The following slides contain advanced material and are optional.
Outline

- Invariants
- Violating the invariant
- Marriage problem
Invariants explained in 60 seconds

- Consistency requirements for a class
- Established after object creation
- Hold, when an object is visible
  - Entry of a routine
  - Exit of a routine

```plaintext
class ACCOUNT
feature
  balance: INTEGER
invariant
  balance >= 0
end
```
Temporary violation

- Invariants can be violated temporarily
  e.g. on object creation

- In Eiffel, invariants are checked on entry and exit of a qualified feature call

- One exception: for calls to creation procedures, invariants are not checked on entry to routine
  - e.g create cell.set_item (1)
  - But checked for normal call: cell.set_item (1)

- See demo.
Public interface of person (without contracts)

```plaintext
class PERSON
  feature
    spouse: PERSON
      -- Spouse of Current.
    marry (a_other: PERSON)
      -- Marry `a_other`.
      do
        ... 
      end
  end
end

class MARRIAGE
  feature
    make
      local
        alice: PERSON
        bob: PERSON
      do
        create alice
        create bob
        bob.marry (alice)
      end
  end
end
```
Write the contracts

class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            ??
        ensure
            ??

    invariant
        ??
end
A possible solution

class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other /= Current
            a_other.spouse = Void
            spouse = Void
        ensure
            spouse = a_other
            a_other.spouse = Current
    end

invariant
    spouse /= Void implies spouse.spouse = Current
    spouse /= Current
end
Implementing *marry*

class PERSON
feature
    spouse: PERSON

marry (a_other: PERSON)
    require
        a_other /= Void
        a_other /= Current
        a_other.spouse = Void
        spouse = Void
    do
        ??
    ensure
        spouse = a_other
        a_other.spouse = Current
    end

invariant
    spouse /= Void implies spouse.spouse = Current
    spouse /= Current
Implementing `marry` I

```plaintext
class PERSON
    feature
        spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other /= Current
            a_other.spouse = Void
            spouse = Void
        do
            a_other.spouse := Current
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

    invariant
        spouse /= Void implies spouse.spouse = Current
        spouse /= Current
end
```

Compiler Error: No assigner command for a_other
class PERSON
feature
  spouse: PERSON

marry (a_other: PERSON)
  require
    a_other /= Void and a_other /= Current
    a_other.spouse = Void
    spouse = Void
  do
    a_other.set_spouse (Current)
    spouse := a_other
  ensure
    spouse = a_other
    a_other.spouse = Current
end

set_spouse (a_person: PERSON)
  do
    spouse := a_person
  end

invariant
  spouse /= Void implies spouse.spouse = Current
  spouse /= Current
end

local
  bob, alice: PERSON
  do
    create bob; create alice
    bob.marry (alice)
    bob.set_spouse (Void)
  -- What about the invariants
  -- of bob and alice?
end
Implementing `marry III`

```plaintext
class PERSON

feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void and a_other /= Current
            a_other.spouse = Void
        spouse = Void
        do
            a_other.set_spouse (Current)
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

feature {PERSON}
    set_spouse (a_person: PERSON)
        do
            spouse := a_person
        end

invariant
    spouse /= Void implies spouse.spouse = Current
    spouse /= Current
end
```

What about the invariant of `a_other` in feature `marry`?
Implementing `marry` : final version

class PERSON

feature

spouse: PERSON

marry (a_other: PERSON)

require

  a_other /= Void
  a_other.spouse = Void
  spouse = Void

do

  spouse := a_other
  a_other.set_spouse (Current)

ensure

  spouse = a_other
  a_other.spouse = Current

end

feature {PERSON}

set_spouse (a_person: PERSON)

do

  spouse := a_person

end

invariant

  spouse /= Void implies spouse.spouse = Current
  spouse /= Current

end
Ending the marriage

```plaintext
class PERSON
feature
  spouse: PERSON

divorce
  require
    spouse /= Void
  do
    spouse.set_spouse (Void)
    spouse := Void
  ensure
    spouse = Void
    (old spouse).spouse = Void
end

invariant
  spouse /= Void implies spouse.spouse = Current
  spouse /= Current
end
```

Hands-On

Is the order of instructions in `divorce` important for the invariant?
What we have seen

- Invariant should only depend on Current object

- If invariant depends on other objects
  - Take care who can change state
  - Take care in which order you change state

- Invariant can be temporarily violated
  - You can still call features on Current object
  - Take care in calling other objects, they might call back

Although writing invariants is not that easy, they are necessary to do formal proofs. This is also the case for loop invariants (which will come later).