Trees:

- There are many kinds of tree data structure.
- For example:

  ```haskell
data BinTree a  = Leaf a 
                   | BinTree a :^: BinTree a 
       deriving Show
```

- The “deriving Show” part makes it possible for us to print out tree values ...

Wouldn’t it be nice ... ...

If we could view these trees in a graphical form

![Diagram of a tree]

Mapping on Trees:

- We can define a mapping operation on trees:

  ```haskell
mapTree :: (a -> b) -> BinTree a -> BinTree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (l :^: r) = mapTree f l :^: mapTree f r
```

- This is an analog of the map function on lists; it applies the function f to each leaf value stored in the tree.

Example: convert every leaf value into a string:

```haskell
Main> mapTree show example
((Leaf "1" :^: (Leaf "5" :^: Leaf "6")):^: ((Leaf "3" :^: Leaf "4"):^: (Leaf 3 :^: Leaf 4) :^: (Leaf 5 :^: Leaf 6)))
```

Example: add one to every leaf value:

```haskell
Main> mapTree (+1) example
```

Still not very pretty ...
Visualizing the Results:
If we could view these trees in a graphical form ...

Graphviz & Dot:
- Graphviz is a set of tools for visualizing graph and tree structures.
- Dot is the language that Graphviz uses for describing the tree/graph structures to be visualized.
- Usage: `dot -Tpng file.dot > file.png`

General Form:
A dot file contains a description of the form `digraph name { stmts }` where each `stmt` is either...

Example:
- To describe (Leaf "a" :^: Leaf "b" :^: Leaf "c"):
  ```
  digraph tree {
      "1" [label=""];
      "1" -> "2";
      "2" [label=""];
      "2" -> "3";
      "3" [label="a"];  
      "2" -> "4";
      "4" [label="b"];  
      "1" -> "5";
      "5" [label="c"];  
  }
  ```
From BinTree to dot:

How can we convert a BinTree value into a dot file?

For simplicity, assume a BinTree String input.

Labels:

- Label leaf nodes with the corresponding strings
- Label internal nodes with the empty string

Node ids:

- What should we use for node ids?

Paths:

Every node can be identified by a unique path:

- The root node of a tree has path []
- The n\textsuperscript{th} child of a node with path p has path (n:p)

```haskell
data Path = [Int]  

data NodeId = String

showPath :: Path -> NodeId
showPath p = "" ++ show p ++ ""
```

Example:

```
actual dot code:

To describe (Leaf "a" :^: Leaf "b" :^: Leaf "c"):

digraph tree {
"[]" [label=""];
"[]" -> "[1]";
"[1]" [label=""];
"[1]" -> "[1,1]";
"[1,1]" [label="a"];
"[1]" -> "[2,1]";
"[2,1]" [label="b"];
"[]" -> "[2]";
"[2]" [label="c"];}
```

Capturing “Tree”-ness:

```haskell
subtrees :: BinTree a -> [BinTree a]
subtrees (Leaf x) = []
subtrees (l :^: r) = [l, r]

nodeLabel :: BinTree String -> String
nodeLabel (Leaf x) = x
nodeLabel (l :": r) = ""
```

Trees -> dot Statements:

```haskell
nodeTree :: Path -> BinTree String -> [String]
nodeTree p t
  = [ showPath p ++ " [label=\"" ++ nodeLabel t ++ \"\"]" ]
    ++ concat (zipWith (edgeTree p) [1..] (subtrees t))

edgeTree :: Path -> Int -> BinTree String -> [String]
edgeTree p n c
  = [ showPath p ++ " -> " ++ showPath p' ]
     ++ nodeTree p' c
  where p' = n : p
```
A Top-level Converter:

```haskell
toDot :: BinTree String -> IO ()
toDot t = writeFile "tree.dot"
  "digraph tree \n  ++ semi (nodeTree [] t)
  ++ ")\n")"
where semi = foldr (\ls -> l ++ ";\n" ++ ls) ""

Now we can generate dot code for our example tree:

Main> toDot (mapTree show example)
Main> !dot -Tpng tree.dot > ex.png
Main>
```

