Program Verification with Dafny (Part I + II + III)

Nuno Macedo

Dafny

- Programming language and tool for developing verified programs
- Developed by Microsoft Research
- Multi-paradigm language (imperative, functional, object-oriented)
- Two main components:
  - **Specifications** (verified statically)
  - **Implementations** (compiles into C#)

Dafny Specifications

- Specifications are declarative, either logic or functional programming
- Immutable instances (value types, in contrast to reference types)
- Constructs include
  - Preconditions (`requires`) and postconditions (`ensures`)
  - Loop variants (`decreases`) and invariants (`invariant`)
  - Assertions (`assert`)
  - Lemmas (`lemma`)
  - Functions (`function`) and predicates (`predicate`)
  - Ghost (`ghost`) variables and methods

Dafny Implementations

- Implementations are mostly imperative (but functional can be used)
- Composed of procedures (`method`)
- Strongly typed (often inferred)
- Some object-oriented features (classes, inheritance, …)
- Supports reference types (arrays, classes, …), mutable instances

Dafny Verification

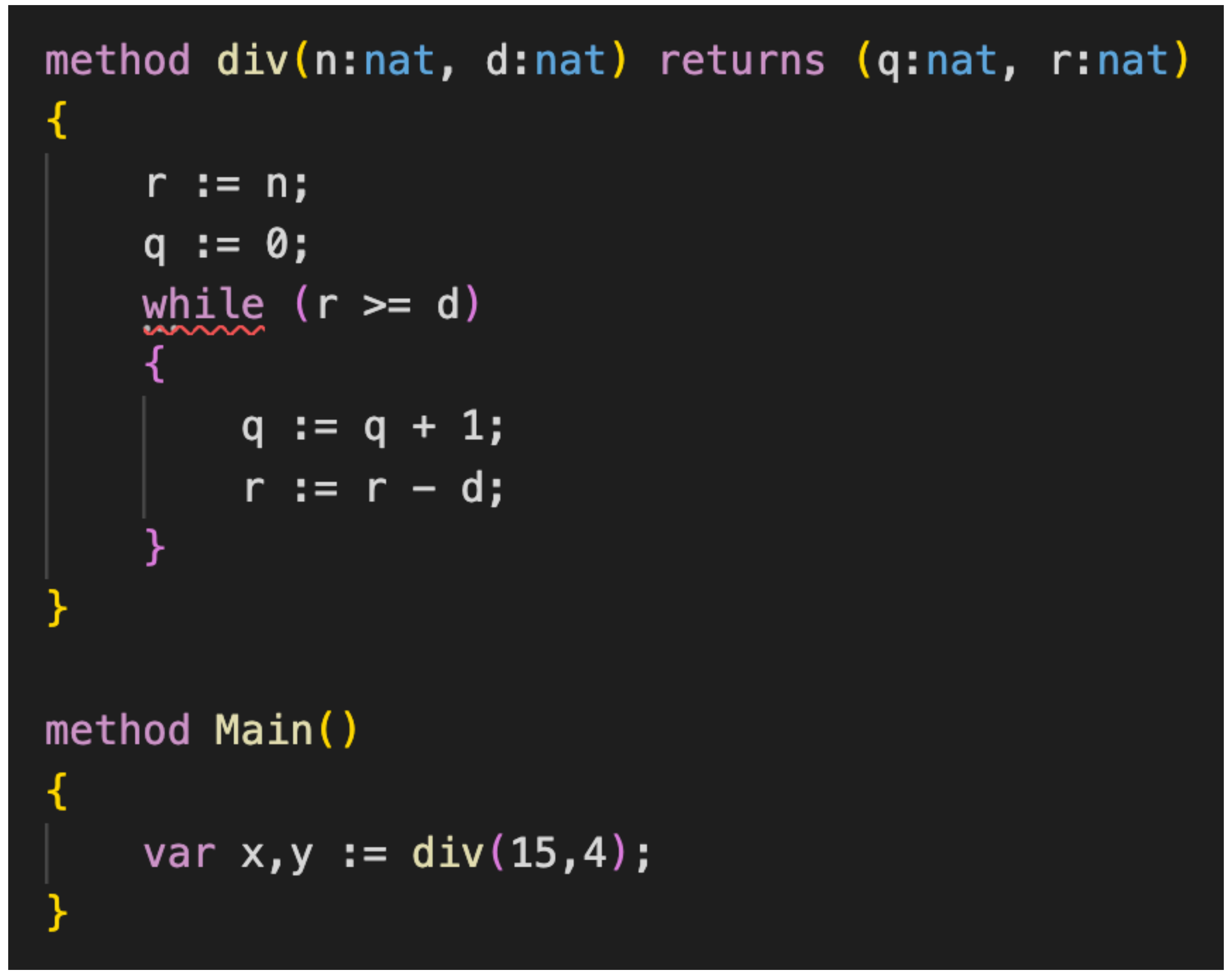
- Methods are annotated with specifications
- Specifications continuously verified statically (compilation-time)
    - Automates part of the tableau proof method
    - Invalid specifications are errors: does not compile
- Uses solvers underneath, shows counter-examples to violations
- Not fully automatic (*undecidable*): may require additional annotations
- Compiles implementations into executable code
- Specifications are **not** part of the executable code

Dafny Examples

- Consider only value types
- No aliasing, assignment is copy
- No arrays, no class objects, no side-effects

Example: Integer Division

- Methods are executable code
- No annotations, but already errors flagged
- Dafny verifies total correctness
  - correctness + termination
- Cannot prove termination without restrictions on input

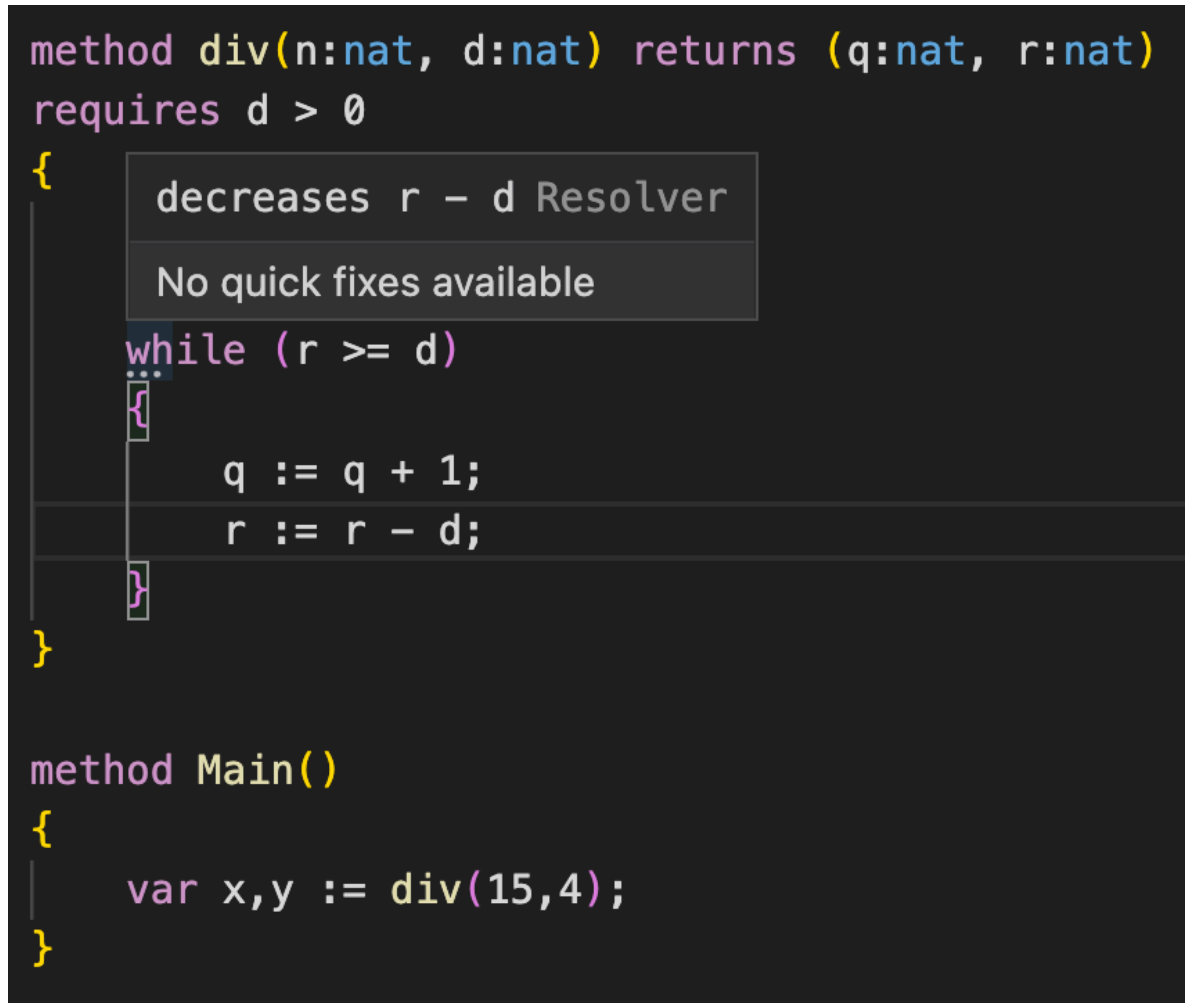


```
method div(n:nat, d:nat) returns (q:nat, r:nat)
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
}
```

Return values part of state, may return multiple.
Variables declared with `var`, type (often) inferred.

Example: Integer Division

- With the `requires` precondition already proves termination
- Loop variant automatically inferred
- Can be made explicit with `decreases`
- Information from types (`nat`, ≥0) also used

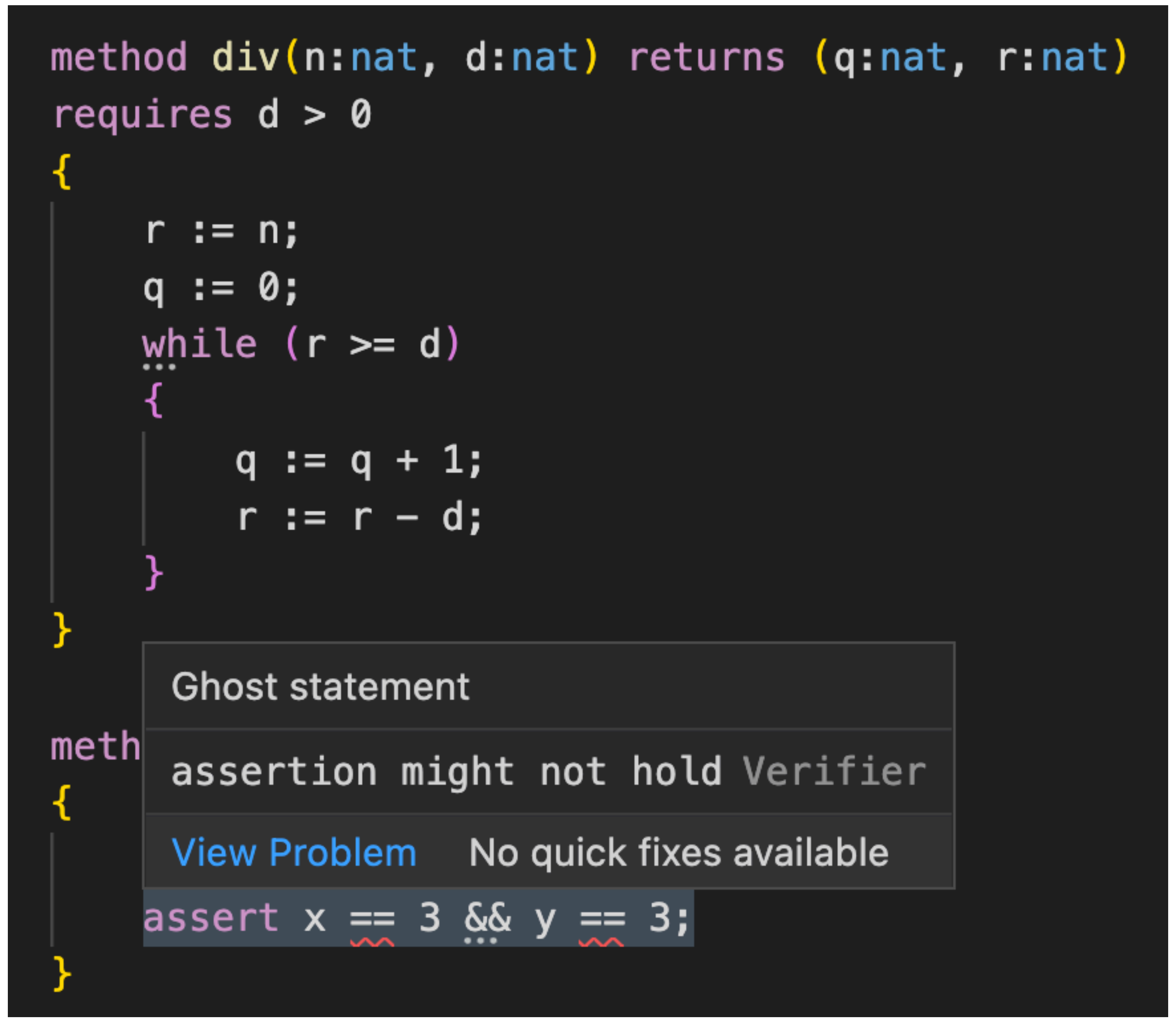


```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
{
    decreases r - d Resolver
    No quick fixes available
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
}
```

Blue annotations are informative.
Code can be executed when there are no errors.

Example: Integer Division

- Can define basic tests as assertions
- These are not checked in runtime as in other languages
- Verified during compilation
- Invalid: methods are black-box for proofs
- Only the pre and postconditions are known



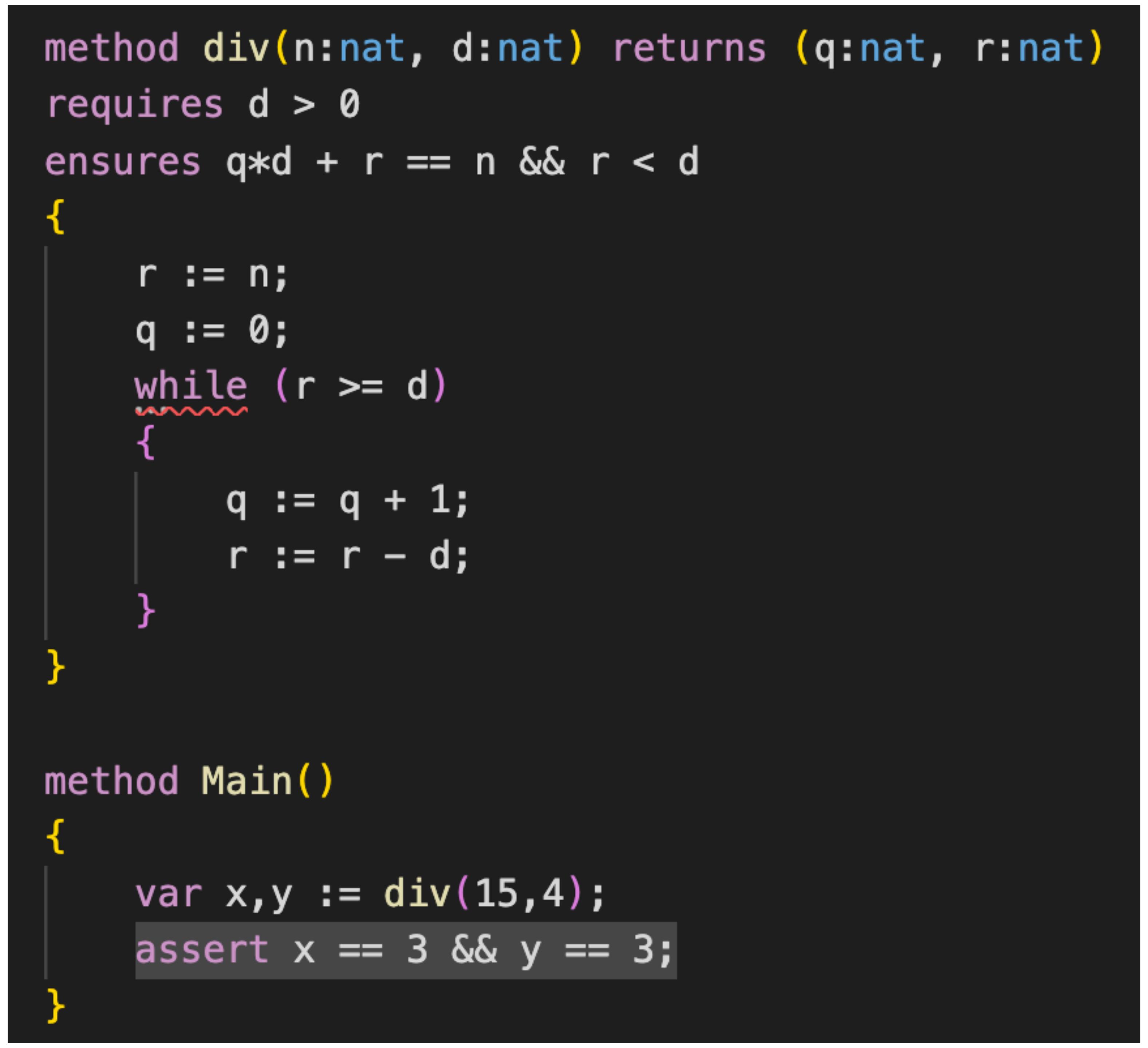
```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

meth
{
    assert x == 3 && y == 3;
}
```

```
Ghost statement
assertion might not hold Verifier
View Problem    No quick fixes available
```

Counter-examples can be shown (not always helpful).

