QuickCheck
QuickCheck is a lightweight tool for random testing of Haskell programs, developed by Koen Claessen and John Hughes.

- Based on specifications of desired properties, expressed as Haskell functions
- Properties are verified on randomly generated test data.
- The class system is used in clever ways to make everything look simple.
A Simple Property of Lists

prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs

Prelude Main> Quickcheck.quickCheck prop_RevApp
OK, passed 100 tests.
prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs

Prelude Main> Quickcheck.quickCheck prop_RevApp
OK, passed 100 tests.

N.b.: the type declaration on the property is required here, because we need to restrict its type to a particular instance — only monomorphic properties can be checked by QuickCheck.
Suppose we mess up the specification:

```haskell
cprop_BadRevApp :: [Int] -> [Int] -> Bool
cprop_BadRevApp xs ys =
    reverse (xs ++ ys) == reverse xs ++ reverse ys
```
Suppose we mess up the specification:

\[
\text{prop\_BadRevApp} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}
\]

\[
\text{prop\_BadRevApp} \; \text{xs} \; \text{ys} = \\
\quad \text{reverse} \; (\text{xs} \; ++ \; \text{ys}) == \text{reverse} \; \text{xs} \; ++ \; \text{reverse} \; \text{ys}
\]

Prelude Main> Quickcheck.quickCheck prop_BadRevApp
Falsifiable, after 4 tests:
[-3,-4,-4]
[-4,-1,1,1]
[-4,-1,1,1]
Many properties are not true universally (for all inputs of appropriate types), but only for inputs satisfying some conditions.

```haskell
ins :: Ord a => a -> [a] -> [a]
ins a [] = [a]
ins a (a':as) = if a < a'
    then a:a':as
    else a':(ins a as)

ordered :: Ord a => [a] -> Bool
ordered (a:a':as) = (a<=a') && (ordered (a':as))
ordered _ = True

prop_BadIns :: Int -> [Int] -> Bool
prop_BadIns a as = ordered (ins a as)
```
Prelude Main> Quickcheck.quickCheck prop_BadIns
Falsifiable, after 9 tests:
4
[5,-3]
We can make a property **conditional** by writing it as
\[
<\text{condition}> \implies <\text{property}>
\]

```haskell
prop_Ins :: Int -> [Int] -> Property
prop_Ins a as = (ordered as) ==> (ordered (ins a as))
```

Prelude Main> Quickcheck.quickCheck prop_Ins
OK, passed 100 tests.
We can make a property **conditional** by writing it as
<condition> ==> <property>:

```
prop_Ins :: Int -> [Int] -> Property
prop_Ins a as = (ordered as) ==> (ordered (ins a as))
```

Prelude Main> Quickcheck.quickCheck prop_Ins
OK, passed 100 tests.

Note that the result type of `prop_Ins` has changed from `Bool` to `Property`. This is because the “testing semantics” of conditional properties is a little more tricky than for simple properties.
insWrong :: Ord a => a -> [a] -> [a]

insWrong a [] = [a]
insWrong a as
  | (length as) == 6 = as ++ [a]
  | otherwise = ins a as

prop_InsWrong :: Int -> [Int] -> Property
prop_InsWrong a as =
  (ordered as) ==> (ordered (insWrong a as))

Prelude Main> Quickcheck.quickCheck prop_InsWrong
OK, passed 100 tests.
QuickCheck provides combinators for investigating the distribution of test cases.

```
collect :: a -> b -> Property
classify :: Bool -> String -> a -> Property
trivial :: Bool -> a -> Property
```

