Visibility Control:

Use and Import Statement Improvements

Lydia Duncan

CHIUW 2020

May 22, 2020

lydia.duncan@hpe.com

chapel-lang.org

@ChapelLanguage

CRAY®
a Hewlett Packard Enterprise company

CRAY CHAPEL

Outline

- Introduction
- Transitivity
- Correctness and Compilation Speed
- Renaming
- Qualified Access and ‘import’
- What’s Next?

Introduction

Introduction

- The Chapel team at Cray/HPE is working towards Chapel 2.0
    - This means determining which language features are likely to be stable

- ‘use’ statements have been part of the language for a long time
    - Enable symbols in one module to be visible in another module
        - Either with *unqualified* access (no module prefix)…

          … or *qualified* access (with the module prefix)

```
use M;
writeln(x);     // prints the value of M.x
writeln(M.y);   // prints the value of M.y
```

Introduction

- Chapel 1.12 / 1.13 improved support for privacy and namespace control, e.g.
  - Added ‘public’ and ‘private’ designators for symbols
  - Added ‘only’ and ‘except’ clauses to ‘use’ statements
    - These limit the symbols brought in for unqualified access
  - These changes were presented at CHUIW 2016

- But we also had several extensions and changes planned that weren’t done
  - Some of these changes would break backwards compatibility

- This talk will cover recent changes, as well as some forward-looking features

Transitivity

Transitivity

- To motivate some of these changes to ‘use’, we need to talk about transitivity
  - Prior to Chapel 1.20, ‘use’ statements were always ‘public’
  - This meant that symbols brought in were made more broadly available

```
module B { use A; … }
module C {
  …
  proc bar() {
    use B;
    writeln(x); // ‘x’ is defined by module ‘A’, but ‘C’ didn’t ‘use A’ itself
  }
}
```

Transitivity

- This was a problem
  - Required increased care when naming symbols…

```
module B { use A; … }
module C {
  var x = 3;
  proc bar(){
    use B;
    writeln(x); // ‘x’ is defined by module ‘A’, so this won’t necessarily print 3!
  }
}
```

Transitivity

- This was a problem
  - Could lead to hijacking if a library you rely on changed its underlying definition
    - Or what modules it relied upon…
    - Or even if modules it relied upon changed!

  - Could also lead to compilation errors when the symbols would conflict

  - Meant that users might rely on implementation details
    - Good language design should give library writers control over what is seen

Transitivity

- Could work around this by limiting the scope of the ‘use’ statement
  - E.g. by putting the ‘use’ inside a function body:

```
module B {
  proc foo() {
    use A;
  }
}
```

  - But this wasn’t always feasible
    - If module is integral to your program, will dramatically increase # of ‘use’s
    - If you need the module due to an argument type, you are out of luck