Example: Integer Division

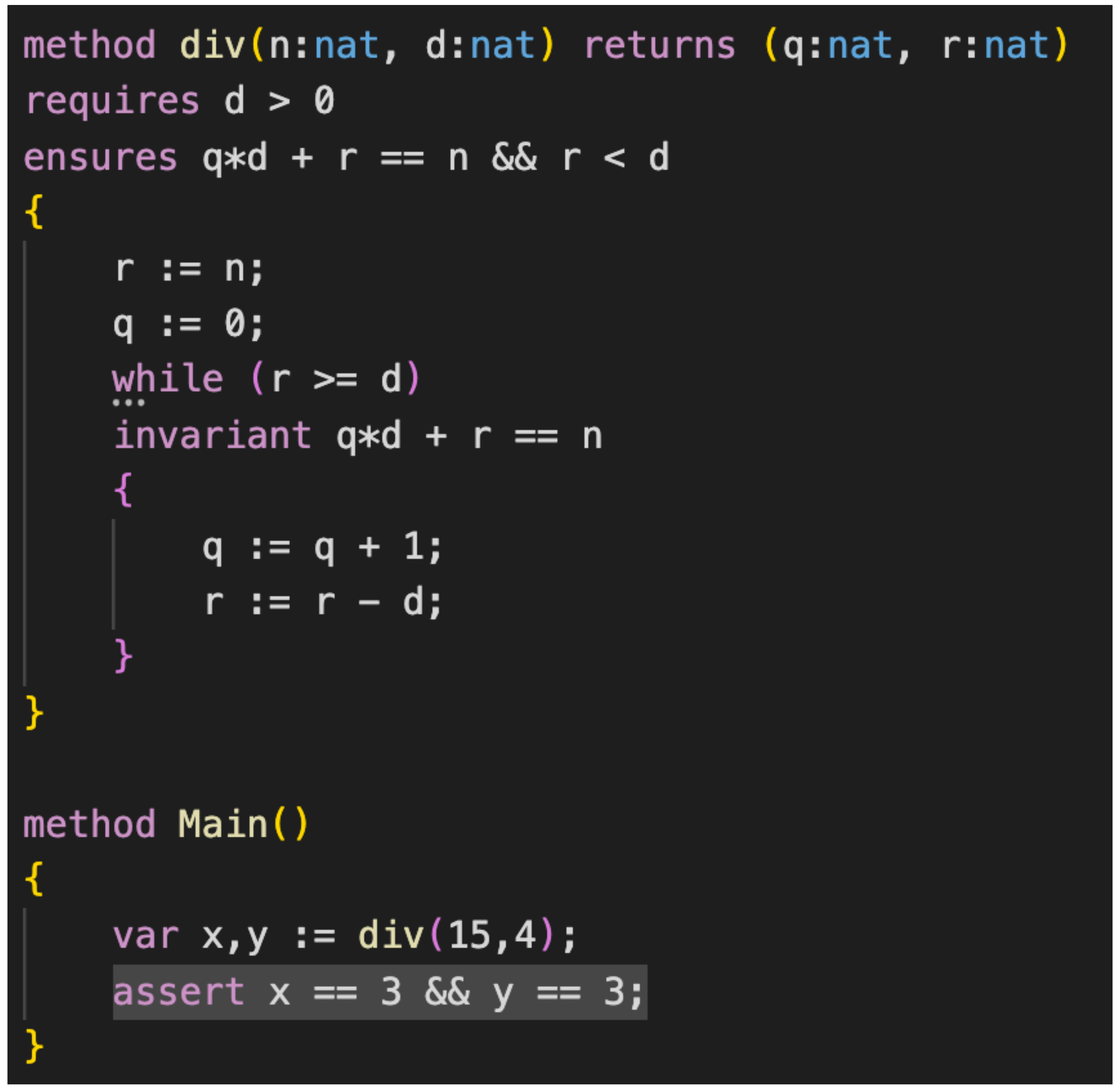
- Postcondition added
- Assertion is now valid: consistent with postcondition
- But still not proved that code satisfied postcondition
- Loops require additional annotations



```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
ensures q*d + r == n && r < d
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
    assert x == 3 && y == 3;
}
```

Example: Integer Division

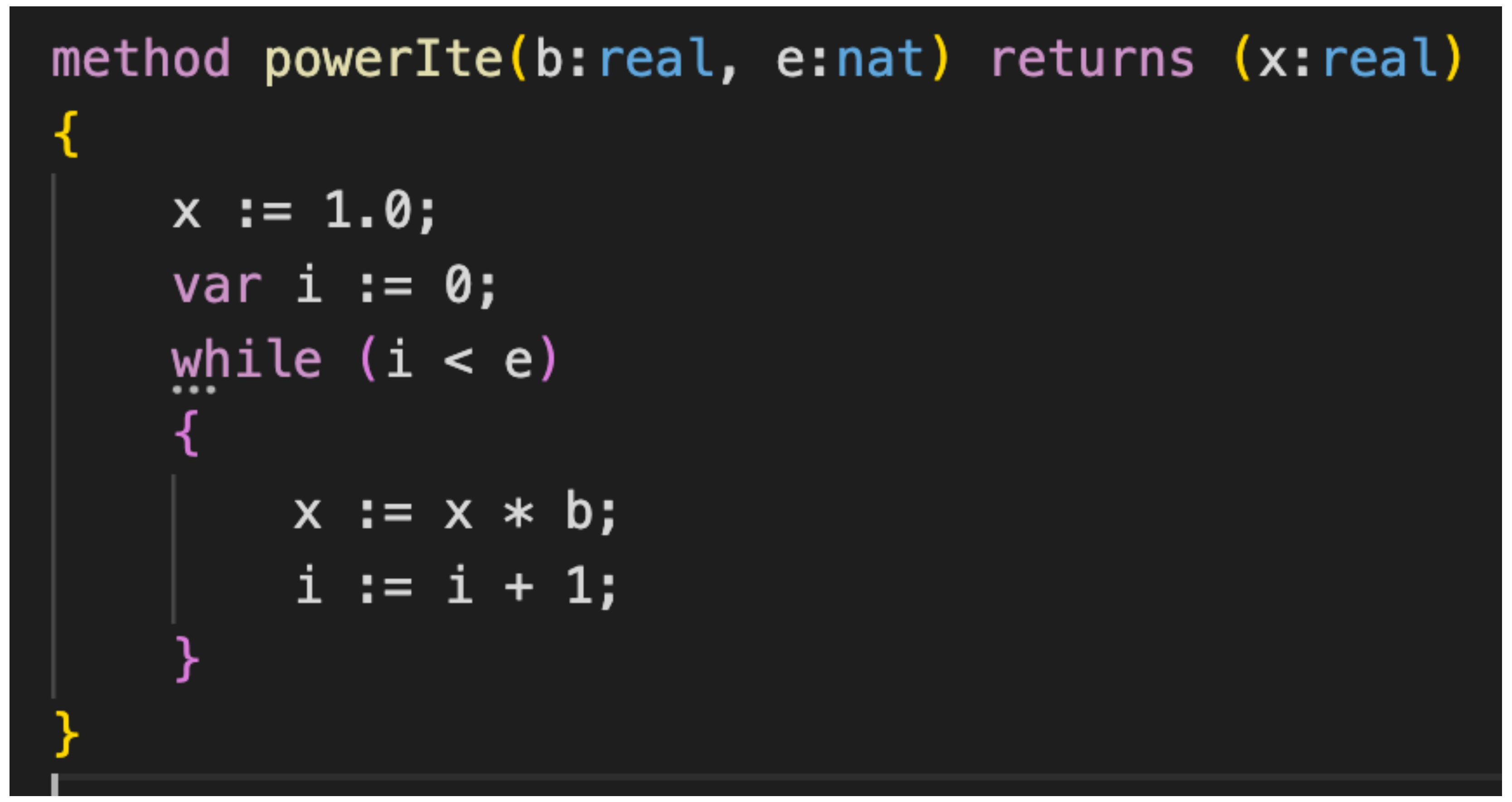
- Loop invariant: must hold before, in all iterations, and after
- Must imply the postcondition
- Usually also states the range of iterator
- Creative step, cannot be automated by tools
- Variants often automatically detected, but not always



```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
ensures q*d + r == n && r < d
{
    r := n;
    q := 0;
    while (r >= d)
    invariant q*d + r == n
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
    assert x == 3 && y == 3;
}
```

Example: Integer Exponentiation

- Iterative, inefficient version (linear)
- How to specify the postcondition declaratively?
- Side-effect free functions

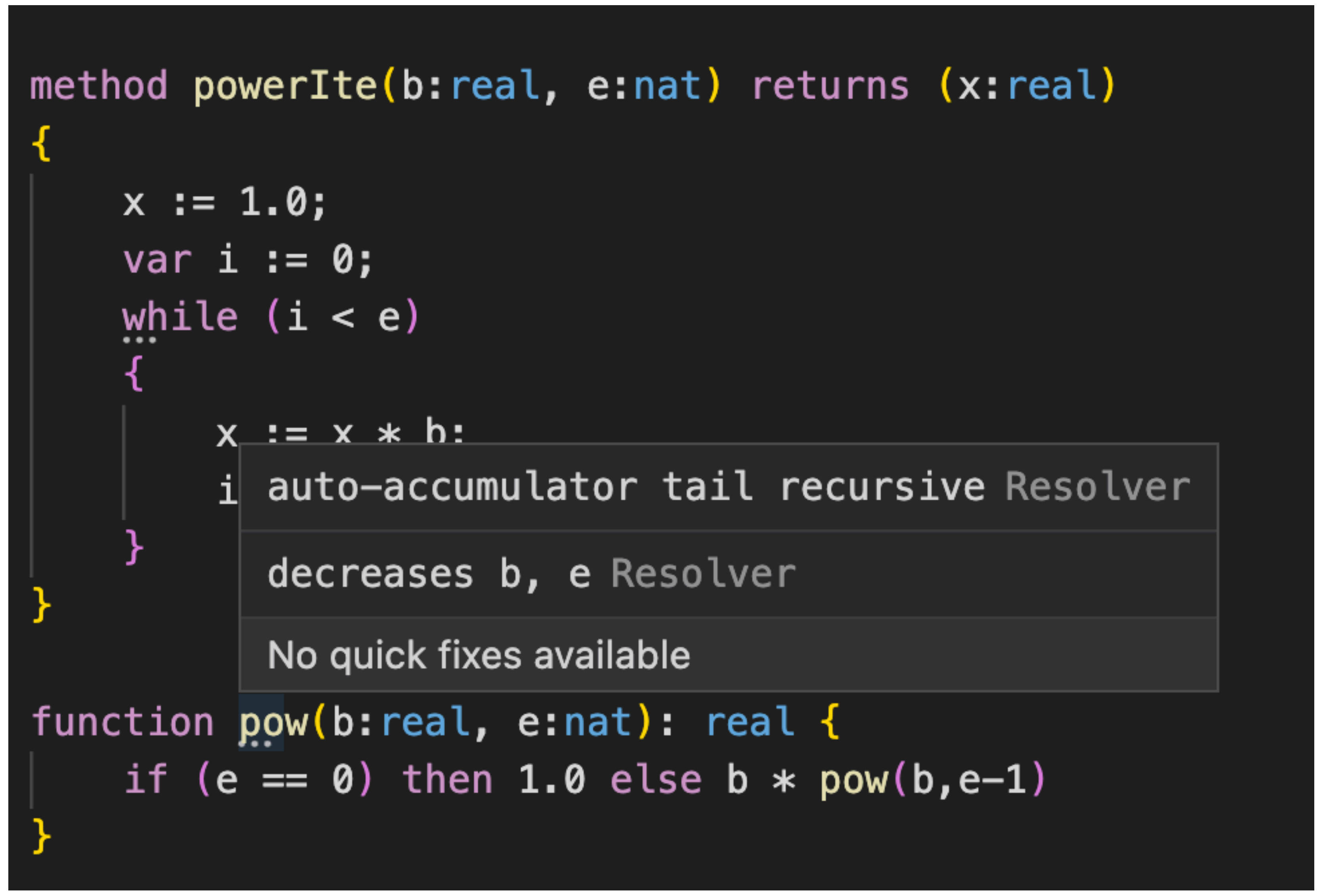


```
method powerIte(b:real, e:nat) returns (x:real)
{
    x := 1.0;
    var i := 0;
    while (i < e)
    {
        x := x * b;
        i := i + 1;
    }
}
```

Real constants require decimal part.

Example: Integer Exponentiation

- Functions: more abstract and declarative definition of behaviour
- Can be recursive (but no iteration)
- Pure, side-effect free, more prone to automatic analysis
    - Don't need postconditions
- Performance doesn't matter: will not compile
- Needs variant to prove termination, but often automatically inferred

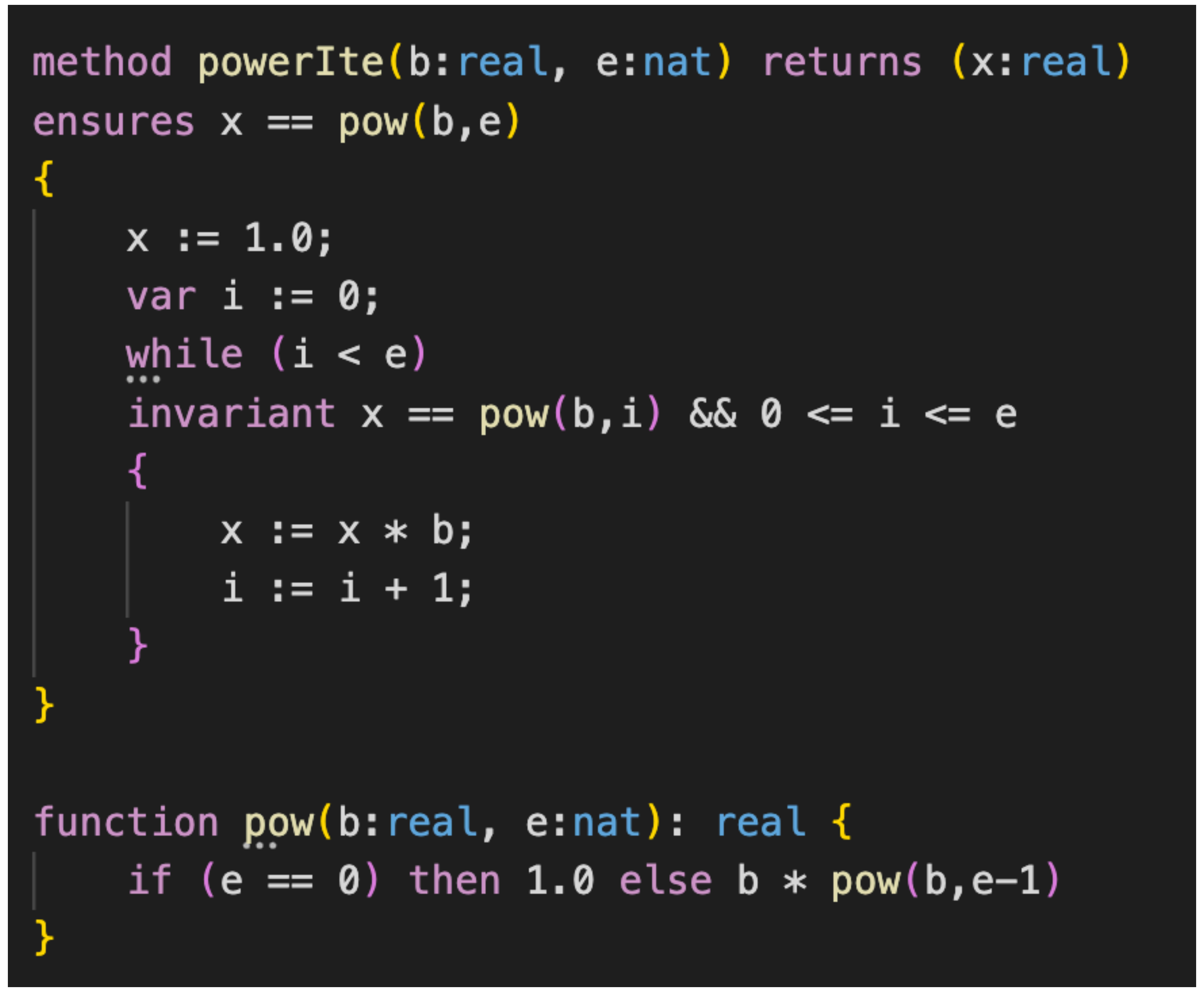


```
method powerIte(b:real, e:nat) returns (x:real)
{
    x := 1.0;
    var i := 0;
    while (i < e)
    {
        x := x * b;
        i
    }
}

function pow(b:real, e:nat): real {
    if (e == 0) then 1.0 else b * pow(b,e-1)
}
```

```
auto-accumulator tail recursive Resolver
decreases b, e Resolver
No quick fixes available
```