What About Other Tree Types?

```haskell
data LabTree l a = Tip a |
                    LFork l (LabTree l a) (LabTree l a)
data STree a     = Empty |
                    Split a (STree a) (STree a)
data RoseTree a  = Node a [RoseTree a]
data Expr       = Var String |
                    IntLit Int |
                    Plus Expr Expr |
                    Mult Expr Expr

Can I also visualize these using Graphviz?
```

Higher-Order Functions:

Essential tree structure is captured using the `subtrees` and `nodeLabel` functions.

What if we abstract these out as parameters?

```haskell
nodeTree' :: (t -> String) ->
            (t -> [t]) ->
            Path -> t -> [String]
edgeTree' :: (t -> String) ->
            (t -> [t]) ->
            Path -> Int -> t -> [String]
```

Adding the Parameters:

```haskell
nodeTree' lab sub p t
  = [ showPath p ++ " [label=" ++ lab ++ "]\n" ]
  ++ concat (zipWith (edgeTree' lab sub p) [1..] (sub t))
edgeTree' lab sub p n c
  = [ showPath p ++ " -> " ++ showPath p' ]
  ++ nodeTree' lab sub p' c
  where p' = n : p
toDot' :: (t -> String) -> (t -> [t]) -> t -> IO ()
toDot' lab sub t
  = writeFile "tree.dot"
  "digraph tree \n  ++ semi (nodeTree' [] t) ++ "),\n")"
where semi = foldr (\ls -> l ++ ";\n" ++ ls) ""

Alternative (Local Definitions):

```haskell
toDot'' :: (t -> String) -> (t -> [t]) -> t -> IO ()
toDot'' lab sub t
  = writeFile "tree.dot"
  "digraph tree \n  ++ semi (nodeTree' [] t) ++ "),\n")"
where semi = foldr (\ls -> l ++ ";\n" ++ ls) ""

nodeTree' p t
  = [ showPath p ++ " [label=" ++ lab ++ "]\n" ]
  ++ concat (zipWith (edgeTree' p) [1..] (sub t))
enodeTree' p n c
  = [ showPath p ++ " -> " ++ showPath p' ]
  ++ nodeTree' p' c
  where p' = n : p
toDotBinTree = toDot' lab sub where
  lab (Leaf x) = x
  lab (l :+: r) = ""
  sub (Leaf x) = []
  sub (l :+: r) = [l, r]
toDotLabTree = toDot' lab sub where
  lab (Tip a)    = a
  lab (LFork s l r) = s
  sub (Tip a)    = []
  sub (LFork s l r) = [l, r]
toDotRoseTree = toDot' lab sub where
  lab (Node x cs) = x
  sub (Node x cs) = cs
```

Specializing to Tree Types:

```haskell
toDotBinTree = toDot' lab sub where
  lab (Leaf x) = x
  lab (l :+: r) = s
  sub (Leaf x) = []
  sub (l :+: r) = [l, r]
toDotLabTree = toDot' lab sub where
  lab (Tip a)    = a
  lab (LFork s l r) = s
  sub (Tip a)    = []
  sub (LFork s l r) = [l, r]
toDotRoseTree = toDot' lab sub where
  lab (Node x cs) = x
  sub (Node x cs) = cs
```
... continued:

toDotSTree = toDot' lab sub
where lab Empty = ""
        lab (Split s l r) = s
        lab (Split s l r) = [l, r]

toDotExpr = toDot' lab sub
where lab (Var s) = s
        lab (IntLit n) = show n
        lab (Plus l r) = "+"
        lab (Mult l r) = "*"
        lab (Var s) = []
        lab (IntLit n) = []
        lab (Plus l r) = [l, r]
        lab (Mult l r) = [l, r]

toDotRoseTree
(Node "a" [Node "b" [],
        Node "c" [],
        Node "d" [Node "e" []]])

toDotExpr (Plus (Mult (Var "x") (IntLit 3))
             (Mult (Var "y") (IntLit 5)))

Example:

Example:

Good and Bad:

Good:
- It works!
- It is general (applies to multiple tree types)
- It provides some reuse
- It reveals important role for subtrees/labelNode

Bad:
- It’s ugly and verbose
- For any given tree type, there’s really only one sensible way to define the subtrees function ...

Type Classes:

What distinguishes "tree types" from other types?

- a value of a tree type can have zero or more subtrees

And, for any given tree type, there's really only one sensible way to do this.

class Tree t where
        subtrees :: t -> [t]

For Instance(s):

instance Tree (BinTree a) where
        subtrees (Leaf x) = []
        subtrees (l ::: r) = [l, r]

instance Tree (LabTree l a) where
        subtrees (Tip a) = []
        subtrees (Lfork s l r) = [l, r]

instance Tree (STree a) where
        subtrees Empty = []
        subtrees (Split s l r) = [l, r]
... continued:

```
instance Tree (RoseTree a) where
    subtrees (Node x cs) = cs

instance Tree Expr where
    subtrees (Var s) = []
    subtrees (IntLit n) = []
    subtrees (Plus l r) = [l, r]
    subtrees (Mult l r) = [l, r]
```

So What?

---

**Generic Operations on Trees:**

```
depth :: Tree t => t -> Int
depth = (1+) . foldl max 0 . map depth . subtrees

size :: Tree t => t -> Int
size = (1+) . sum . map size . subtrees

paths :: Tree t => t -> [[t]]
paths t | null br = [[t]]
| otherwise = [tip | b <- br, p <- paths b]
    where br = subtrees t

dfs :: Tree t => t -> [t]
dfs t = t : concat (map dfs (subtrees t))
```

`Tree t =>` means "any type t, so long as it is a `Tree` type ..." (i.e., so long as it has a `subtrees` function)

---

**Implicit Parameterization:**

- An operation with a type `Tree t =>` ... is implicitly parameterized by the definition of a `subtrees` function of type `t -> [t]`
- (The implementation doesn’t have to work this way ...)
- Because there is at most one such function for any given type `t`, there is no need for us to write the `subtrees` parameter explicitly
- That’s good because it can mean less clutter, more clarity

---

**Labeled Trees:**

- To be able to convert trees into dot format, we need the nodes to be labeled with strings.
- Not all trees are labeled in this way, so we create a subclass
  ```
  class Tree t => LabeledTree t where
      label :: t -> String
  ```
- (Is this an appropriate use of overloading?)

---

**LabeledTree Instances:**

```
instance LabeledTree (BinTree String) where
    label (Leaf x) = x
    label (l :+: r) = ""

instance LabeledTree (LabTree String String) where
    label (Tip a) = a
    label (LFork s l r) = s

instance LabeledTree (STree String) where
    label Empty = ""
    label (Split s l r) = s
```

Needs hugs -98, for example

---

... continued:

```
instance LabeledTree (RoseTree String) where
    label (Node x cs) = x

instance LabeledTree Expr where
    label (Var s) = s
    label (IntLit n) = show n
    label (Plus l r) = "+
    label (Mult l r) = "*

...`
**Generic Tree -> dot:**

```
toDot :: LabeledTree t => t -> IO ()
toDot t = writeFile "tree.dot"
  ("digraph tree {
    "+ semi (nodeTree [t] ++ "\n")
  where semi = foldr (\l is -> l ++ "\n" ++ is) ""

nodeTree :: LabeledTree t => Path -> t -> [String]
nodeTree p t
  = [ showPath p ++ " [label="\" ++ label t ++ "\"] " ]
    ++ concat (zipWith (edgeTree p) [1..] (subtrees t))

edgeTree :: LabeledTree t => Path -> Int -> t -> [String]
edgeTree p n c = [ showPath p ++ " -> " ++ showPath p' ]
    ++ nodeTree p' c
  where p' = n : p
```