Transitivity

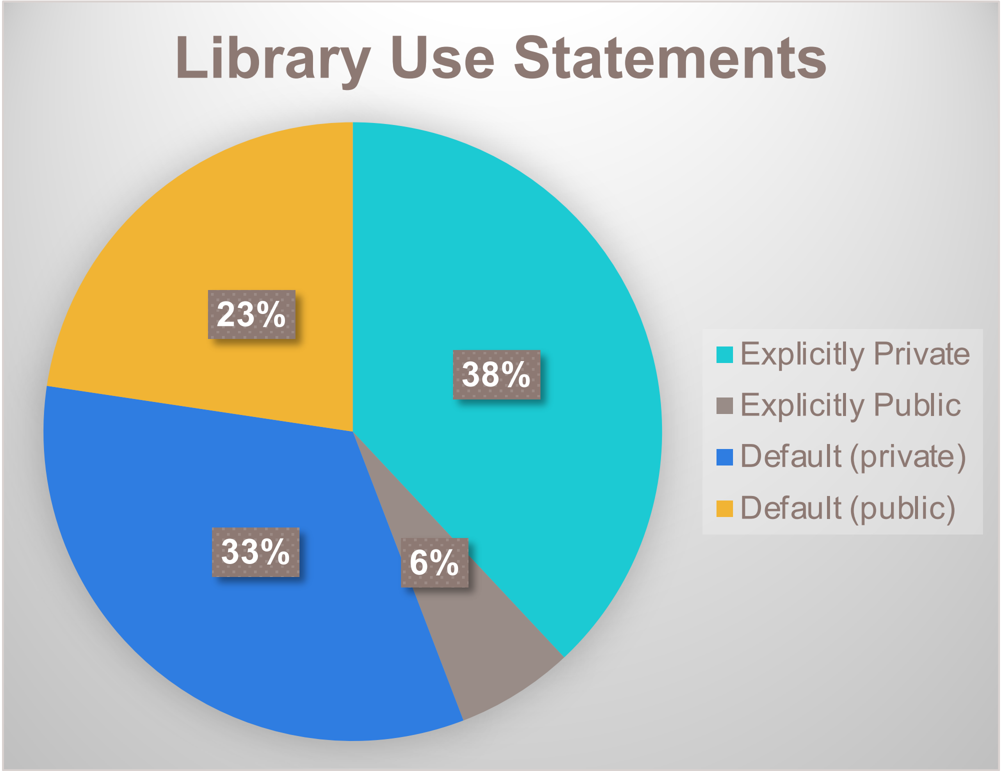
- We added ‘public’ and ‘private’ specifiers to ‘use’ statements

```
private use A;
```

  - This allows library writers to intentionally choose which ‘use’s are visible
  - We also switched the default to ‘private’
    - This could result in some broken code, but the fixes should be simple
    - And default code will be safer going forward

Transitivity: Impact on Libraries

- All default ‘use’ statements in the standard and package libraries are now ‘private’
- All default ‘use’ statements in the internal libraries are still ‘public’
  - Want to make some of these ‘private’, too, but it’s not trivial
- Many ‘use’ statements are now either explicitly ‘public’ or ‘private’



Transitivity: Impact on Libraries

- As a result, some modules are no longer available by default to user programs
  - RangeChunk, SysCTypes, CommDiagnostics now require an explicit ‘use’
    - Some of these were accidentally included before (e.g. CommDiagnostics)
    - Others we knew had been getting included, but didn’t want to still do so
  - These modules were not used in the common case
    - So not including them by default makes sense

Transitivity: Impact on Libraries

- And default-included symbols no longer take precedence over outer-scoped
  - Here’s an example of when that was a problem:

```
var e = 17;
{
  use Mod;
  // Used to print Math.e because of the default ‘public use’ of Math by Mod
  // Now, the default ‘use’ is ‘private’, so it prints ‘17’
  writeln(e);
}
```

Transitivity

- Transitive ‘use’ statements are powerful, but often have broad consequences
- Giving users control over the transitivity of their ‘use’ statements is valuable
  - Users have better knowledge of what is appropriate for their code

- Changing the default transitivity makes code safer
  - Users must actively choose to make a ‘use’ transitive
  - Therefore, they are more likely to understand what doing so means

- And limiting the transitivity of library ‘use’s improves the user experience
  - It reduces the potential for namespace confusion

Correctness and Compilation Speed

Or:
Why Is My Compilation Slower? An Apology

Correctness and Compilation Speed

- Function resolution in the compiler had an “optimization”, standardModuleSet
  - Had been in the compiler since the Dawn of Time*
  - Basically, treated every module used by default as though it was in one scope
  - This made it easy to resolve default symbols
    - Too easy…

*The compiler has not been around since the Dawn of Time

Correctness and Compilation Speed

- This “optimization” assumed everything was visible everywhere
  - As a result, some internal modules were accessing modules they didn’t use
    - And that weren’t transitively available to them, either:

```
module ChapelBase {
  // needed ‘private use ChapelEnv;’ to access ‘CHPL_NETWORK_ATOMICS’
  …
  config param useAtomicTaskCnt =
                CHPL_NETWORK_ATOMICS != "none";
}
```