Example: Integer Exponentiation

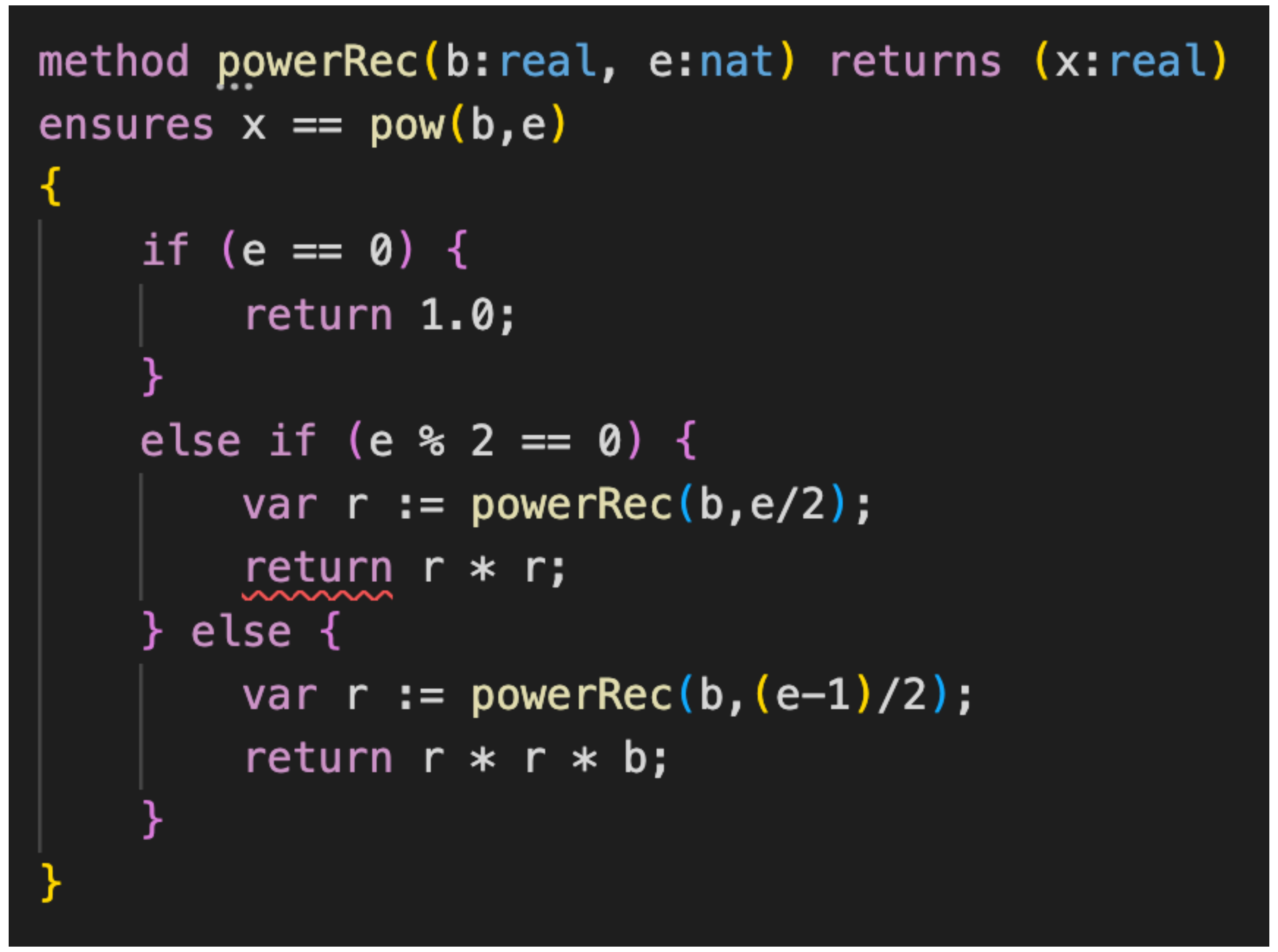
- Proved that the imperative implementation conforms to the functional specification



```
method powerIte(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    x := 1.0;
    var i := 0;
    while (i < e)
    invariant x == pow(b,i) && 0 <= i <= e
    {
        x := x * b;
        i := i + 1;
    }
}

function pow(b:real, e:nat): real {
    if (e == 0) then 1.0 else b * pow(b,e-1)
}
```

Example: Integer Exponentiation

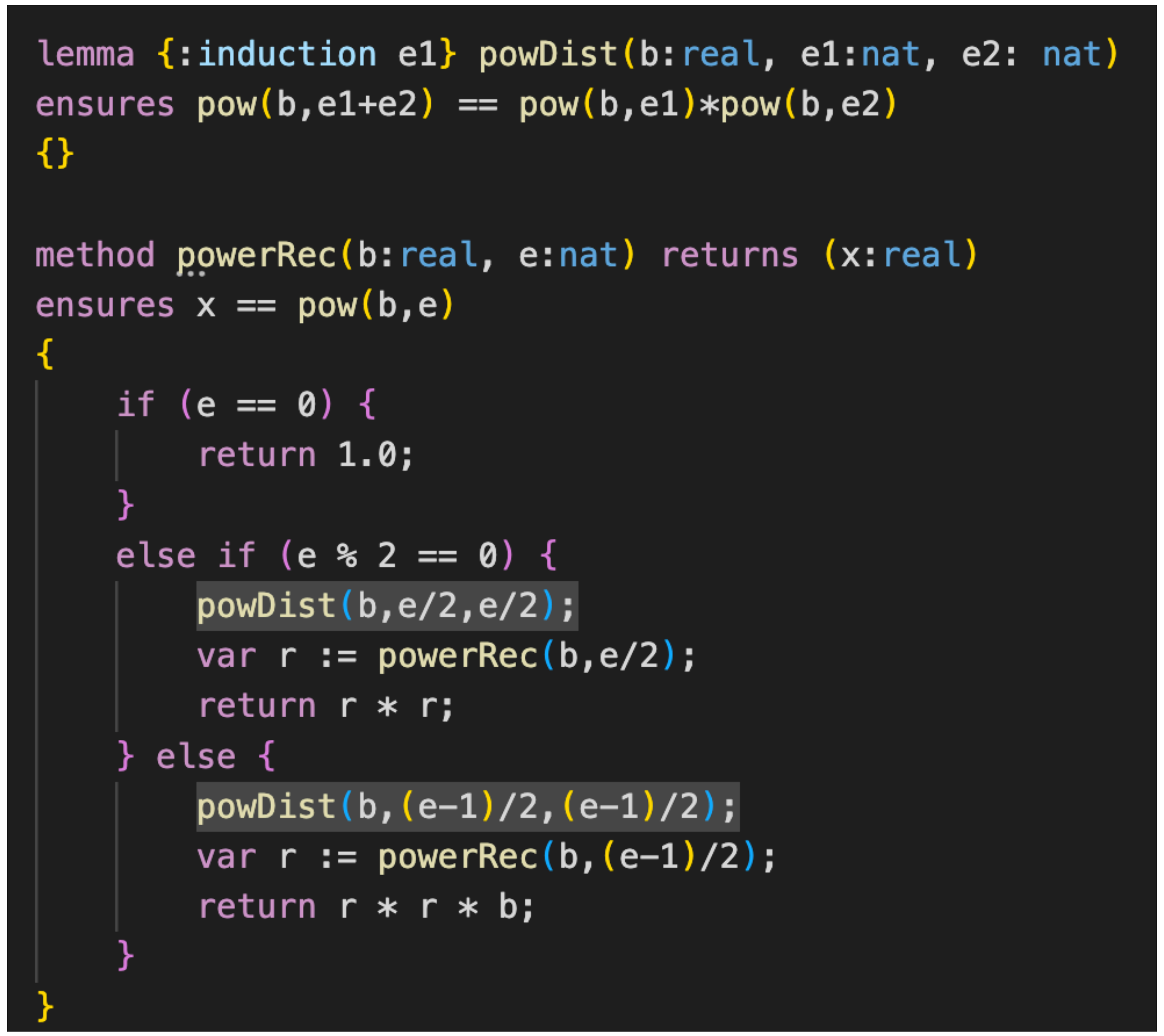
- What if we want a more efficient implementation?
- Recursive version, but could be iterative
- Logarithmic, based on $a^n \times a^m = a^{n+m}$
- Solvers don't know all theorems!
- Requires user input
- Not trivial to identify what lemma is needed (solving process is black box)



```
method powerRec(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    if (e == 0) {
        return 1.0;
    }
    else if (e % 2 == 0) {
        var r := powerRec(b,e/2);
        return r * r;
    } else {
        var r := powerRec(b,(e-1)/2);
        return r * r * b;
    }
}
```

Example: Integer Exponentiation

- Sometimes hints can be provided as assertions
- Force the solver to calculate intermediary results
- In more complex cases, we need to define auxiliary lemmas
- The lemmas themselves must be proved correct: not always trivial!
- Here, instructed to prove with induction

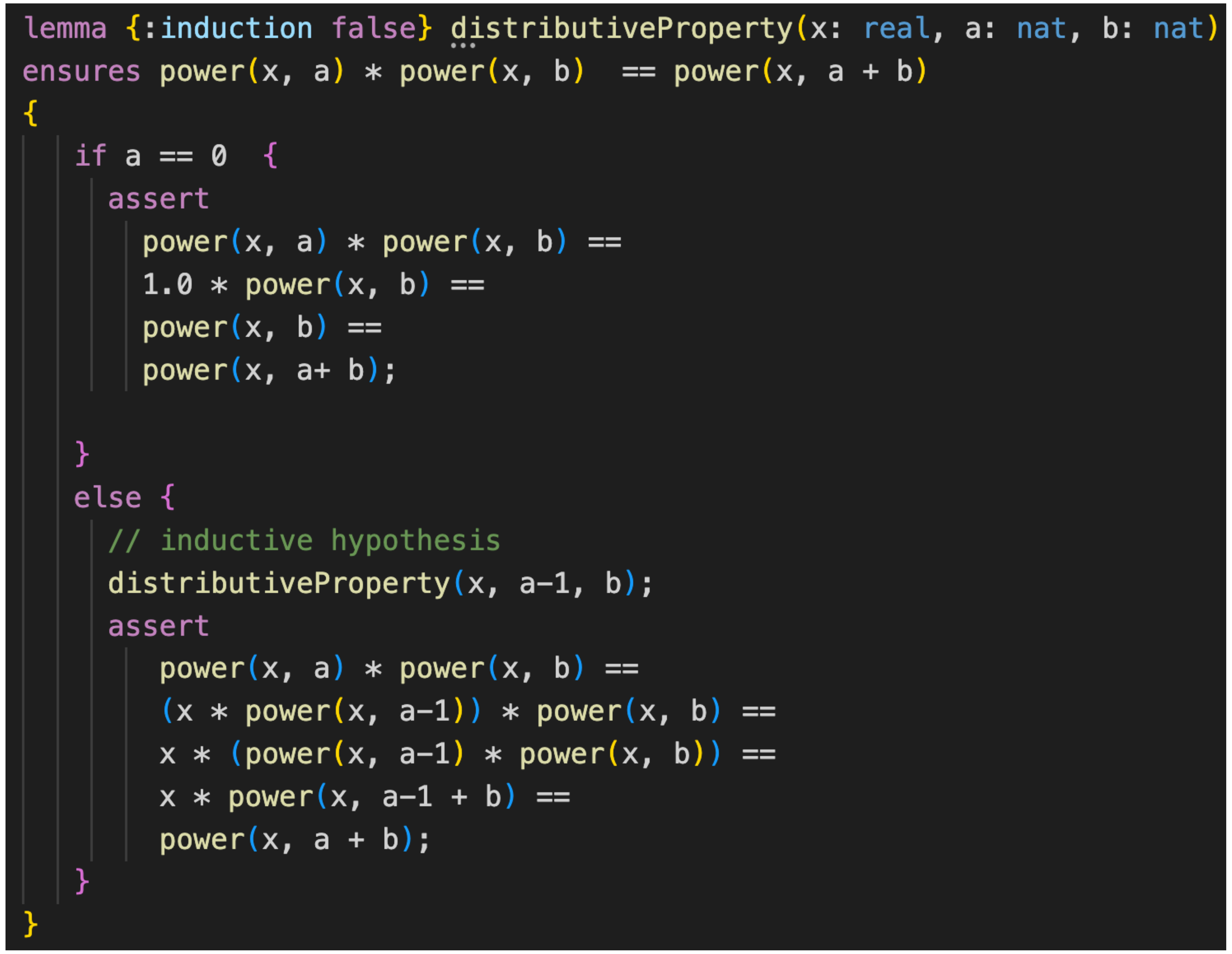


```
lemma {:induction e1} powDist(b:real, e1:nat, e2: nat)
ensures pow(b,e1+e2) == pow(b,e1)*pow(b,e2)
{}

method powerRec(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    if (e == 0) {
        return 1.0;
    }
    else if (e % 2 == 0) {
        powDist(b,e/2,e/2);
        var r := powerRec(b,e/2);
        return r * r;
    } else {
        powDist(b,(e-1)/2,(e-1)/2);
        var r := powerRec(b,(e-1)/2);
        return r * r * b;
    }
}
```

Example: Integer Exponentiation

- Proofs in Dafny are done by showing intermediate steps using `asserts`

- Often by induction, base and inductive case



```
lemma {:induction false} distributiveProperty(x: real, a: nat, b: nat)
ensures power(x, a) * power(x, b)  == power(x, a + b)
{
  if a == 0  {
    assert
      power(x, a) * power(x, b) ==
      1.0 * power(x, b) ==
      power(x, b) ==
      power(x, a+ b);

  }
  else {
    // inductive hypothesis
    distributiveProperty(x, a-1, b);
    assert
      power(x, a) * power(x, b) ==
      (x * power(x, a-1)) * power(x, b) ==
      x * (power(x, a-1) * power(x, b)) ==
      x * power(x, a-1 + b) ==
      power(x, a + b);
  }
}
```

`{:induction false}` turns off automatic proofs by induction.

Complex Types

- Up until now, we’ve dealt with atomic values
- To handle more complex structures, we need reference types
- Pointers to memory sections, may be shared
- Values are no longer immutable, may have side effects

Arrays

- Typical concept in programming languages, used at implementation level
- Are typed, can be generic or concrete
- Declared with `array<T>`, creates object in memory, returns a pointer
- Instances created with new `T[n]`, initialised as `new T[n]`$[a_1, \ldots, a_n]$
- Library of operations (see reference material)
- Calls are always tested to be within bounds and no null points
- By default, non-null arrays (use `array?` to allow nulls)

Collections

- Specifications can’t handle arrays
- Need more abstract types, mathematical, higher-level
- Back to value types: immutable, comparison is content, assignment is copy
- Efficiency of data structures irrelevant, not executed
- Available **collections** (see reference material for operations):
    - Sequence
    - Set
    - Multiset
    - Map
    - String

Annotating Code over Arrays

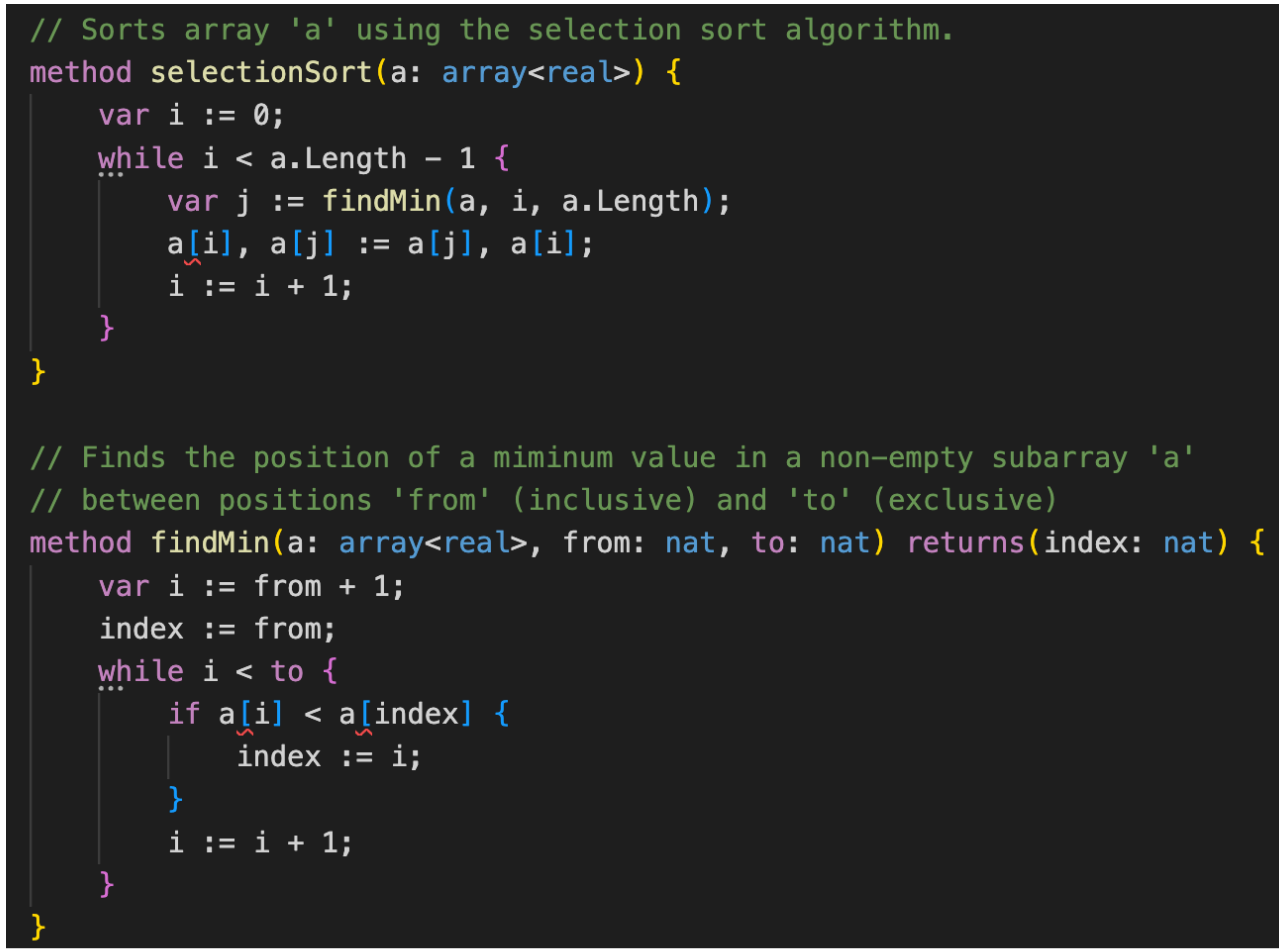
- We cannot reason about arrays, must convert to a collection
- E.g., array to sequence through slices `a[m..n]`
- Can use `forall`/`exists` *quantifications* to iterate over structures
- Variables must be *bounded* (e.g., indices of an array)