**Example:**

```
toDot (Node "a" [Node "b" [],
    Node "c" [],
    Node "d" [Node "e" []]])
```

```
+    
 o   
 x 3 y
```

**Example:**

```
toS (Plus (Mult (Var "x") (IntLit 3))
  (Mult (Var "y") (IntLit 5)))
```

```
+    
 o   
 x 3 y
```

**Example:**

```
Main> toDot example
ERROR - Unresolved overloading
*** Type : LabeledTree (BinTree Int) => IO ()
*** Expression : toDot example
```

```
Main>
```

**The Functor Class:**

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

instance Functor [] where ...
instance Functor Maybe where ...

```
-- fmap id == id
-- fmap (f . g) == fmap f . fmap g
```
Tree Instances:

```haskell
instance Functor BinTree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (l :+: r) = fmap f l :+: fmap f r

instance Functor (LabTree l) where
    fmap f (Tip a) = Tip (f a)
    fmap f (LFork s l r) = LFork s (fmap f l) (fmap f r)

instance Functor STree where
    fmap f Empty = Empty
    fmap f (Split s l r) = Split (f s) (fmap f l) (fmap f r)

instance Functor RoseTree where
    fmap f (Node x cs) = Node (f x) (map (fmap f) cs)
```

Why no instance for Expr?

Example:

```
Main> toDot (fmap show (example :+: example))
```

```
Main> depth (example :+: example)
6
Main>
```

Between One and All:

- Haskell allows us to define (monomorphic) functions that have just one possible instantiation:
  ```haskell
  not :: Bool -> Bool
  ```

- And (polymorphic) functions that work for all instantiations:
  ```haskell
  id :: a -> a
  ```

- But not all functions fit comfortably into these two categories ...

Type Classes:

- We’ve been exploring one of the most novel features that was introduced in the design of Haskell

- Similar ideas are now filtering in to other popular languages (e.g., “concepts” in C++)

- We’ll spend the rest of our time in this lecture looking at the original motivation for type classes

Addition:

- What type should we use for the addition operator (+)?

- Picking a monomorphic type like
  ```haskell
  Int -> Int -> Int
  ```
  is too limiting, because this can’t be applied to other numeric types

- Picking a polymorphic type like
  ```haskell
  a -> a -> a
  ```
  is too general, because addition only works for “numeric types” ...

Equality:

- What type should we use for the equality operator (==)?

- Picking a monomorphic type like
  ```haskell
  Int -> Int -> Bool
  ```
  is too limiting, because this can’t be applied to other numeric types

- Picking a polymorphic type like
  ```haskell
  a -> a -> Bool
  ```
  is too general, because there is no computable equality on function types ...
**Numeric Literals:**

- What type should we use for the type of the numeric literal 0?

- Picking a monomorphic type like `Int` is too limiting, because then it can’t be used for other numeric types.
  - And functions like `sum = foldl (+) 0` inherit the same restriction and can only be used on limited types.

- Picking a polymorphic type like `a` is too general, because there is no meaningful interpretation for zero at all types ...

**Workarounds (1):**

- We could use different names for the different versions of an operator at different types:
  - `(+)` :: `Int` -> `Int` -> `Int`
  - `(+')` :: `Float` -> `Float` -> `Float`
  - `(+")` :: `Integer` -> `Integer` -> `Integer`
  - ...

- Apart from the fact that this is really ugly, it prevents us from defining general functions that use addition (again, `sum = foldl (+) 0`)

**Workarounds (2):**

- We could just define the “unsupported” cases with dummy values.
  - 0 :: `Int` produces an integer zero
  - 0 :: `Float` produces a floating point zero
  - 0 :: `Int` -> `Bool` produces some undefined value (e.g., sends the program into an infinite loop)

- Attitude: “More fool you, programmer, for using zero with an inappropriate type!”