  - There were many other examples of bad behavior enabled by it

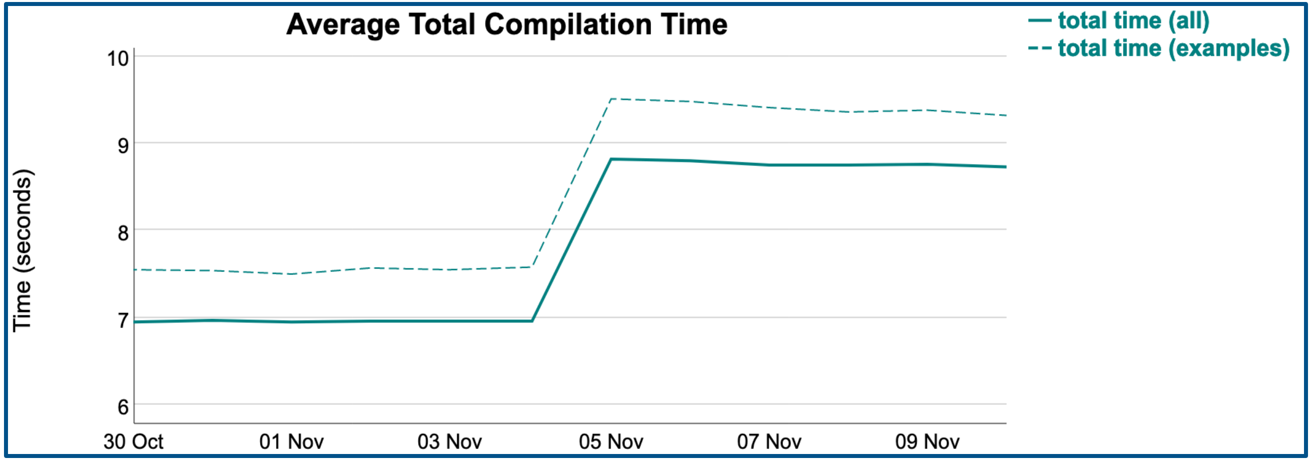
Correctness and Compilation Speed

- There wasn’t a way to reconcile this “optimization” and ‘private’ at all
  - ‘private’ depends on the module hierarchy being maintained
    - Both for ‘private use’ and ‘private’ symbols
  - standardModuleSet removes that hierarchy entirely

- It enabled a lot of bugs
- And made the internal modules harder to maintain as a result