Annotating Code with Side-effects

- Dafny may need the *frames* of certain procedures
- A `method` must declare the shared memory it `modifies` (can read everything)
- A `function` must declare what it `reads` (cannot modify anything)
- Postconditions may refer to the memory state before execution (`old(..)`)
- Loop invariants can also refer to the state in the previous iteration

Example: Selection Sort

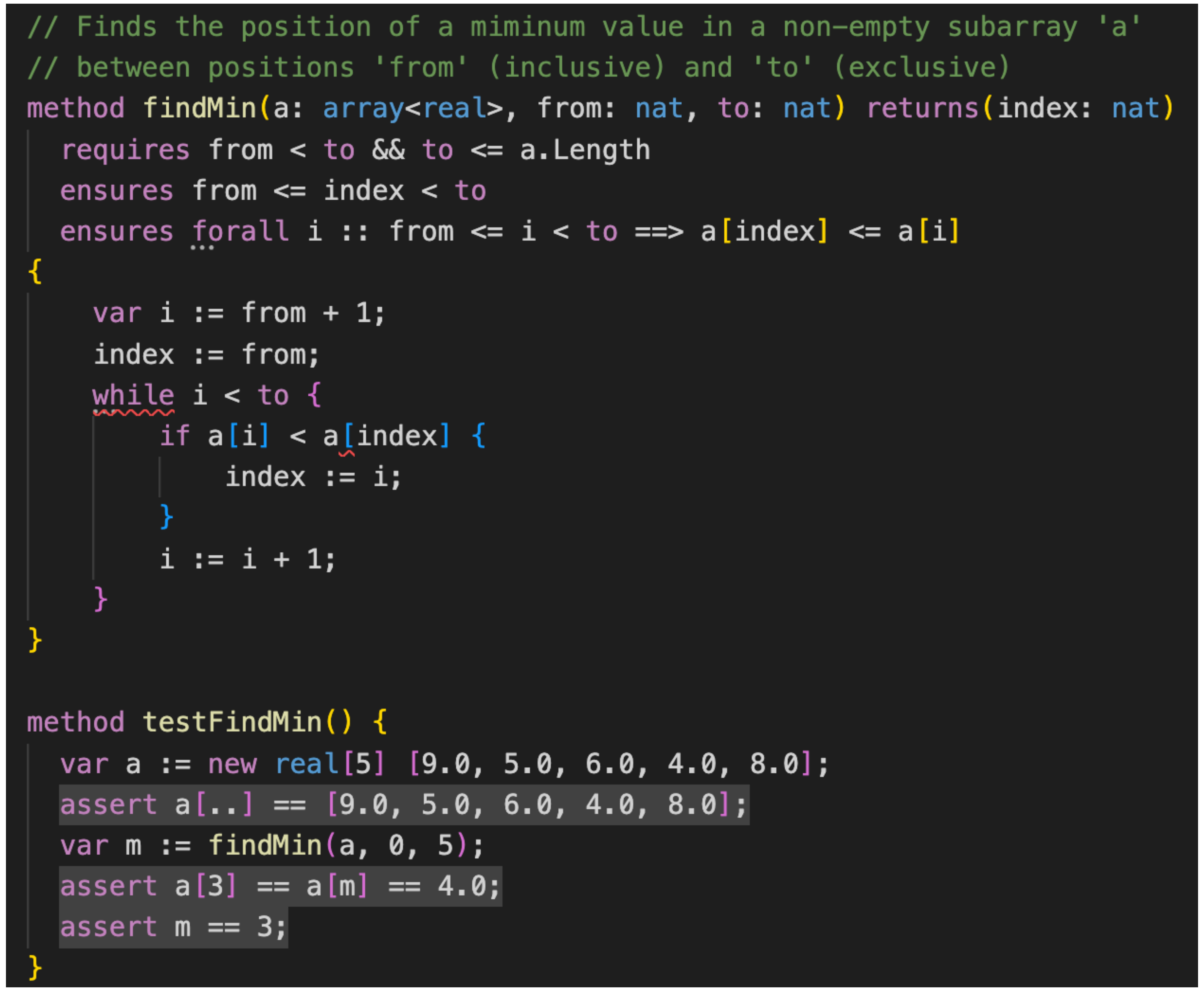
- Implementation of bubble sort, auxiliary function finds minimum in segment
- https://en.wikipedia.org/wiki/Selection_sort
- Without any spec, already errors thrown: *index out of range*
- If array had type `array?`, additional error: *may be null*
- Also says the array cannot be updated by `selectionSort` (frame conditions)



```
// Sorts array 'a' using the selection sort algorithm.
method selectionSort(a: array<real>) {
    var i := 0;
    while i < a.Length - 1 {
        var j := findMin(a, i, a.Length);
        a[i], a[j] := a[j], a[i];
        i := i + 1;
    }
}

// Finds the position of a miminum value in a non-empty subarray 'a'
// between positions 'from' (inclusive) and 'to' (exclusive)
method findMin(a: array<real>, from: nat, to: nat) returns(index: nat) {
    var i := from + 1;
    index := from;
    while i < to {
        if a[i] < a[index] {
            index := i;
        }
        i := i + 1;
    }
}
```

Example: Selection Sort

- Pre-condition: guarantee that accesses to array are valid
- How to encode postcondition of `findMin`?
- Value at found index less than *all* all the others
- Universal quantification `forall`, must be bounded in range
- No need to specify the content to the array: *cannot be changed* unless stated in `modifies`
- Assert in the test needed to guide the solver: not always trivial to identify

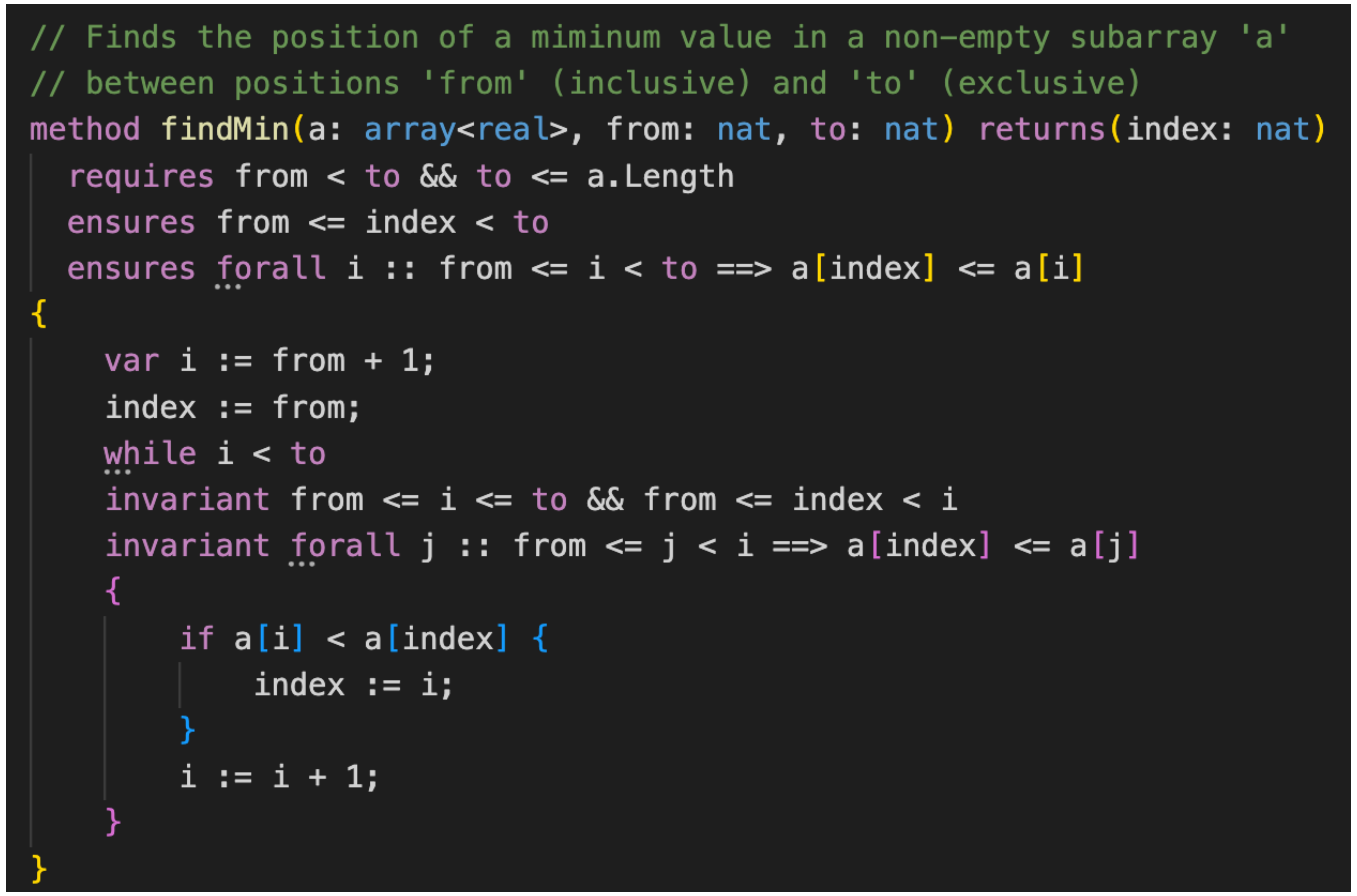


```
// Finds the position of a miminum value in a non-empty subarray 'a'
// between positions 'from' (inclusive) and 'to' (exclusive)
method findMin(a: array<real>, from: nat, to: nat) returns(index: nat)
  requires from < to && to <= a.Length
  ensures from <= index < to
  ensures forall i :: from <= i < to ==> a[index] <= a[i]
{
    var i := from + 1;
    index := from;
    while i < to {
        if a[i] < a[index] {
            index := i;
        }
        i := i + 1;
    }
}

method testFindMin() {
  var a := new real[5] [9.0, 5.0, 6.0, 4.0, 8.0];
  assert a[..] == [9.0, 5.0, 6.0, 4.0, 8.0];
  var m := findMin(a, 0, 5);
  assert a[3] == a[m] == 4.0;
  assert m == 3;
}
```

Multiple `ensures` / `requires` are conjuncted.

Example: Selection Sort

- Same quantifications can be used in loop invariants
- Loop variant automatically detected
- Again, must lead into postcondition

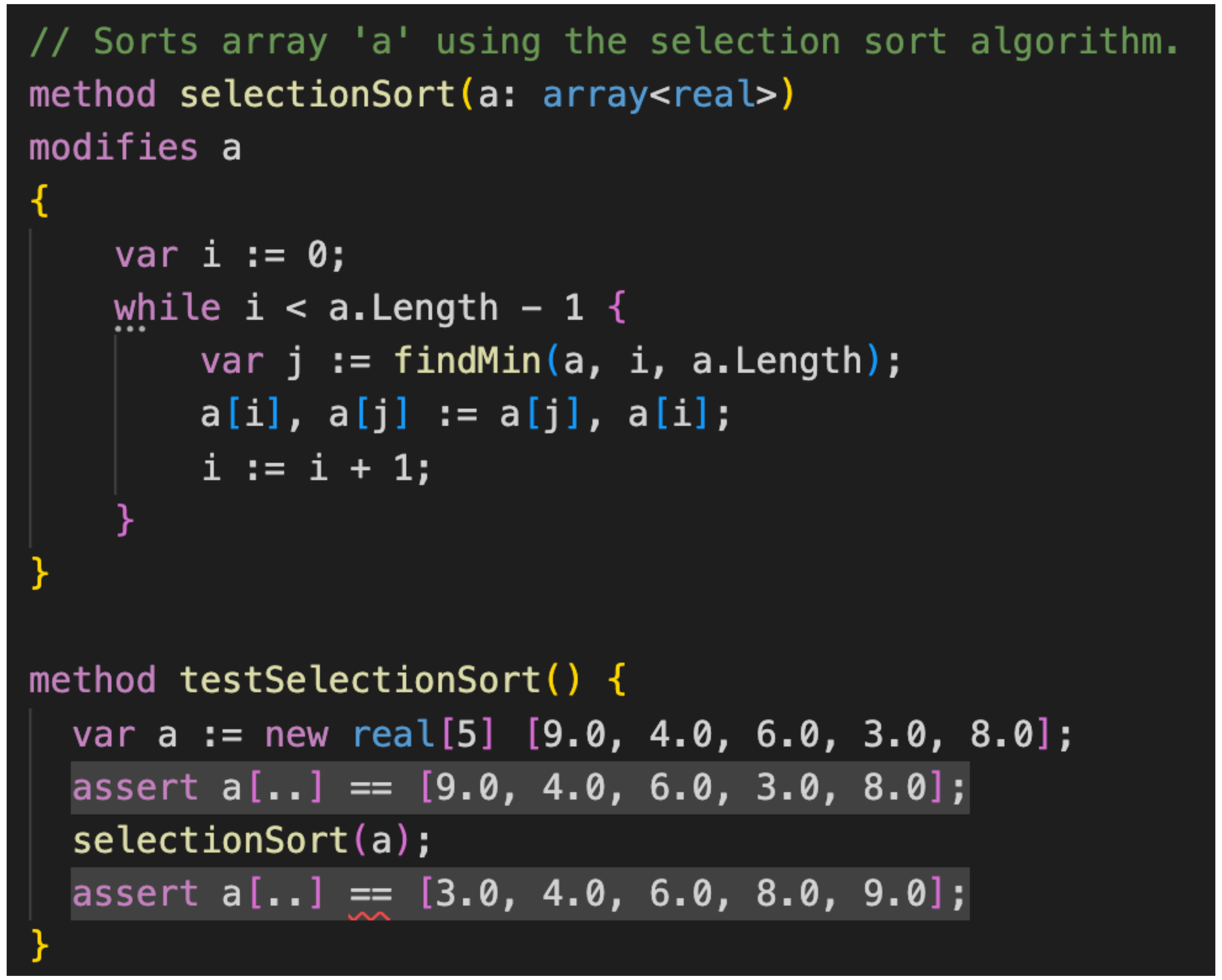


```
// Finds the position of a miminum value in a non-empty subarray 'a'
// between positions 'from' (inclusive) and 'to' (exclusive)
method findMin(a: array<real>, from: nat, to: nat) returns(index: nat)
  requires from < to && to <= a.Length
  ensures from <= index < to
  ensures forall i :: from <= i < to ==> a[index] <= a[i]
{
    var i := from + 1;
    index := from;
    while i < to
    invariant from <= i <= to && from <= index < i
    invariant forall j :: from <= j < i ==> a[index] <= a[j]
    {
        if a[i] < a[index] {
            index := i;
        }
        i := i + 1;
    }
}
```

Multiple `invariant` are conjuncted.