**Workarounds (3):**

- We could inspect the values of arguments that are passed in to each function to determine which interpretation is required.

- Works for `(+)` and `(==)` (although still requires that we assign a polymorphic type, so those problems remain)

- But it won’t work for 0. There are no arguments here from which to infer the type of zero that is required; that information can only be determined from the context in which it is used.

**Workarounds (4):**

- Miranda and Orwell (two predecessors of Haskell) included a type called “`Num`” that included both floating point numbers and integers in the same type
  - `data Num = In Integer | Fl Float`

- Now (+) can be treated as a function of type `Num` -> `Num` -> `Num` and applied to either integers or floats, or even mixed argument types.

- But we’ve lost a lot: types don’t tell us as much, and basic arithmetic operations are more expensive to implement ...

**Between a rock ...**

- In these examples, monomorphic types are too restrictive, but polymorphic types are too general.

- In designing the language, the Haskell Committee had planned to take a fairly conservative approach, consolidating the good ideas from other languages that were in use at the time.

- But the existing languages used a range of awkward and ad-hoc techniques and nobody had a good, general solution to this problem ...
“How to make ad-hoc polymorphism less ad-hoc”

- In 1989, Philip Wadler and Stephen Blott proposed an elegant, general solution to these problems.
- Their approach was to introduce a way of talking about sets of types (“Type Classes”) and their elements (“Instances”).
- The Haskell committee decided to incorporate this innocent and attractive idea into the first version of Haskell ...

Type Classes:

- A type class is a set of types.
- Haskell provides several built-in type classes, including:
  - **Eq**: types whose elements can be compared for equality
  - **Num**: numeric types
  - **Show**: types whose values can be printed as strings
  - **Integral**: types corresponding to integer values,
  - **Enum**: types whose values can be enumerated (and hence used in \([m..n]\) notation)

A (Not-Well Kept) Secret:

- Users can define their own type classes.
- This can sometimes be very useful.
- It can also be abused.
- For now, we’ll just focus on understanding and using the built-in type classes ...

Instances:

- The elements of a type class are known as the instances of the class.
- If \(C\) is a class and \(t\) is a type, then we write \(C t\) to indicate that \(t\) is an element-instance of \(C\).
- (Maybe we should have used \(t \in C\), but the \(\in\) symbol wasn’t available in the character sets or on the keyboards of last century’s computers ... :-)

Instance Declarations:

- The instances of a class are specified by a collection of instance declarations:
  ```haskell
  instance Eq Int
  instance Eq Integer
  instance Eq Float
  instance Eq Double
  instance Eq Bool
  instance Eq a => Eq [a]
  instance Eq a => Eq (Maybe a)
  instance (Eq a, Eq b) => Eq (a,b)
  ...
  ```

... continued:

- In set notation, this is equivalent to saying that:
  ```haskell
  Eq = \{ Int, Integer, Float, Double, Bool \} 
  \cup \{ [t] \mid t \in Eq \}
  \cup \{ Maybe t \mid t \in Eq \}
  \cup \{ (t_1, t_2) \mid t_1 \in Eq, t_2 \in Eq \}
  ```
- \(Eq\) is an infinite set of types, but it doesn’t include all types (e.g., types like \(Int -> Int\) and \([[Int] -> Bool]\) are not included)
Derived Instances (1):

- The prelude provides a number of types with instance declarations that include those types in the appropriate classes.

- Classes can also be extended with definitions for new types by using a deriving clause:
  ```haskell
data T = ... deriving Show
data S = ... deriving (Show, Ord, Eq)
```

- The compiler will check that the types are appropriate to be included in the specified classes.