Correctness and Compilation Speed

- So we removed the standardModuleSet …

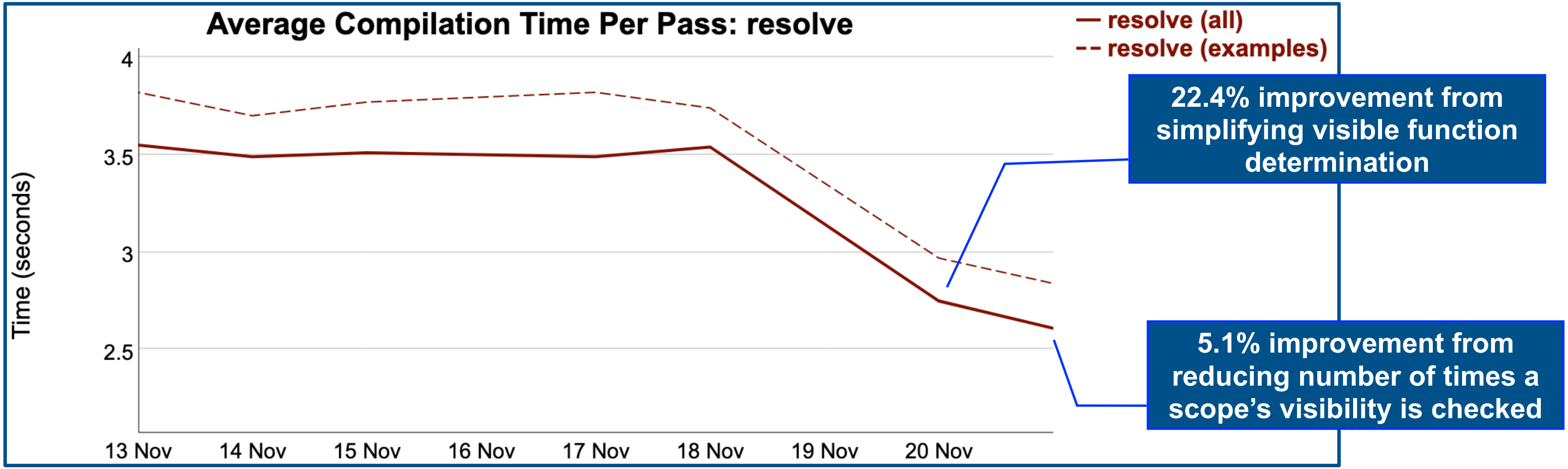


… resulting in an average slowdown of 26.7% for our testing suite as a whole …

… and roughly 37% for arkouda!

Correctness and Compilation Speed

- Still, removing the “optimization” was the right thing to do
- So we set about looking at ways to mitigate this impact
  - Mostly by improving parts of function resolution



22.4% improvement from simplifying visible function determination

5.1% improvement from reducing number of times a scope’s visibility is checked

Correctness and Compilation Speed

- Ultimately, compilation is still slower than it was
  - But most of the impact from this change has been recovered
  - We're hoping to work more on compilation in this release cycle
- The default libraries are more accurate and less tangled than they were before
  - Though work still needs to be done to disentangle them further

Renaming

Renaming

- ‘use’ statements can limit the symbols brought in to ‘only’ specific symbols

```
use Mod only veryLongName;
```

- When ‘only’ lists were added, they also enabled symbol renaming

```
use Mod only veryLongName as vln;
writeln(vln);  // Prints value of ‘Mod.veryLongName’
```

- This allowed users to avoid:
  - conflicts with symbols brought in from other modules,
  - shadowing symbols at outer scopes that share the same name,
  - and having to type long descriptive names repeatedly.

Renaming

- As a side effect, we could rename submodules when using their parent module

```
        use Mod only InnerMod as IM;
```

- We decided to extend this to enable renaming when the module itself is used…

```
        use Mod.InnerMod as IM;
```

… which allowed top-level modules to be renamed for the first time

```
        use Mod as M;
```

Qualified Access and ‘import’

Import Statements

- ‘use’ statements have been imprecise
  - Default behavior brought every visible symbol into scope
    - However, could limit the symbols brought in with ‘except’ and ‘only’ lists
  - Design focused on “programming in the small” scenarios

- Users desired a feature for more precise access of module symbols
  - One better suited for maintaining large-scale software
  - Ideally, without breaking current code

Import Statements: This Effort

- We designed and implemented the ‘import’ statement as an alternative to ‘use’
  - Simplest form enables qualified access to the symbols in a module:

```
import MyModule;
writeln(MyModule.sym1); // Enabled by the ‘import’
writeln(sym1);          // Not enabled, won’t work
```

  - This was previously only achievable with “empty” ‘use’ statements, e.g.

```
use MyModule only;
use MyModule except *;
```

Import Statements: Accessing Module Contents

- Can also enable unqualified access to a single symbol within a module:

```
import MyModule.sym1;
writeln(sym1);             // Enabled by the 'import'
writeln(MyModule.sym1);    // Not enabled by the 'import'
```