Example: Selection Sort

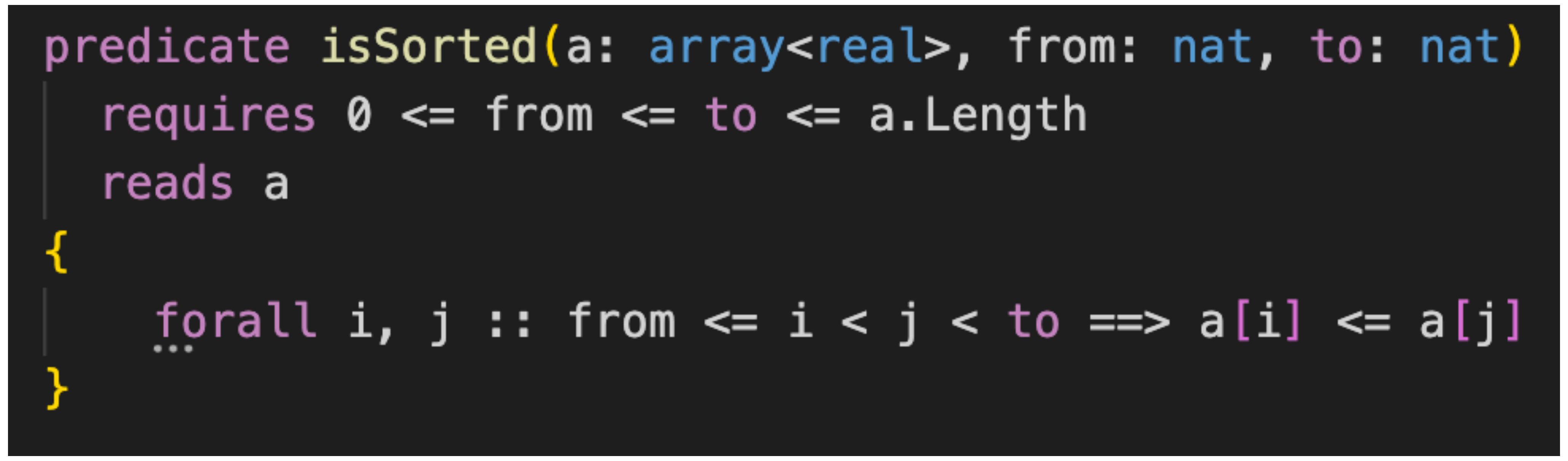
- Sort must be allowed to modify the array
- Sorting guarantees:
  - Elements are ordered
  - Elements are the same (permutation)



```
// Sorts array 'a' using the selection sort algorithm.
method selectionSort(a: array<real>)
modifies a
{
    var i := 0;
    while i < a.Length - 1 {
        var j := findMin(a, i, a.Length);
        a[i], a[j] := a[j], a[i];
        i := i + 1;
    }
}

method testSelectionSort() {
  var a := new real[5] [9.0, 4.0, 6.0, 3.0, 8.0];
  assert a[..] == [9.0, 4.0, 6.0, 3.0, 8.0];
  selectionSort(a);
  assert a[..] == [3.0, 4.0, 6.0, 8.0, 9.0];
}
```

Example: Selection Sort

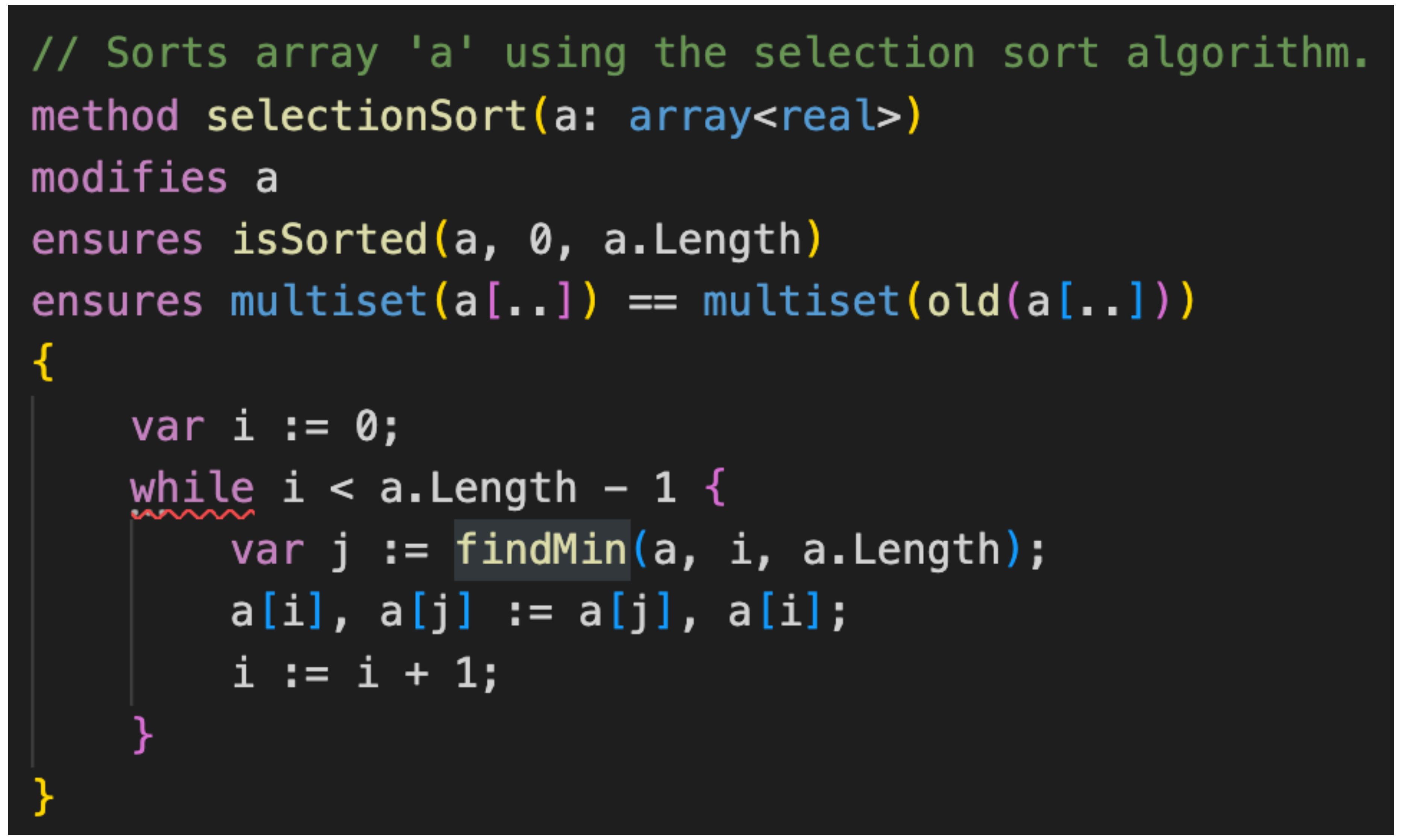
- We can use auxiliary `predicates` (similar to functions, by return Booleans)
- Functions/methods must also declare what the *read*



```
predicate isSorted(a: array<real>, from: nat, to: nat)
  requires 0 <= from <= to <= a.Length
  reads a
{
    forall i, j :: from <= i < j < to ==> a[i] <= a[j]
}
```

Example: Selection Sort

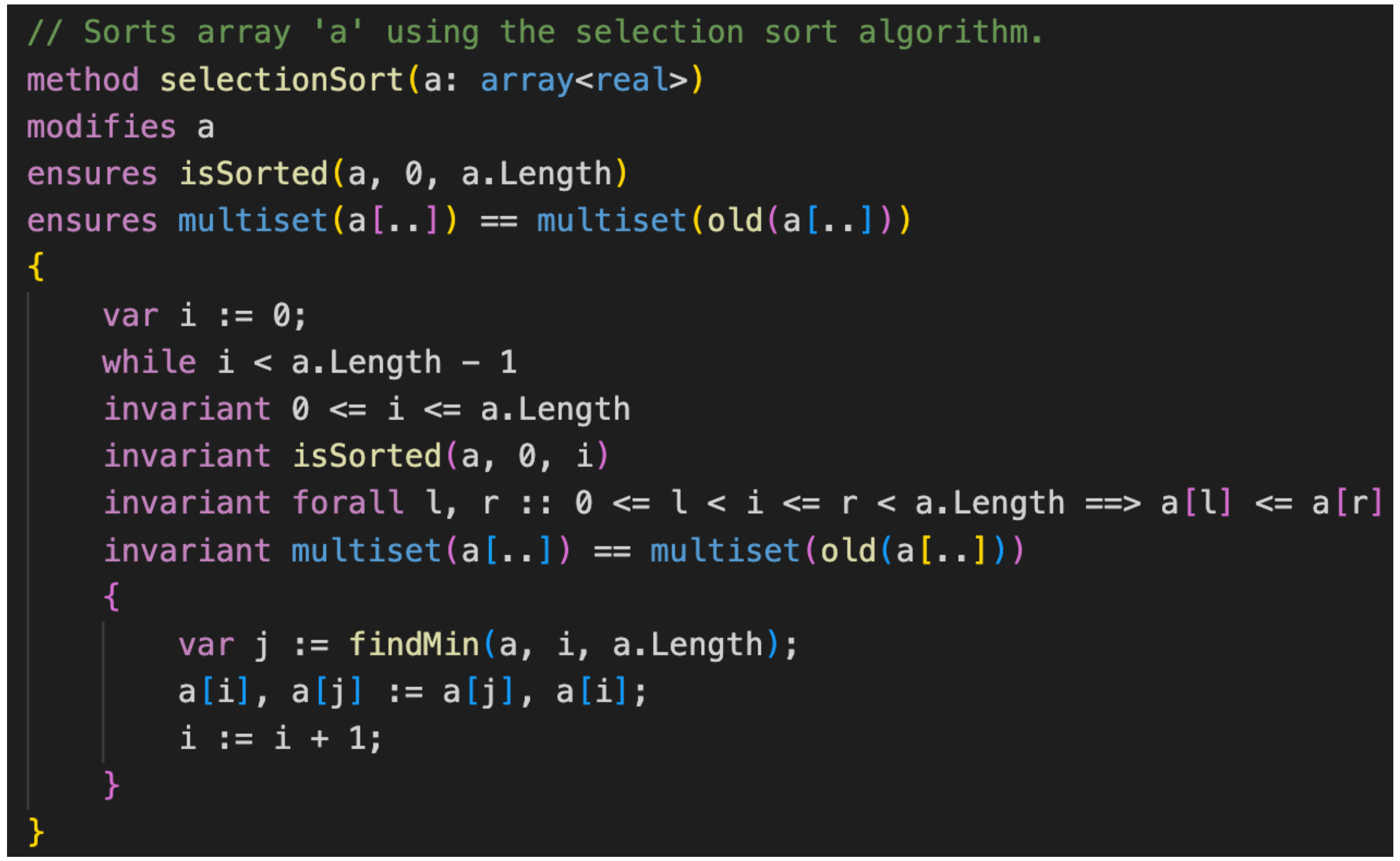
- To define the permutation, we have to refer to the *initial* value of the array
- The *multiset* collection tests if the same values appear in the same quantity
- First convert to sequence with `[..]` slice
- They can be compared directly: value types, compares content, not pointer



```
// Sorts array 'a' using the selection sort algorithm.
method selectionSort(a: array<real>)
modifies a
ensures isSorted(a, 0, a.Length)
ensures multiset(a[..]) == multiset(old(a[..]))
{
    var i := 0;
    while i < a.Length - 1 {
        var j := findMin(a, i, a.Length);
        a[i], a[j] := a[j], a[i];
        i := i + 1;
    }
}
```

Example: Selection Sort

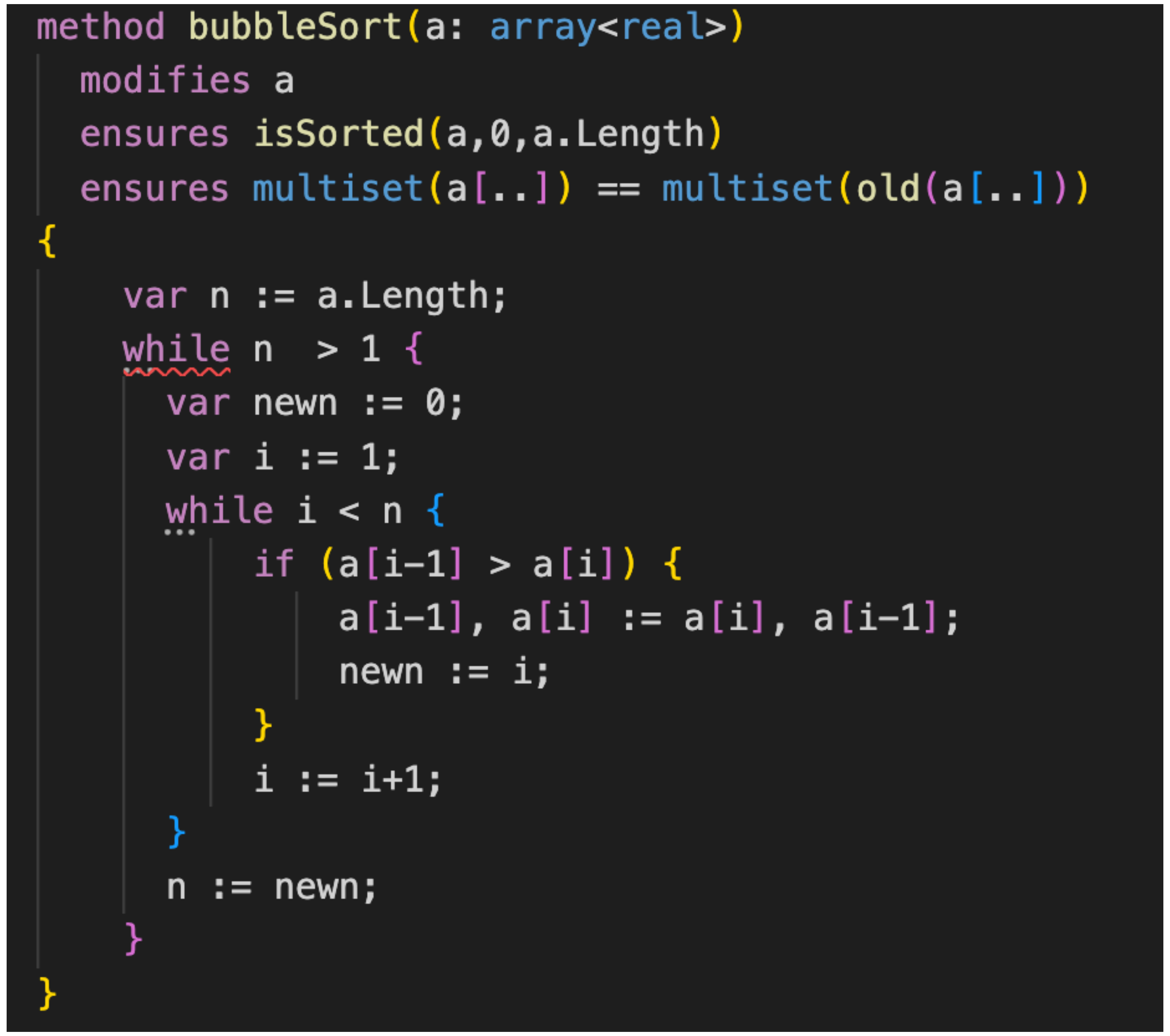
- Invariants can use the same operators as postconditions
- Describe changes in array at each iteration



```
// Sorts array 'a' using the selection sort algorithm.
method selectionSort(a: array<real>)
modifies a
ensures isSorted(a, 0, a.Length)
ensures multiset(a[..]) == multiset(old(a[..]))
{
    var i := 0;
    while i < a.Length - 1
    invariant 0 <= i <= a.Length
    invariant isSorted(a, 0, i)
    invariant forall l, r :: 0 <= l < i <= r < a.Length ==> a[l] <= a[r]
    invariant multiset(a[..]) == multiset(old(a[..]))
    {
        var j := findMin(a, i, a.Length);
        a[i], a[j] := a[j], a[i];
        i := i + 1;
    }
}
```

Example: Bubble Sort

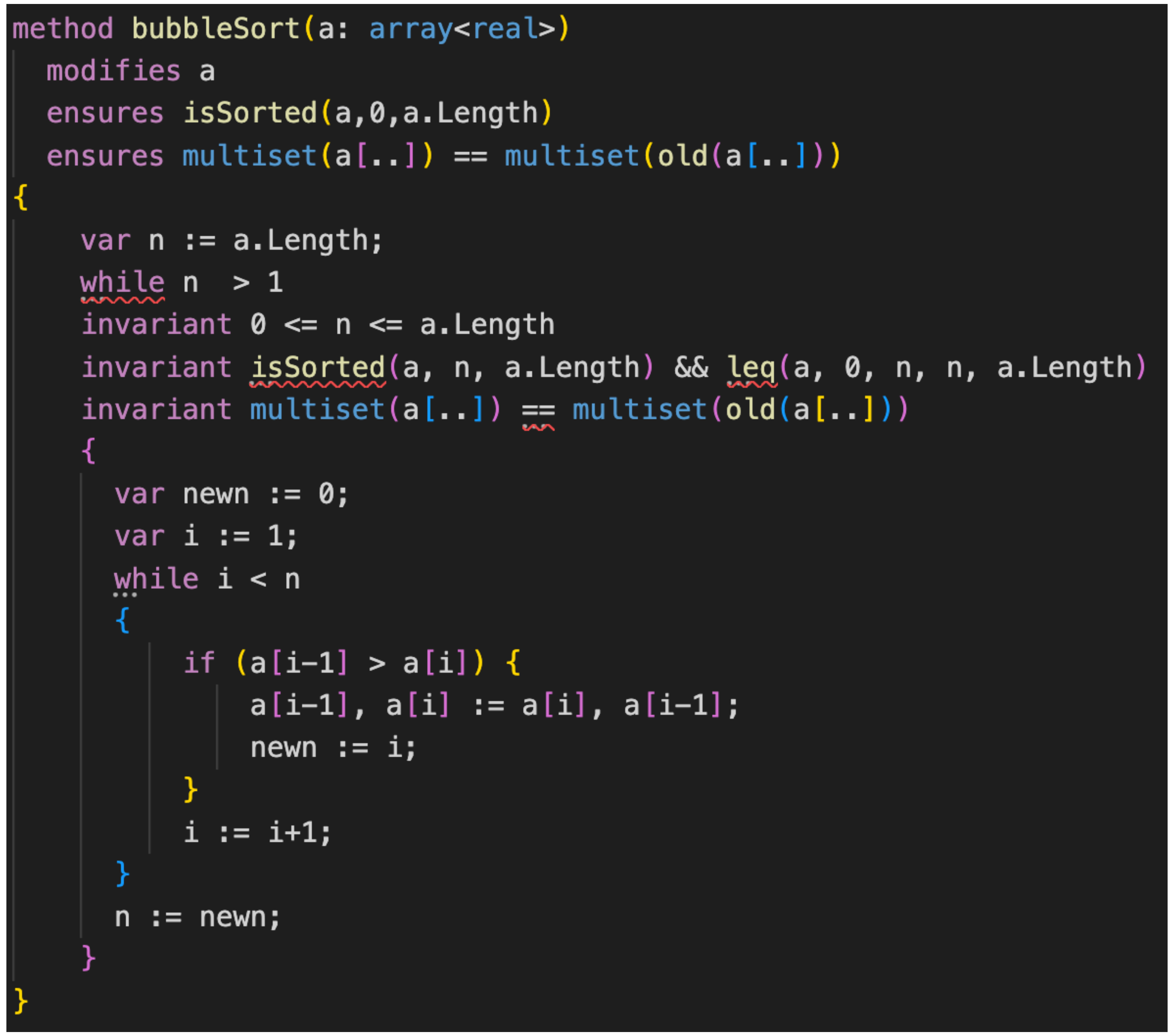
- Same precondition and postcondition (does the same!)
- However, now we have two nested loops
- The outer loop must guarantee into postcondition
- The inner loop must lead guarantee outer loop's invariant
- https://en.wikipedia.org/wiki/Bubble_sort



```
method bubbleSort(a: array<real>)
  modifies a
  ensures isSorted(a,0,a.Length)
  ensures multiset(a[..]) == multiset(old(a[..]))
{
    var n := a.Length;
    while n  > 1 {
      var newn := 0;
      var i := 1;
      while i < n {
        if (a[i-1] > a[i]) {
          a[i-1], a[i] := a[i], a[i-1];
          newn := i;
        }
        i := i+1;
      }
      n := newn;
    }
}
```

