iteratable-expr` could be a range, array, or iterator

- **Examples**

```plaintext
var A: [1..3] string = [" DO", " RE", " MI"];

for i in 1..3 { write(A[i]); }    // DO RE MI
for a in A { a += "LA"; } write(A);  // DOLA RELA MILA
```
5-body in Chapel: main()

```chapel
proc main() {
    initSun();
    writef("%.9r\n", energy());
    for 1..numsteps do
        advance(0.01);
        writef("%.9r\n", energy());
}
```

- **Function Declaration**
- **Function Call**
- **Formatted I/O**
  - *not covered here*
- **Looping over a Range**
5-body in Chapel: advance()

```chapel
advance(0.01);
...
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
      mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }
}

for b in bodies do
  b.pos += dt * b.v;
```
advance(0.01);
...
proc advance(dt) {
    for i in 1..numbodies {
        for j in i+1..numbodies {
            const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

            bodies[i].v -= dpos * bodies[j].mass * mag;
            bodies[j].v += dpos * bodies[i].mass * mag;
        }
    }

    for b in bodies do
        b.pos += dt * b.v;
}
5-body in Chapel: advance()

```chapel
advance(0.01);  
...
proc advance(dt) {
    for i in 1..numbodies {
        for j in i+1..numbodies {
            const dpos = bodies[i].pos - bodies[j].pos,
                       mag = dt / sqrt(sumOfSquares(dpos))**3;
            bodies[i].v -= dpos * bodies[j].mass * mag;
            bodies[j].v += dpos * bodies[i].mass * mag;
        }
    }

    for b in bodies do
        b.pos += dt * b.v;
}
```
Procedures, by example

● Example to compute the area of a circle

```plaintext
proc area(radius: real): real {
    return 3.14 * radius**2;
}
writeln(area(2.0)); // 12.56
```

● Example of argument default values, naming

```plaintext
proc writeCoord(x: real = 0.0, y: real = 0.0) {
    writeln((x,y));
}
writeCoord(2.0); // (2.0, 0.0)
writeCoord(y=2.0); // (0.0, 2.0)
writeCoord(y=2.0, 3.0); // (3.0, 2.0)
```

Argument and return types can be omitted.
5-body in Chapel: Using Iterators

```chapel
iter triangle(n) {
    for i in 1..n do
        for j in i+1..n do
            yield (i,j);
}

proc advance(dt) {
    for (i,j) in triangle(numbodies) {
        const dpos = bodies[i].pos - bodies[j].pos,
        mag = dt / sqrt(sumOfSquares(dpos))**3;
        ...
    }
    ...
}
```
Additional Base Language Notes / Material
5-body in Chapel: advance() using references

```chapel
proc advance(dt) {
    for i in 1..numbodies {
        for j in i+1..numbodies {
            ref bi = bodies[i],
            bj = bodies[j];

            const dpos = bi.pos - bj.pos,
                        mag = dt / sqrt(sumOfSquares(dpos))**3;

            bi.v -= dpos * bj.mass * mag;
            bj.v += dpos * bi.mass * mag;
        }
    }

    for b in bodies do
        b.pos += dt * b.v;
}
```
Reference Declarations

● Syntax:

```
ref-decl:
  ref ident = expr;
```

● Meaning:
  ● Causes ‘ident’ to refer to variable specified by ‘expr’
  ● Subsequent reads/writes of ‘ident’ refer to that variable
  ● Not a pointer: no way to reference something else with ‘ident’
  ● Similar to a C++ reference

● Examples

```
var A: [1..3] string = [" DO", " RE", " MI"];  
ref a2 = A[2];
a2 = " YO";
for i in 1..3 { write(A[i]); }  // DO YO MI
```
## Primitive Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
<th>Currently-Supported Bit Widths</th>
<th>Default Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>false</td>
<td>8, 16, 32, 64</td>
<td>impl. dep.</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>64</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>64</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>complex</td>
<td>complex floating points</td>
<td>0.0 + 0.0i</td>
<td>64, 128</td>
<td>128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>&quot;&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Syntax**

```
primitive-type: type-name [( bit-width )]
```

**Examples**

```
int(16)  // 16-bit int
real(32) // 32-bit real
uint     // 64-bit uint
```
Enum Types

- A lot like enum types in C:
  ```
  enum color {red, green, blue};  // can also be assigned values
  ```
- But can also be printed!
  ```
  var myColor = color.red;
  writeln(myColor);  // prints ‘red’
  ```
- And support built-in iterators and queries:
  ```
  for c in color do ...
  ...color.size...
  ```

- By default, must be fully-qualified to avoid conflicts:
  ```
  var myColor = red;  // error by default
  ```
- But, may be ‘use’d like modules to avoid qualifying
  ```
  use color;  // can use standard filters, renaming, etc.
  var myColor = red;  // OK!
  ```
Type Aliases and Casts