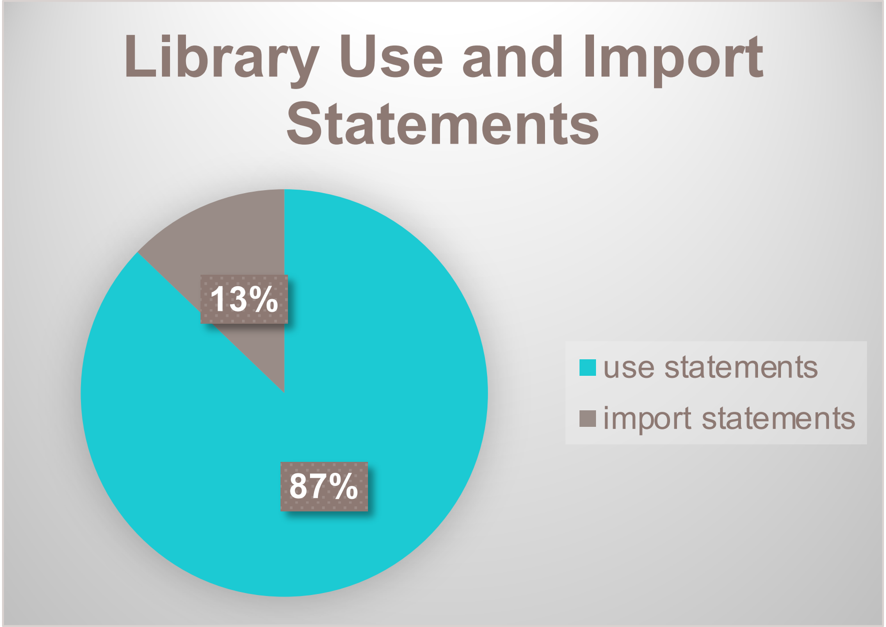
- Or multiple symbols within a module:

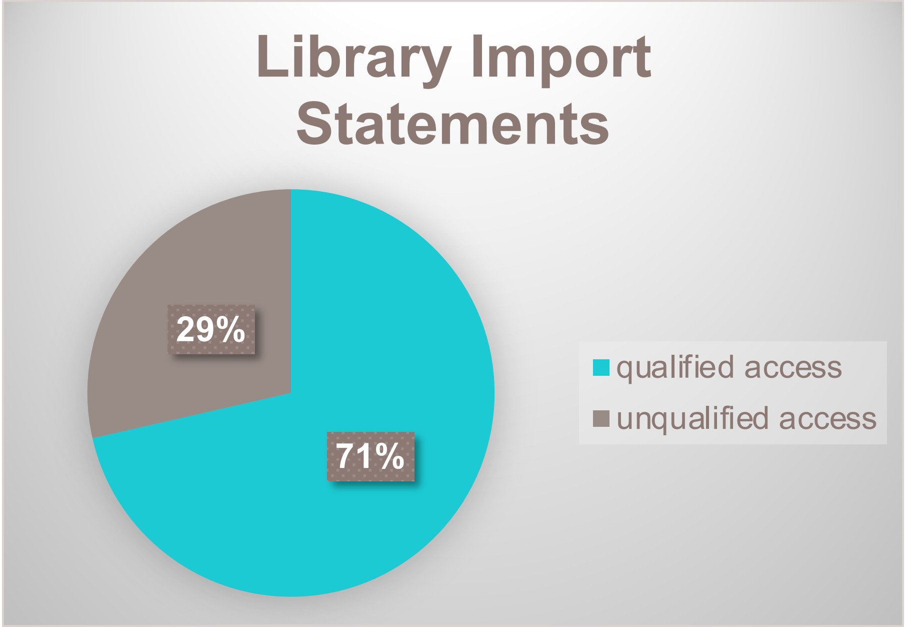
```
import MyModule.{sym1, sym2, sym3};
```

- Neither of these options was available previously
  - 'use' statements always enabled qualified access in addition to unqualified

Import Statements: Impact on Libraries

- We replaced all “empty” ‘use’ statements in the libraries with ‘import’ statements
- And are starting to use other variants, too