To see information about distribution, use `verboseCheck` instead of `quickCheck`.
prop_INSWrong’ :: Int -> [Int] -> Property
prop_INSWrong’ a as =
  (ordered as) ==> 
    collect (length as) $
    classify (ordered (a:as)) "at-head" $
    classify (ordered (as++[a])) "at-tail" $
  (ordered (insWrong a as))
Prelude Main> Quickcheck.verboseCheck prop_InsWrong'
...
OK, passed 100 tests.
42% 0, at-head, at-tail.
12% 1, at-tail.
11% 2, at-tail.
9% 2, at-head.
7% 2.
7% 1, at-head.
6% 1, at-head, at-tail.
2% 3, at-tail.
2% 3.
1% 4, at-head.
1% 3, at-head.
We can try to fix the distribution by adding another condition:

```haskell
prop_InsWrong'' :: Int -> [Int] -> Property
prop_InsWrong'' a as =
  (ordered as) && (length as >= 5) ==> 
  (ordered (insWrong a as))
```
We can try to fix the distribution by adding another condition:

```haskell
prop_InsWrong'' :: Int -> [Int] -> Property
prop_InsWrong'' a as =
  (ordered as) && (length as >= 5) ==> 
  (ordered (insWrong a as))
```

However:

```
Prelude Main> Quickcheck.quickCheck prop_InsWrong''
Arguments exhausted after 0 tests.
```
class Arbitrary a where
  arbitrary :: Gen a
class Arbitrary a where
  arbitrary :: Gen a

QuickCheck provides generators for most base types such as \texttt{Int}, \texttt{Char}, \texttt{Float}, and lists.

QuickCheck also provides combinators for building custom generators...
newtype Gen a = Gen (Rand -> a)
-- (roughly; in fact, Gen is an abstract type)

choose :: (Int,Int) -> Gen Int

oneof :: [Gen a] -> Gen a
oneof [return Heads, return Tails]

frequency :: [(Int, Gen a)] -> Gen a
frequency [(1, return Heads), (2, return Tails)]

e tc...

N.b.: The returns here are because Gen is a monad.
We can use these primitives to build generators for a variety of types. E.g. ...

```haskell
instance Arbitrary Int where
  arbitrary = choose (-20,20)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
  arbitrary = liftM2 (,) arbitrary arbitrary
```
orderedList =
  do a <- frequency
     [(1, return []),
      (7, liftM2 (:) arbitrary arbitrary)]
  return (sort a)

prop_InsWrong''' :: Int -> Property
prop_InsWrong''' a =
  forAll orderedList $ \ as -> ordered (insWrong a as)
A Custom Generator for Ordered Lists

orderedList =
    do a <- frequency
       [[(1, return []),
         (7, liftM2 (:) arbitrary arbitrary)]
    return (sort a)

prop_InsWrong''' :: Int -> Property
prop_InsWrong''' a =
    forAll orderedList $ \ as -> ordered (insWrong a as)

Prelude Main> Quickcheck.quickCheck prop_InsWrong'''
Falsifiable, after 19 tests:
  0
  [-5,0,3,5,7,8]

Whew.
Here is a naive definition of arbitrary lists:

```haskell
instance Arbitrary a => Arbitrary [a] where
    arbitrary =
        oneof [return [],
               liftM2 (:) arbitrary arbitrary]
```

Why is this not what we want?
Here is a naive definition of arbitrary lists:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary =
    oneof [return [],
           liftM2 (:) arbitrary arbitrary]
```

Why is this not what we want?
Better:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary =
    frequency [(1, return []),
               (7, liftM2 (:) arbitrary arbitrary)]
```
However, in some cases we need to be even more careful...

data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving Show

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency
      [(1, liftM Leaf arbitrary),
       (2, liftM2 Branch arbitrary arbitrary)]

What goes wrong?
However, in some cases we need to be even more careful...

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving Show

instance Arbitrary a => Arbitrary (Tree a) where
    arbitrary =
        frequency
            [(1, liftM Leaf arbitrary),
             (2, liftM2 Branch arbitrary arbitrary)]
```

What goes wrong?

```
Prelude Main> Quickcheck.quickCheck prop_SomeTreeProperty
Stack space overflow: current size 1048576 bytes.
```
Given our definition, an arbitrary tree has only a 50% chance of being finite!

Intuition: If the first few choices yield Branches, then the only way for the tree to be finite is for many subtrees to choose (with 1/3 probability) to be leaves.
We need to be able to control the size of the generated data. This is accomplished by changing the definition of the \texttt{Gen} monad:

\begin{verbatim}
    newtype Gen a = Gen (Int \rightarrow Rand \rightarrow a)
\end{verbatim}
The **sized** combinator given the programmer access to the “current size bound.”