Example: Bubble Sort

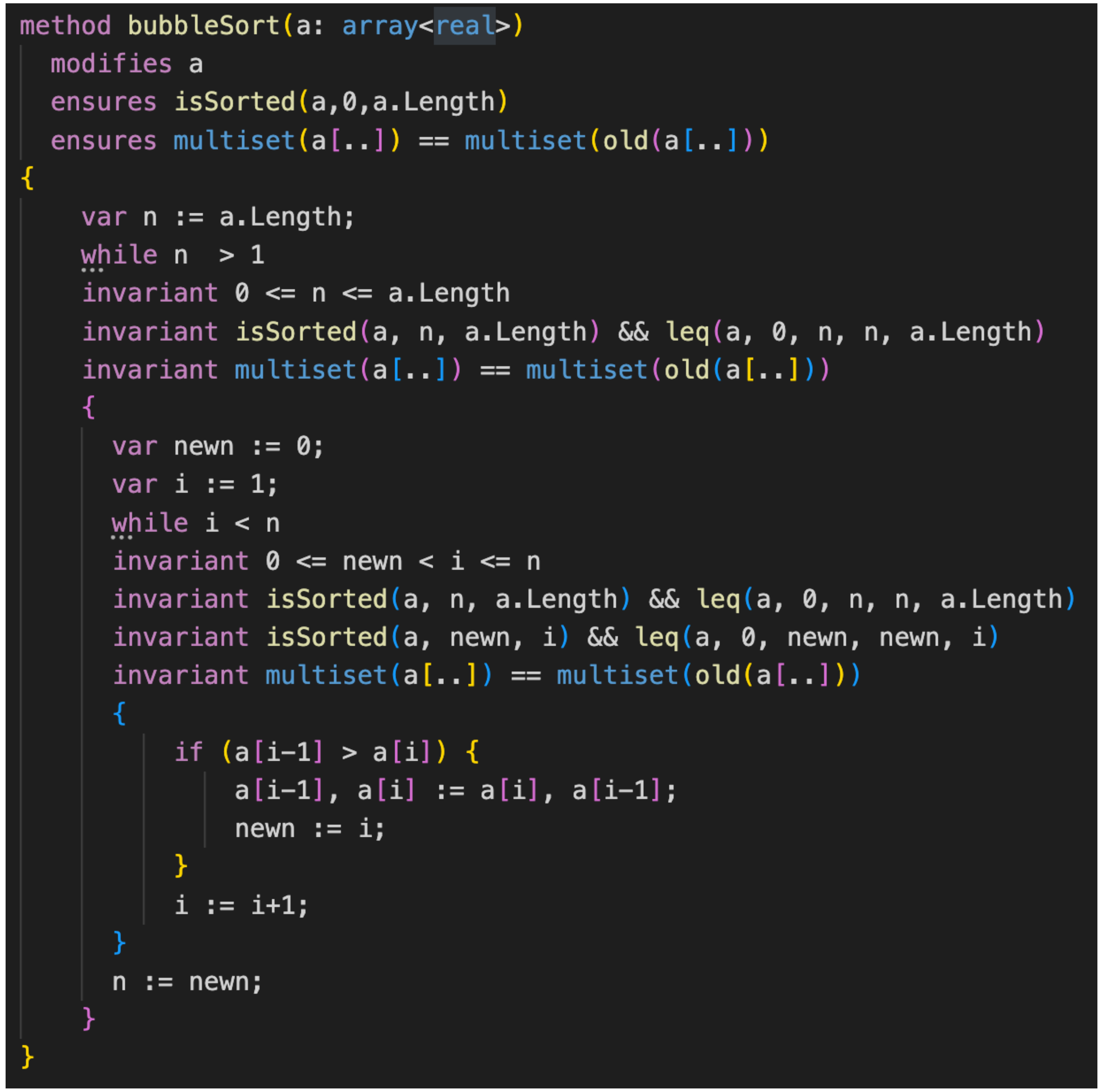
- To verify complex algorithms, one fully understand its behaviour
- Outer loop: keeps pushing larger values up to an already ordered segment
- This right segment is ordered, and any value in the unordered left segment is smaller



```
method bubbleSort(a: array<real>)
  modifies a
  ensures isSorted(a,0,a.Length)
  ensures multiset(a[..]) == multiset(old(a[..]))
{
    var n := a.Length;
    while n  > 1
    invariant 0 <= n <= a.Length
    invariant isSorted(a, n, a.Length) && leq(a, 0, n, n, a.Length)
    invariant multiset(a[..]) == multiset(old(a[..]))
    {
      var newn := 0;
      var i := 1;
      while i < n
      {
          if (a[i-1] > a[i]) {
              a[i-1], a[i] := a[i], a[i-1];
              newn := i;
          }
          i := i+1;
      }
      n := newn;
    }
}
```

Example: Bubble Sort

- Inner loop: goes through the left segment and swaps
- Registers last swapped position
- Three segments in left segment: before last swap, between last swap and current position, beyond current position
- Important: doesn't break the restrictions between left and right segment required by outer loop



```
method bubbleSort(a: array<real>)
  modifies a
  ensures isSorted(a,0,a.Length)
  ensures multiset(a[..]) == multiset(old(a[..]))
{
    var n := a.Length;
    while n  > 1
    invariant 0 <= n <= a.Length
    invariant isSorted(a, n, a.Length) && leq(a, 0, n, n, a.Length)
    invariant multiset(a[..]) == multiset(old(a[..]))
    {
      var newn := 0;
      var i := 1;
      while i < n
      invariant 0 <= newn < i <= n
      invariant isSorted(a, n, a.Length) && leq(a, 0, n, n, a.Length)
      invariant isSorted(a, newn, i) && leq(a, 0, newn, newn, i)
      invariant multiset(a[..]) == multiset(old(a[..]))
      {
          if (a[i-1] > a[i]) {
              a[i-1], a[i] := a[i], a[i-1];
              newn := i;
          }
          i := i+1;
      }
      n := newn;
    }
}
```

Design by Contract

- Approach proposed by Bertrand Meyer for the Eiffel language
- Inspired by Hoare logic, adapted to object-oriented paradigm
- A *contract* is an agreement between a client and a supplier, obligations and benefits
- The client is the caller: obliged to respect *preconditions* (benefit of supplier)
- The supplier is the method: obliged to respect *postconditions* (benefit of client)
- Applied to object-oriented programming: *class invariant* must also be preserved

*“**Correctness** is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.”*

Bertrand Meyer, ACM Software System Award of the ACM for "impact on software quality"

Class Invariants

- Class invariants define the integrity of the state of class instances
- Must hold at all times
- Methods must preserve: assume at entry, guarantee after execution
- Constructor must guarantee after execution
- May be broken during execution

Design by Contract in Dafny

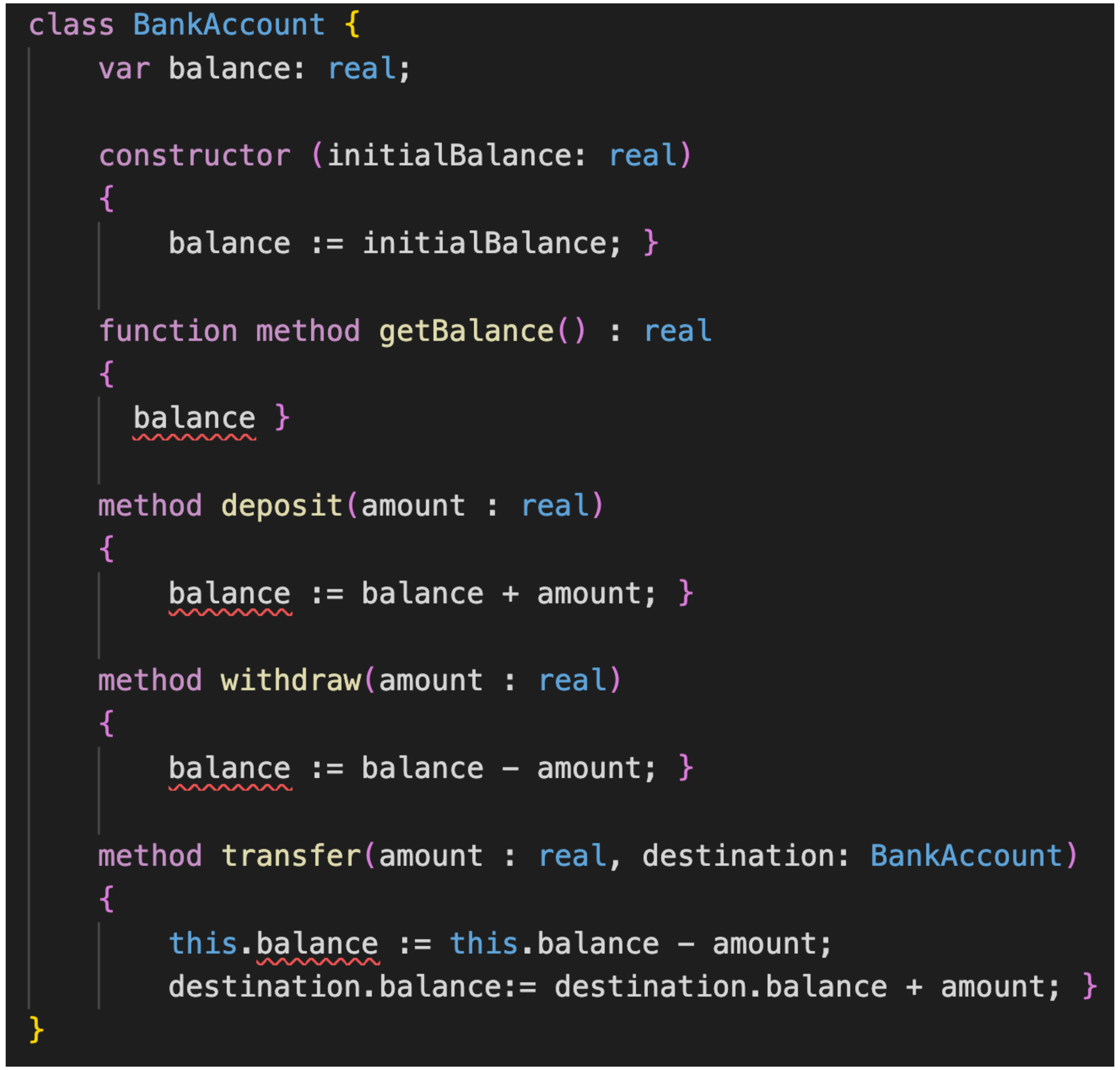
- Dafny has typical features from object-oriented paradigm
- Classes are reference types: methods must declare frame conditions
- Frame conditions at the object level, not fields
- Class methods are annotated with preconditions and postconditions
- Besides input and output, may refer to the class fields
- Postcondition can use `old` for state before execution

Class Invariants in Dafny

- Class invariant is special predicate `Valid()`
- All methods require `Valid()` and ensure `Valid()`
- Constructor must ensure `Valid()`
- `{ :autocontract }` annotation injects these annotations (and frame conditions) automatically

Example: Bank Account

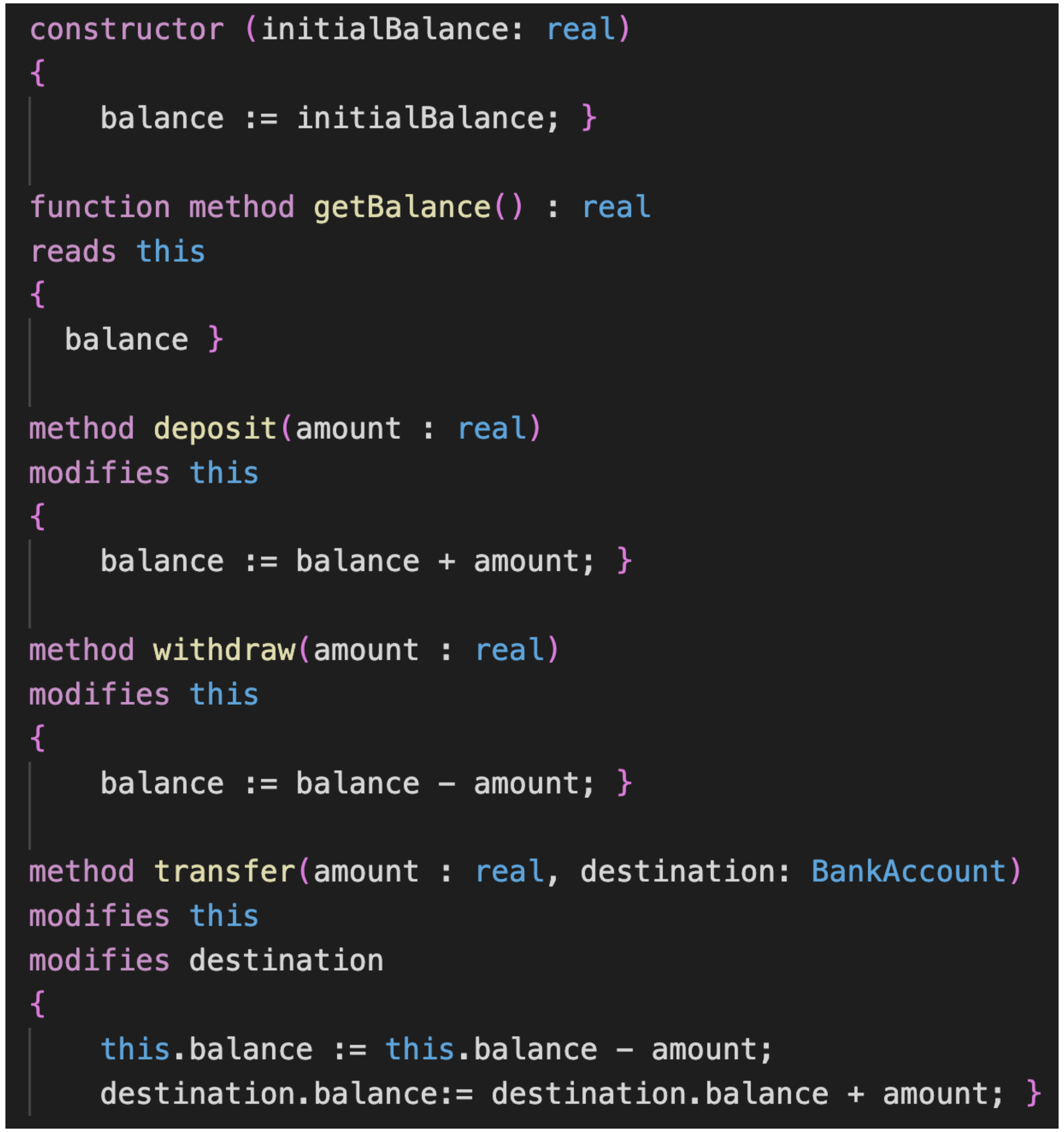
- Classes are typical object-oriented:
  - `constructors`
  - fields (`var`), can be `constants`
  - `methods` (or `function methods`)
  - abstract classes (`trait`)
  - inheritance (`extends`)
- Classes are reference types, need frame conditions



```
class BankAccount {
    var balance: real;

    constructor (initialBalance: real)
    {
        balance := initialBalance; }

    function method getBalance() : real
    {
      balance }

    method deposit(amount : real)
    {
        balance := balance + amount; }

    method withdraw(amount : real)
    {
        balance := balance - amount; }

    method transfer(amount : real, destination: BankAccount)
    {
        this.balance := this.balance - amount;
        destination.balance:= destination.balance + amount; }
}
```