● Basic Syntax

\[
type-alias-declaration: \\
\quad type \ identifier = type-expr; \\
\]

\[
cast-expr: \\
\quad expr : type-expr \\
\]

● Semantics

- type aliases are simply symbolic names for types
- casts are supported between any primitive types

● Examples

\[
\text{type} \ \text{elementType} = \text{complex}(64); \\
5:\text{int}(8) \quad // \text{store value as int}(8) \text{ rather than int}\n\"54\":\text{int} \quad // \text{convert string to an int}\n249:elementType \quad // \text{convert int to complex}(64) \\
\]
Basic Operators and Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
<th>Overloadable</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>cast</td>
<td>left</td>
<td>no</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>! ~</td>
<td>logical and bitwise negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division and modulus</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>unary + −</td>
<td>positive identity and negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>+ −</td>
<td>addition and subtraction</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>shift left and shift right</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;= &gt;= &lt; &gt;</td>
<td>ordered comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>== !=</td>
<td>equality comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise/logical and</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>^</td>
<td>bitwise/logical xor</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>l</td>
<td>bitwise/logical or</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short-circuiting logical and</td>
<td>left</td>
<td>via isTrue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short-circuiting logical or</td>
</tr>
</tbody>
</table>
Control Flow: Braces vs. Keywords

Control flow statements specify bodies using curly brackets (compound statements)

- Conditional statements
  ```plaintext
  if cond { computeA(); } else { computeB(); }
  ```

- While loops
  ```plaintext
  while cond {
    compute();
  }
  ```

- For loops
  ```plaintext
  for indices in iterable-expr {
    compute();
  }
  ```

- Select statements
  ```plaintext
  select key {
    when value1 { compute1(); }
    when value2 { compute2(); }
    otherwise { compute3(); }
  }
  ```
They also support keyword-based forms for single-statement cases

- **Conditional statements**

```plaintext
if cond then computeA(); else computeB();
```

- **While loops**

```plaintext
while cond do
  compute();
```

- **For loops**

```plaintext
for indices in iterable-expr do
  compute();
```

- **Select statements**

```plaintext
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise do compute3();
}
```
Control Flow: Braces vs. Keywords

Of course, since compound statements are single statements, the two forms can be mixed...

- **Conditional statements**
  ```
  if cond then { computeA(); } else { computeB(); }
  ```

- **While loops**
  ```
  while cond do {
    compute();
  }
  ```

- **Select statements**
  ```
  select key {
    when value1 do { compute1(); }
    when value2 do { compute2(); }
    otherwise do { compute3(); }
  }
  ```

- **For loops**
  ```
  for indices in iterable-expr do {
    compute();
  }
  ```

Copyright 2016 Cray Inc.
Procedures and iterator features

- **pass by keyword (argument name)**
  ```
  proc foo(name, age) { ... }
  foo(age=32, name="Tim");
  ```

- **default argument values**
  ```
  proc foo(name, age=18) { ... }
  foo(name="Tim");
  ```

- **formal type queries**
  ```
  proc foo(x: ?t, y: [?D] t) { ... }
  proc bar(x: int(?w)) { ... }
  ```

- **overloading**
  - including where clauses to filter overloads
    ```
    proc foo(x: int(?w), y: int(?w2)) where w = 2*w2 { ... }
    proc foo(x: int(?w), y: int(?w2)) { ... }
    proc foo(x, y) { ... }
    ```
Methods

Methods are like procedures with an implicit

- Chapel supports both *primary methods*:
  
  ```chapel
  class circle {
    proc area() { return pi*radius**2; }
  }
  ```

- and *secondary methods*:
  ```chapel
  proc circle.circumference() {
    return 2*pi*radius;
  }
  ```

- Moreover, secondary methods can be defined for any type:
  ```chapel
  proc int.square() {
    return this**2;
  }
  ```

  ```chapel
  writeln(5.square());  // prints 25
  ```
Paren-less procedures

Procedures without arguments don’t need parenthesis

```plaintext
proc circle.diameter {
  return 2*radius;
}

writeln(cl.radius, " ", cl.diameter);
```

Support time/space tradeoffs without code changes

- Store value with variable/field?
- Or compute on-the-fly with paren-less procedure/method?
  - Like fields, such methods don’t dispatch dynamically
Function Calls vs. Array Accesses

- Chapel doesn’t distinguish between call and array access
  - An “array access” is simply a call to a special method named “this()”
    ```chapel
    class circle {
        proc this(x: int, y: real) {
            // do whatever we want here...
        }
    }
    myCircle[2, 4.2]; // calls circle.this()
    
    Related: parens/square brackets can be used for either case:
    ```
    ```chapel
    A[i, j] or A(i, j) // these are both accesses to array A
    foo() or foo[] // these are both function calls to foo()
    ```
  - By convention, we tend to use [] for arrays and () for function calls
  - but Fortran programmers may be happy to get to use () for arrays…?
  - Like paren-less methods, view this as another time-space tradeoff
    - can implement something as a function or as an array
    - since Chapel’s arrays are quite rich, access is not necessarily O(1) anyway
Default object iterators