Operations:

- The prelude also provides a range of functions, with restricted polymorphic types:
  ```haskell
(==) :: Eq a => a -> a -> Bool
(+ ) :: Num a => a -> a -> a
min :: Ord a => a -> a -> a
show :: Show a => a -> String
fromInteger :: Num a => Integer -> a
```

- A type of the form \( C \ a \rightarrow T(a) \) represents all types of the form \( T(t) \) for any type \( t \) that is an instance of the class \( C \).

Terminology:

- An expression of the form \( C \ t \) is often referred to as a constraint, a class constraint, or a predicate.

- A type of the form \( C \ t \Rightarrow \ldots \) is often referred to as a restricted type or as a qualified type.

- A collection of predicates \( (C \ t, D \ t', \ldots) \) is often referred to as a context. The parentheses can be dropped if there is only one element.

Type Inference:

- Type Inference works just as before, except that now we also track constraints.

- Example: \( \text{null} \ 	ext{xs} = (\text{xs} == []) \)
  - Assume \( \text{xs} :: a \)
  - Pick \( (==) :: b -> b -> \text{Bool} \) with the constraint \( \text{Eq} \ b \)
  - Pick instance \( [] :: [c] \)
  - From \( (\text{xs} == []) \), we infer \( a = b = [c] \), with result type of \( \text{Bool} \)
  - Thus: \( \text{null} :: \text{Eq} [c] => [c] -> \text{Bool} \)

Examples:

- We can treat the integer literal 0 as sugar for \( \text{fromInteger} \ 0 \), and hence use this as a value of any numeric type
  - Strictly speaking, its type is \( \text{Num} \ a => a \), which means any type, so long as it’s numeric ...

- We can use \( (==) \) on integers, booleans, floats, or lists of any of these types ... but not on function types

- We can use \( (+) \) on integers or on floating point numbers, but not on Booleans

... continued:

- **Note:** In this case, it would probably be better to use the following definition:
  ```haskell
null :: [a] -> \text{Bool}
null [] = \text{True}
null (x:xs) = \text{False}
```

- The type \( [a] -> \text{Bool} \) is more general than \( \text{Eq} a => [a] -> \text{Bool} \), because the latter only works with “equality types”
Inheriting Predicates:

- Predicates in the type of a function \( f \) can “infect” the type of a function that calls \( f \).

- The functions:
  
  \[
  \text{member } xs \; x = \text{any} \; (x==) \; xs
  \]
  
  \[
  \text{subset } xs \; ys = \text{all} \; (\text{member } ys) \; xs
  \]

  have types:

  \[
  \text{member} :: \text{Eq } a \Rightarrow [a] \rightarrow a \rightarrow \text{Bool}
  \]
  
  \[
  \text{subset} :: \text{Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
  \]

... continued:

- For example, now we can define:

  \[
  \text{data Day} = \text{Sun}|\text{Mon}|\text{Tue}|\text{Wed}|\text{Thu}|\text{Fri}|\text{Sat}
  \]
  
  \[
  \text{deriving} \; (\text{Eq}, \text{Show})
  \]

  And then apply \text{member} and \text{subset} to this new type:

  ```
  Main> \text{member} [\text{Mon, Tue, Wed, Thu, Fri}] \; \text{Wed}
  True
  Main> \text{subset} [\text{Mon, Sun}] [\text{Mon, Tue, Wed, Thu, Fri}]
  False
  Main>
  ```

Eliminating Predicates:

- Predicates can be eliminated when they are known to hold.