```
sized :: (Int -> Gen a) -> Gen a
sized f = Gen (\n r -> m n r
    where Gen m = f n)
```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree

arbTree 0 = liftM Leaf arbitrary
arbTree n =
  frequency
    [(1, liftM Leaf arbitrary),
     (4, liftM2 Branch (arbTree (n 'div' 2))
      (arbTree (n 'div' 2)))]
Generating Random Functions

Since Haskell encourages higher-order programming, we may well want to use QuickCheck to test functions that take other functions as parameters. To do so, we need to be able to generate random functions. Surprisingly, this is possible.
We want to build a function generator of type \( \text{Gen} \ (a \to b) \).

Unpacking the definition of \( \text{Gen} \), we find that this is \( \text{Int} \to \text{Rand} \to a \to b \).

But this type is isomorphic to \( a \to \text{Int} \to \text{Rand} \to b \), which is the representation of \( a \to \text{Gen} \ b \).
I.e., we can define a function

\[
\text{promote} :: (a \to \text{Gen } b) \to (\text{Gen } (a\to b))
\]

\[
\text{promote } f = \text{Gen } (\n r \to
\quad \a \to
\quad \quad \text{m n r}
\quad \text{where Gen } m = f \ a)
\]

We can now use \texttt{promote} to build a generator for a function type \(a\to b\), given a function that takes an \(a\) and uses it to construct a \(b\) generator that depends in some way on the \(a\) argument.

Where do such functions come from?
class Coarbitrary a where
    coarbitrary :: a -> (Gen b -> Gen b)

I.e., `coarbitrary` takes a value of `a` and yields a generator transformer that takes a `b` generator and yields a new `b` generator whose behavior depends on the `a` argument.
We can now use arbitrary and coarbitrary, together with promote, to generate random functions as needed:

\[
\text{instance } (\text{Coarbitrary } a, \text{Arbitrary } b) => \\
\text{Arbitrary } (a \to b) \text{ where} \\
\text{arbitrary} = \text{promote} (a \to \text{coarbitrary } a \text{ arbitrary})
\]

All we need to do now is to define some instances of the class Coarbitrary.
Recall that all our generators were ultimately based on the `choose` function (which generates uniformly distributed integers from a given range).

Similarly, the foundation of all our generator transformers is a function

\[
\text{variant} :: \text{Int} \to \text{Gen } a \to \text{Gen } a
\]

\[
\text{variant } v \ (\text{Gen } m) = \\
\quad \text{Gen } (\lambda n \ r \to m \ n \ (\text{rands } r \ !!(v+1)))
\]

where `rands r0 = r1 : rands r2`

\[
\quad \text{where } (r1, r2) = \text{Random.split } r0
\]
instance Coarbitrary Bool where
    coarbitrary b = if b then variant 0 else variant 1
instance Arbitrary Int where
    arbitrary = sized $ \ n \rightarrow \ choose \ (-n,n)
    coarbitrary n = variant
                  (if n \geq 0 \ \then \ 2*n \ \else \ 2*(-n) + 1)
instance (Coarbitrary a, Coarbitrary b) => Coarbitrary (a, b) where
  coarbitrary (a, b) = coarbitrary a . coarbitrary b

Note how function composition (\( . \)) is used to combine the generator transformers for types \( a \) and \( b \).
instance Coarbitrary a => Coarbitrary [a] where
    coarbitrary [] = variant 0
    coarbitrary (a:as) = coarbitrary a . variant 1 . coarbitrary
• Thinking about properties (specifications) of functions is useful even when no errors are found by testing them.

• Indeed, many users report that, when errors are found by QuickCheck, they are just as often errors in the properties as in the code!

• The properties make excellent documentation, in part because they can be re-verified automatically as part of regression testing.
These slides are partly based on a nice presentation of QuickCheck by Jue Wang.