- Objects can support default iterators
  ```java
class circle {
    iter these() {
        // yield whatever we want…
    }
}
for items in myCircle do … // invokes circle.these()
```

- Similar to the ‘this()’ default accessor
- Overloads can support parallel or parallel zippered iteration
  - (true for any iterator)
Generic procedures can be defined using type and param arguments:

```plaintext
proc foo(type t, x: t) { ... }
proc bar(param bitWidth, x: int(bitWidth)) { ... }
```

Or by simply omitting an argument type (or type part):

```plaintext
proc goo(x, y) { ... }
proc sort(A: []) { ... }
```

Generic procedures are instantiated for each unique argument signature:

```plaintext
foo(int, 3);  // creates foo(x:int)
foo(string, "hi");  // creates foo(x:string)
goo(4, 2.2);  // creates goo(x:int, y:real)
```
Generic Objects

Generic objects can be defined using type and param fields:

```plain
record Table { param size: int; var data: size*int; } record Matrix { type eltType; ... }
```

Or by simply eliding a field type (or type part):

```plain
class Triple { var x, y, z; }
```

Generic objects are instantiated for each unique type signature:

```plain
// instantiates Table, storing data as a 10-tuple
var myT: Table(10);
// instantiates Triple as x:int, y:int, z:real
var my3: Triple(int, int, real) = new Triple(1, 2, 3.0);
```
Modules

- **Syntax**

\[
\text{module-def:} \\
\quad \text{module identifier \{ code \}} \\
\text{module-use:} \\
\quad \text{use module-identifier;}
\]

- **Semantics**

- all Chapel code is stored in modules
- *use*-ing a module makes its symbols visible in that scope
- module-level statements are executed at program startup
  - typically used to initialize the module
- for convenience, a file containing code outside of a module declaration creates a module with the file’s name
Use Statement: Import Control

Use statements support import control

- ‘except’ keyword prevents unqualified access to symbols in list
  ```
  use M except bar;  // All of M’s symbols other than bar can be named directly
  ```
- ‘only’ keyword limits unqualified access to symbols in list
  ```
  use M only foo;    // Only M’s foo can be named directly
  ```
- Permits user to avoid importing unnecessary symbols
  - Including symbols which cause conflicts

```module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo;
    foo();
    var a = bar;  // Now finds myMod.bar, rather than M.bar
  }
}
module M {
  var bar = 13;
  proc foo() { ... }
}```
Use Statement: Symbol Renaming

● Use’d symbols can also be renamed:
  
  use M only bar as barM;

● Allows users to avoid...
  
  ...naming conflicts between multiple used modules
  ...shadowing outer variables with same name
  ...while still making that symbol available for access

module myMod {
  var bar = true;

  proc myFunc() {
    use M only foo, bar as barM;
    foo();
    var a = bar;    // Still finds myMod.bar, rather than M.bar
    var b = barM;   // refers to M.bar
  }
}

module M {
  var bar = 13;
  proc foo() { ... }
}
Modules: Public/Private Declarations

- All module-level symbols are public by default
  ```
  proc foo() { ... }  // public, since not decorated
  ```

- Module-level symbols can be declared public/private:
  ```
  private var bar = ...;
  public proc baz() { ... }
  ```

- Can be used in declarations of:
  - Modules
  - Vars, consts, and params
    - including configs
  - Procedures and iterators

- Future work: extend to other symbols
  - Particularly types / object members
Program Entry Point: main()

- **Semantics**
  - Chapel programs start by:
    - initializing all modules
    - executing main(), if it exists

```chapel
M1.chpl:
use M2;
writeln("Initializing M1");
proc main() { writeln("Running M1"); }

M2.chpl:
module M2 {
  writeln("Initializing M2");
}
```

```
% chpl M1.chpl M2.chpl
% ./a.out
Initializing M2
Initializing M1
Running M1
```
Argument and Return Intents

- **Arguments can optionally be given intents**
  - (blank): varies with type; follows principle of least surprise
    - most types: `const`
    - arrays, domains, sync vars, atomics: passed by reference
  - `in`: copies actual into formal; permits changes
  - `out`: copies formal into actual at procedure return
  - `inout`: does both of the above
  - `ref`: pass by reference
  - `const [ref | in]`: disallows modification of the formal
  - `param/type`: formal must be a param/type (evaluated at compile-time)

- **Return types can also have intents**
  - (blank)/`const`: cannot be modified (without assigning to a variable)
  - `ref`: permits modification back at the callsite
  - `type`: returns a type (evaluated at compile-time)
  - `param`: returns a param value (evaluated at compile-time)
Other Base Language Features not covered here

- Interoperability with external code
- Compile-time features for meta-programming
  - type/param procedures
  - folded conditionals
  - unrolled for loops
  - user-defined compile-time warnings and errors
- Type select statements, argument type queries
- Unions
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.