Example: Bank Account

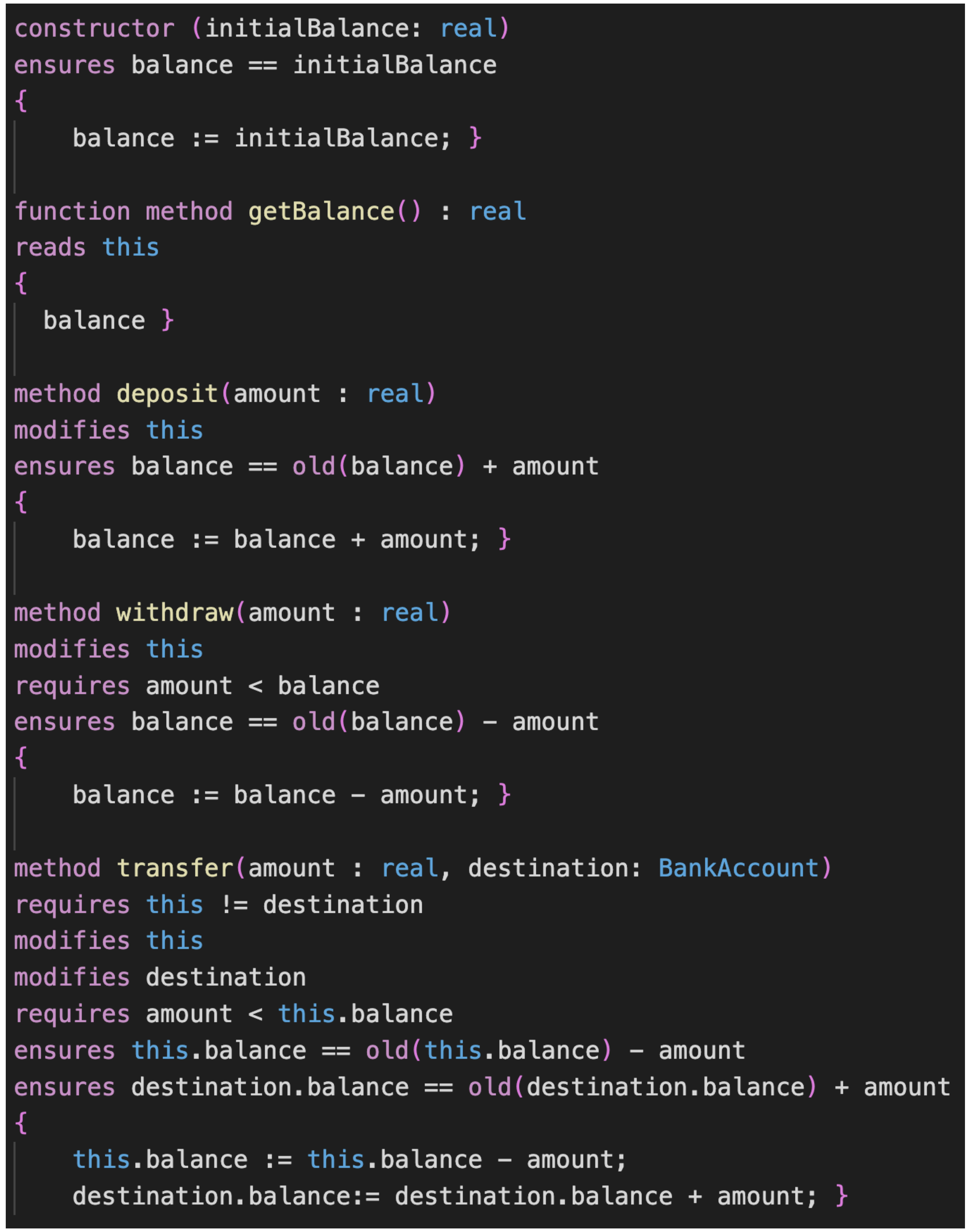
- Frame conditions at the level of the object, not its variables
  - `this` for parent object
- Functions must declare `reads` and methods `modifies`
- Constructor doesn't need frame conditions



```
constructor (initialBalance: real)
{
    balance := initialBalance; }

function method getBalance() : real
reads this
{
  balance }

method deposit(amount : real)
modifies this
{
    balance := balance + amount; }

method withdraw(amount : real)
modifies this
{
    balance := balance - amount; }

method transfer(amount : real, destination: BankAccount)
modifies this
modifies destination
{
    this.balance := this.balance - amount;
    destination.balance:= destination.balance + amount; }
```

Example: Bank Account

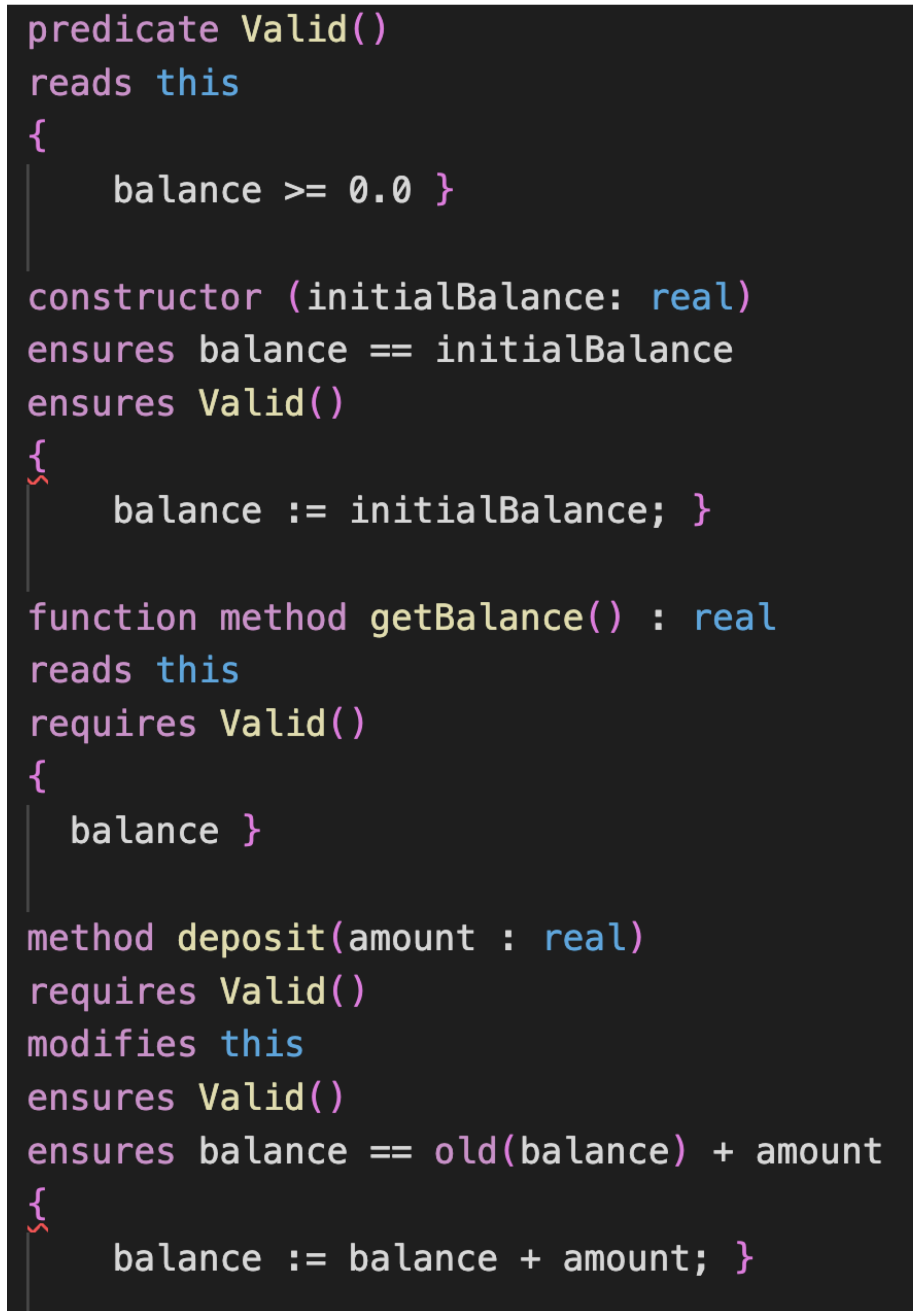
- Code of constructors and methods is still opaque to verification
- Must declare postconditions, even if apparently trivial



```
constructor (initialBalance: real)
ensures balance == initialBalance
{
    balance := initialBalance; }

function method getBalance() : real
reads this
{
  balance }

method deposit(amount : real)
modifies this
ensures balance == old(balance) + amount
{
    balance := balance + amount; }

method withdraw(amount : real)
modifies this
requires amount < balance
ensures balance == old(balance) - amount
{
    balance := balance - amount; }

method transfer(amount : real, destination: BankAccount)
requires this != destination
modifies this
modifies destination
requires amount < this.balance
ensures this.balance == old(this.balance) - amount
ensures destination.balance == old(destination.balance) + amount
{
    this.balance := this.balance - amount;
    destination.balance:= destination.balance + amount; }
```

Example: Bank Account

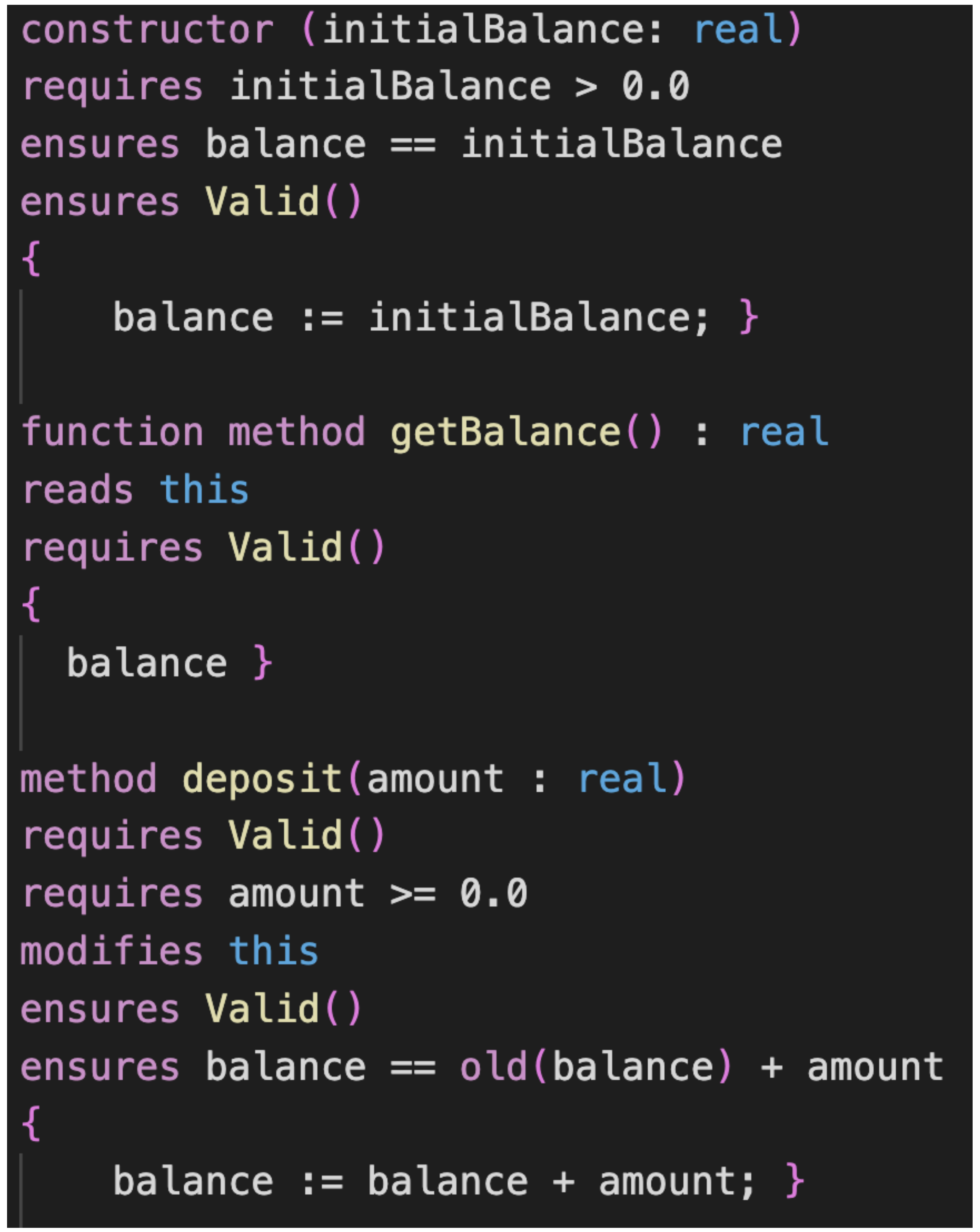
- Class invariant defined as a special predicate `Valid()`
- All methods assume invariant holds and preserve it
- Constructors only have to ensure
- Function methods do not need to ensure it (state not updated)



```
predicate Valid()
reads this
{
    balance >= 0.0 }

constructor (initialBalance: real)
ensures balance == initialBalance
ensures Valid()
{
    balance := initialBalance; }

function method getBalance() : real
reads this
requires Valid()
{
  balance }

method deposit(amount : real)
requires Valid()
modifies this
ensures Valid()
ensures balance == old(balance) + amount
{
    balance := balance + amount; }
```

Example: Bank Account

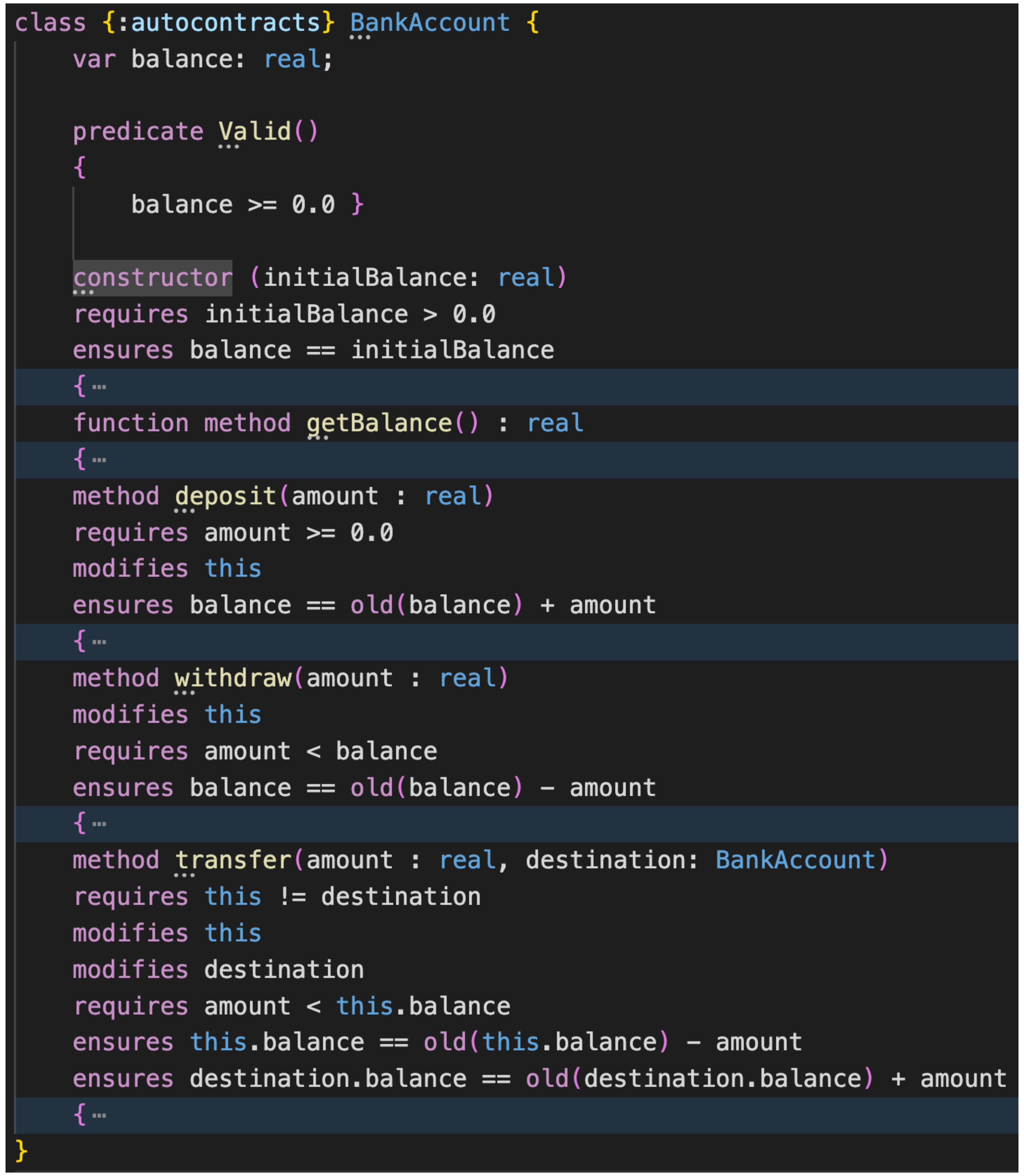
- Preconditions were too weak to preserve invariant
- Recall we don't need the weakest precondition
  - New accounts need positive balance



```
constructor (initialBalance: real)
requires initialBalance > 0.0
ensures balance == initialBalance
ensures Valid()
{
    balance := initialBalance; }

function method getBalance() : real
reads this
requires Valid()
{
  balance }

method deposit(amount : real)
requires Valid()
requires amount >= 0.0
modifies this
ensures Valid()
ensures balance == old(balance) + amount
{
    balance := balance + amount; }
```

Example: Bank Account

- Autocontracts option generates contracts from `Valid()` invariant
  - Remove a needed precondition to check
- Also detects frame conditions
- Other postconditions must still be defined



```
class {:autocontracts} BankAccount {
    var balance: real;

    predicate Valid()
    {
        balance >= 0.0 }

    constructor (initialBalance: real)
    requires initialBalance > 0.0
    ensures balance == initialBalance
    {…
    function method getBalance() : real
    {…
    method deposit(amount : real)
    requires amount >= 0.0
    modifies this
    ensures balance == old(balance) + amount
    {…
    method withdraw(amount : real)
    modifies this
    requires amount < balance
    ensures balance == old(balance) - amount
    {…
    method transfer(amount : real, destination: BankAccount)
    requires this != destination
    modifies this
    modifies destination
    requires amount < this.balance
    ensures this.balance == old(this.balance) - amount
    ensures destination.balance == old(destination.balance) + amount
    {…
}
```