Import Statements: Renaming

- Modules that are imported can be renamed:

```
import MyModule as Foo;
writeln(Foo.sym1);         // Enabled by the ‘import’
writeln(sym1);             // Not enabled by the ‘import’
writeln(MyModule.sym1);    // Not enabled by the ‘import’
```

- As can symbols that are imported for unqualified access:

```
import MyModule.sym1 as x;  // or:
import MyModule.{sym1 as x, sym2 as y};
```

Import Statements: Nested Modules

- Nested modules must be named using their parent modules…

```
module OuterMod {
  import InnerMod;                // error: looks for top-level module 'InnerMod'
  import OuterMod.InnerMod;       // OK: names module starting from top-level
  writeln(InnerMod1.sym1);
  module InnerMod { var sym1 = …; }
}
```

- Unlike ‘use’ statements, ‘import’ statements can’t use relative naming
  - E.g. ‘OuterMod’ can’t just write ‘import InnerMod;’

Import Statements: Nested Modules

- Nested modules can be named directly in certain circumstances:
  - E.g. after being made available by another ‘import’ or ‘use’

```
use OuterMod;        // makes ‘OuterMod’s symbols available
import InnerMod;     // ‘InnerMod’ visible due to ‘use OuterMod’
```

- Both ‘use’ and ‘import’ can shorten the path with ‘this’ if within a parent module…

```
module OuterMod {
  module InnerMod { … }
  import this.InnerMod; // Enabled by being within ‘OuterMod’
}
```

Import Statements: Nested Modules

- Nested modules can also be imported using ‘super’ if within a sibling module
  - Like ‘this’, ‘super’ also works for ‘use’ statements

```
module OuterMod {
  module InnerMod { … }
  module SiblingMod {
    // Enabled by being within OuterMod.SiblingMod
    import super.InnerMod;
  }
}
```

Import Statements: Nested Modules

- Using ‘this’/‘super’ makes ‘use’ and ‘import’ safer than using relative names
  - Origin of relatively used modules is much more obvious to the reader

  - This style of ‘use’ makes code more robust to later changes
    - If dependency defines another module with same name, won’t conflict

Import Statements: Public / Private

- ‘import’ statements can be declared ‘public’ or ‘private’
  - Default is ‘private’
    - as with ‘use’, reduces unintentional leaking of names
  - ‘public’ means symbols brought in are *re-exported*

```
module Mod {
  public import OtherMod;
}
module ThirdMod {
  import Mod.OtherMod; // ‘OtherMod’ acts like a submodule of ‘Mod’
}
```

Import Statements: Impact

- The ‘import’ statement supports module access in a more precise manner
  - Its default behavior minimally extends the scope

- It also enables new functionality:
  - Can re-export symbols
  - Can bring symbols in for unqualified access without enabling qualified access

What’s Next?

What’s Next?

- Extend ‘import’ to support multiple expressions in a single statement

```
import Mod1.{a, b}, Mod2.{x, y}; // Should this be allowed?
```

  - See issue #14971 and #15583

- Enable re-exporting for ‘use’ statements
  - See issue #15282

- Implement ability to ‘use’ module and disable qualified access (issue #15457)

```
use Mod as _;
writeln(Mod.x); // Wouldn’t work, not enabled by this ‘use’
writeln(x);     // Would still work, enabled by this ‘use’
```

What’s Next?

- Design story for ‘private’ fields/methods and types
  - See issue #6067

- Continue reviewing the set of symbols made available by default

- Continue improving ‘use’ statements within internal modules

Acknowledgements

Special thanks to Brad Chamberlain, Michael Ferguson, and Engin Kayraklioglu for contributing to the implementation, and to the rest of the Chapel team (past and present) and Bryant Lam for contributing to the design of this work.

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?

✉ lydia.duncan@hpe.com

@ChapelLanguage

chapel-lang.org

CRAY®

a Hewlett Packard Enterprise company

cray.com

@cray_inc

linkedin.com/company/cray-inc-/ in