- Given the standard prelude function:

  \[
  \text{sum} :: \text{Num } a \Rightarrow [a] \rightarrow a
  \]

  and a definition

  \[
  \text{gauss} = \text{sum} [1..10::\text{Integer}]
  \]

  we could infer a type

  \[
  \text{gauss} :: \text{Num Integer} \Rightarrow \text{Integer}
  \]

  But then simplify this to

  \[
  \text{gauss} :: \text{Integer}
  \]

Detecting Errors:

- Errors can be raised when predicates are known not to hold:

  ```
  Prelude> 'a' + 1
  ERROR - Cannot infer instance
  *** Instance   : \text{Num Char}
  *** Expression : 'a' + 1
  Prelude> (\x -> x)
  ERROR - Cannot find "show" function for:
  *** Expression : \x -> x
  *** Of type    : a -> a
  Prelude>
  ```

Derived Instances (2):

- What if you define a new type and you can’t use a derived instance?
  - Example: \text{data Set } a = \text{Set } [a] \; \text{deriving } \text{Num}
  - What does it mean to do arithmetic on sets?
  - How could the compiler figure this out from the definition above?

- What if you define a new type and the derived equality is not what you want?
  - Example: \text{data Set } a = \text{Set } [a]
  - We’d like to think of \text{Set } [1,2] and \text{Set } [2,1] and \text{Set } [1,1,1,2,2,1,2] as equivalent sets

Example: Derived Equality

- The derived equality for \text{Set} gives us:

  \[
  \text{Set } xs \; == \; \text{Set } ys \; = \; xs \; == \; ys
  \]

- And the equality on lists gives us:

  ```
  [] \; == \; [] \; = \; \text{True}
  (x:xs) \; == \; (y:ys) \; = \; (x==y) \; \&\& \; (xs==ys)
  - \; == \; - \; = \; \text{False}
  ```

- A derived equality function tests for structural equality ... what we need for \text{Set} is not a structural equality
Class Declarations:

- Before we can define an instance, we need to look at the class declaration:

  ```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) or (/=)
  x == y      = not (x /= y)
  x /= y       = not (x == y)
  
  To define an instance of equality, we will need to provide an implementation for at least one of the operators (==) or (/=)
  
  class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) or (/=)
  x == y      = not (x /= y)
  x /= y       = not (x == y)
```

Member Functions:

- In a class declaration

  ```haskell
class C a where
  f, g, h :: T(a)
  
  From a user’s perspective, just like any other type qualified by a predicate
  
  From an implementer’s perspective, these are the operations that we have to code to define an instance
  
  class C a where
  f, g, h :: T(a)
  ````

Instance Declarations:

- We can define a non-structural equality on the Set datatype using the following:

  ```haskell
  instance Eq a => Eq (Set a) where
  Set xs == Set ys
  = (xs `subset` ys) && (ys `subset` xs)
  
  This works as we’d like ...
  ````

Overloading:

- Type classes support the definition of overloaded functions

  "Overloading", because a single identifier can be overloaded with multiple interpretations

  But just because you can ... it doesn’t mean you should!

  Use judiciously, where appropriate, where there is a coherent, unifying view of each overloaded function should do

  ```haskell
  overloads
  ````

Defining New Classes:

- Can I define new type classes in my program or library?
  - Yes!

- Should I define new type classes in my program or library?
  - Yes, if it makes sense to do so!
  - What common properties would the instances to share, and how should this be reflected in the choice of the operators?
  - Does it make sense for the meaning of a symbol to be uniquely determined by the types of the values that are involved?

Beware of Ambiguity!

- What if there isn’t enough information to resolve overloading?
  - Early versions of Hugs would report an error if you tried to evaluate `show []`.
  - The system infers a type `Show a => String`, and doesn’t know what type to pick for the "ambiguous" variable `a`.
  - It could make a difference: `show ([]:[Int]) = "["`, but `show ([]:[Char]) = "\""

  - Recent versions use defaulting to pick a default choice ...
  - ... but the results there are also less than ideal …
Summary:

- Type classes provide a way to describe sets of types and related families of operations that are defined on their instances.
- A range of useful type classes are built-in to the prelude.
- Classes can be extended by deriving new instances or defining your own.
- New classes can also be defined.
- Once you've experienced programming with type classes, it's hard to go without ...