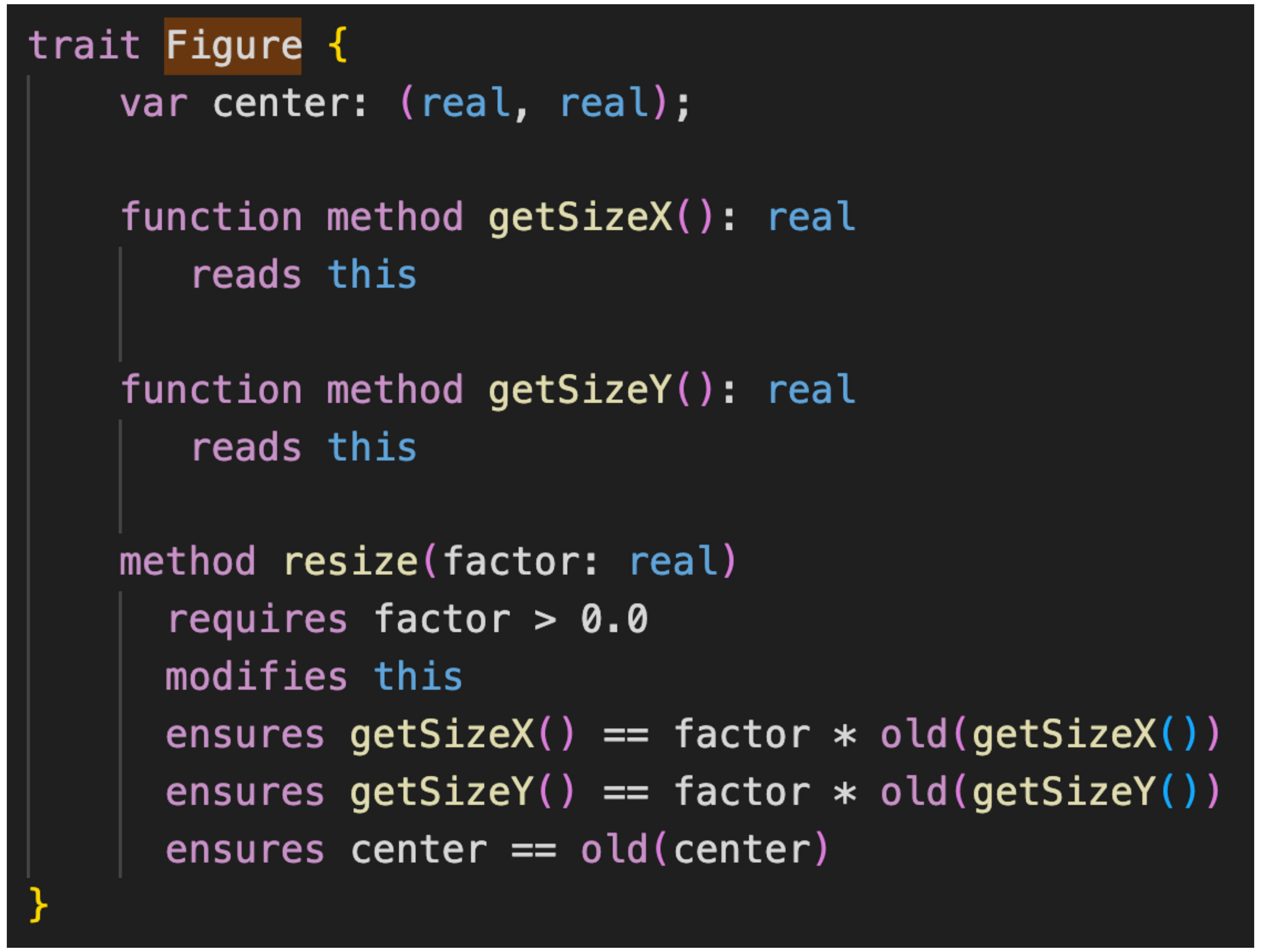
Inheritance

- Based on the concept of *behavioural subtyping*
- Stronger than method subtyping (generalize inputs, restrict outputs)
- Takes into consideration specifications, the contract
    - Preconditions can be weakened
    - Postconditions can be strengthened
    - Invariants must be preserved

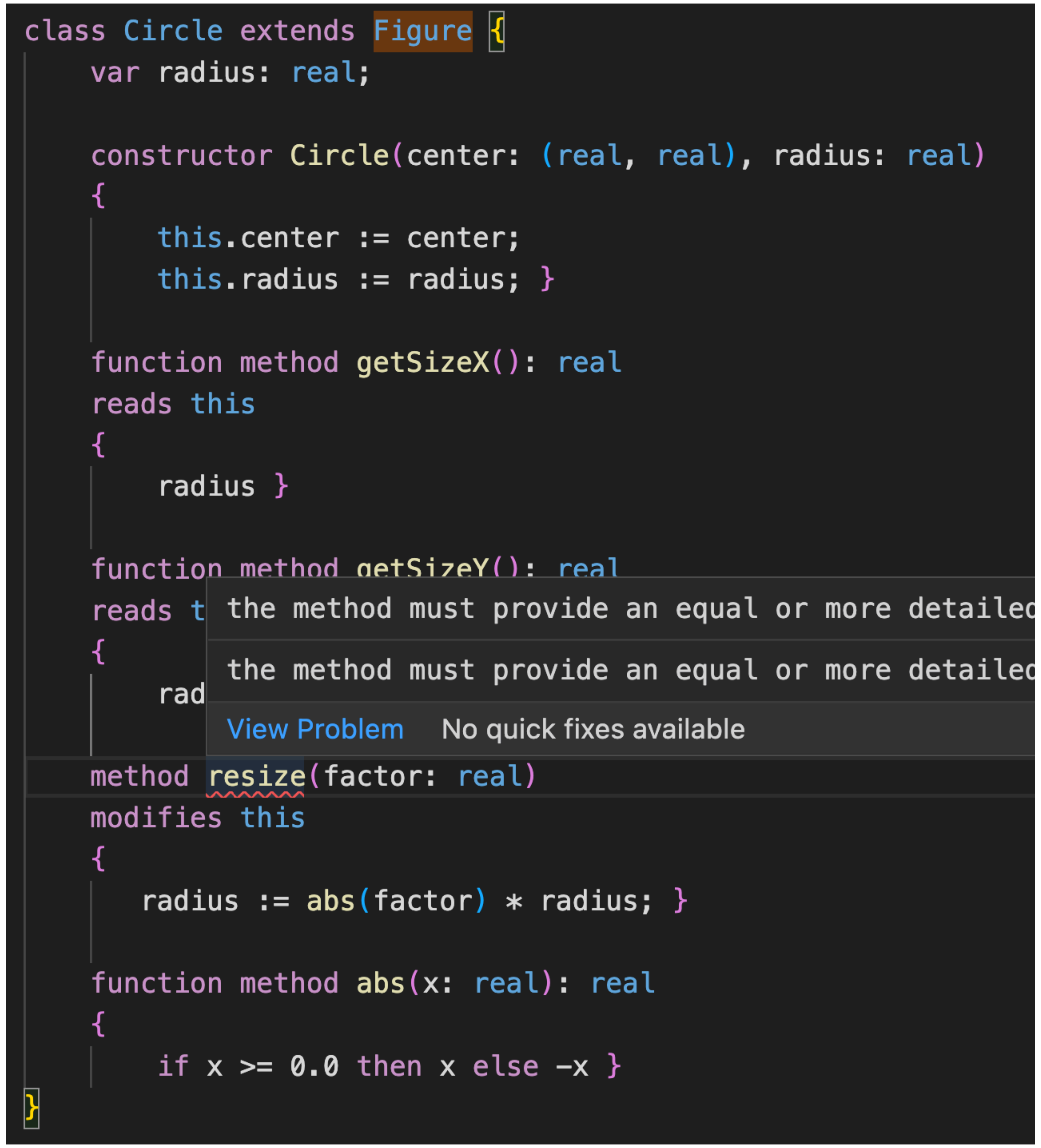
Inheritance in Dafny

- Classes can extend abstract classes (*traits*)
- Subclasses must have:
  - Preconditions that are entailed by the precondition of the superclass
  - Postconditions that entail the postcondition of the superclass
- Subclasses *do not* inherit parent’s annotations (if same, must be rewritten)

Example: Figures



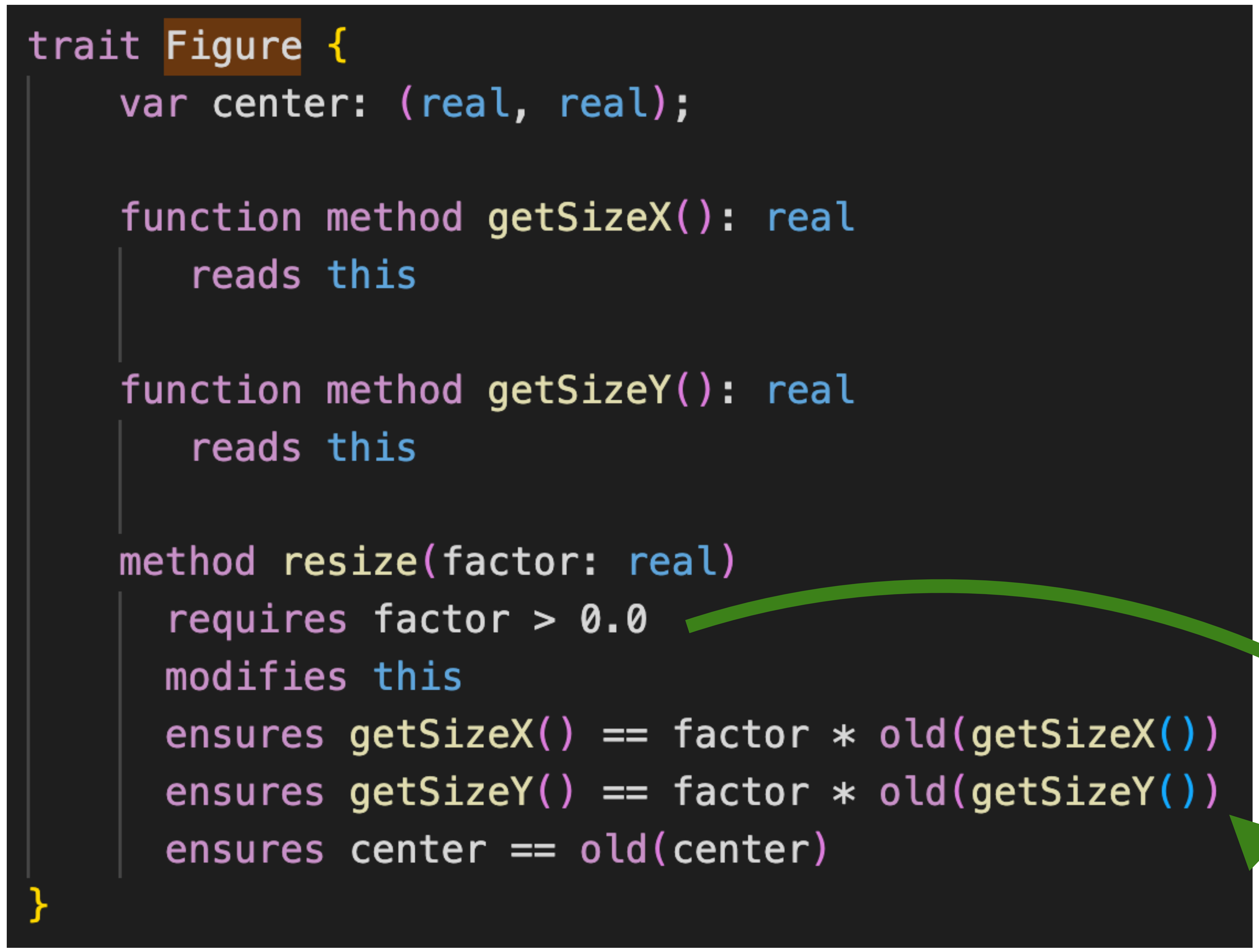
```
trait Figure {
    var center: (real, real);

    function method getSizeX(): real
       reads this

    function method getSizeY(): real
       reads this

    method resize(factor: real)
      requires factor > 0.0
      modifies this
      ensures getSizeX() == factor * old(getSizeX())
      ensures getSizeY() == factor * old(getSizeY())
      ensures center == old(center)
}
```



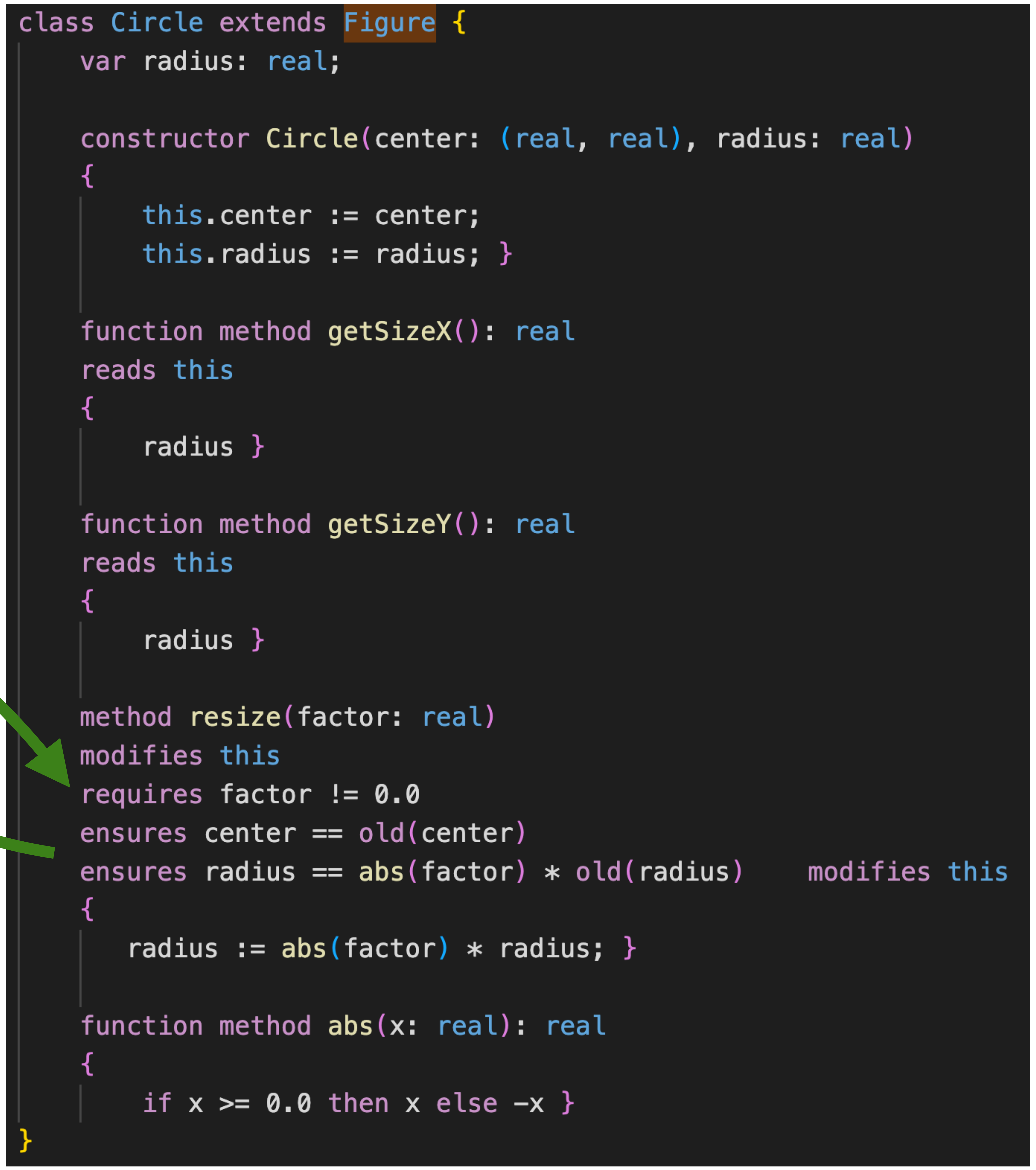
```
class Circle extends Figure {
    var radius: real;

    constructor Circle(center: (real, real), radius: real)
    {
        this.center := center;
        this.radius := radius; }

    function method getSizeX(): real
    reads this
    {
        radius }

    function method getSizeY(): real
    reads t the method must provide an equal or more detailed
    {       the method must provide an equal or more detailed
        rad View Problem    No quick fixes available
    method resize(factor: real)
    modifies this
    {
       radius := abs(factor) * radius; }

    function method abs(x: real): real
    {
        if x >= 0.0 then x else -x }
}
```

- Methods and functions can have undefined bodies in traits

Example: Figures



```
trait Figure {
    var center: (real, real);

    function method getSizeX(): real
       reads this

    function method getSizeY(): real
       reads this

    method resize(factor: real)
      requires factor > 0.0
      modifies this
      ensures getSizeX() == factor * old(getSizeX())
      ensures getSizeY() == factor * old(getSizeY())
      ensures center == old(center)
}
```



```
class Circle extends Figure {
    var radius: real;

    constructor Circle(center: (real, real), radius: real)
    {
        this.center := center;
        this.radius := radius; }

    function method getSizeX(): real
    reads this
    {
        radius }

    function method getSizeY(): real
    reads this
    {
        radius }

    method resize(factor: real)
    modifies this
    requires factor != 0.0
    ensures center == old(center)
    ensures radius == abs(factor) * old(radius)     modifies this
    {
       radius := abs(factor) * radius; }

    function method abs(x: real): real
    {
        if x >= 0.0 then x else -x }
}
```

- Methods in subclasses must have stronger postconditions
- Preconditions cannot be stronger (can be weaker)