The following slides contain advanced material and are optional.
Outline

- Void-safety
  - Problem of void-calls
  - A solution to void-calls
  - Attached types
  - Certified attachment patterns
  - Object test
  - Void-safety in other languages

For detailed information, see
“Avoid a Void: The eradication of null dereferencing”
http://s.eiffel.com/void_safety_paper
I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

By Tony Hoare, 2009
Problem of void-calls

- Entities are either
  - Attached: referencing an object
  - Detached: void

- Calls on detached entities produce a runtime error

- Runtime errors are bad...

How can we prevent this problem?
Solution to void-calls

A call $f.x(...)\) is only allowed, if $f$ is statically attached.

- **Statically attached**: checked at compile time
- **Dynamically attached**: attached at runtime

**Consistency:**

If $f$ is statically attached, its possible runtime values are dynamically attached.
Statically attached entities

- Attached types
  - Types which cannot be void
  - `x: attached STRING`

- Certified attachment patterns (CAP)
  - Code pattern where attachment is guaranteed
  - `if x /= Void then x.f end` (where x is a local)

- Object test
  - Assign result of arbitrary expression to a local
  - Boolean value indicating if result is attached
  - `if attached x as l then l.f end`
Attached types

- Can declare type of entities as **attached** or **detachable**
  - _att_: **attached** STRING
  - _det_: **detachable** STRING

Attached types
- Can call features: _att_.to_upper
- Can be assign to detachable: _det_ := _att_
- Cannot be set to void: _att_ := Void

Detachable types
- No feature calls: _det_.to_upper
- Cannot be assign to attached: _att_ := _det_
- Can be set to void: _det_ := Void
Attached types (cont.)

- Entities need to be initialized
  - Detachable: void
  - Attached: assignment or creation

- Initialization rules for attached types
  - **Locals**: before first use
  - **Attributes**: at end of each creation routine
  - Compiler uses simple control-flow analysis

- Types without attachment mark
  - Currently defaults to detachable
  - In future will be switched to attached
Attached types demo

- EiffelStudio settings
- Declaration
- Error messages
Certified attachment pattern (CAP)

- Code patterns where attachment is guaranteed
- Basic CAP for locals and arguments
  - Void check in conditional or semi-strict operator
  - Setter or creation

```plaintext
capitalize (a_string: detachable STRING)
  do
    if a_string /= Void then
      a_string.to_upper
    end
  ensure
    a_string /= Void implies a_string.is_upper
  end
```
Different CAPs for locals and arguments
- Void check in contract
- Void check in conditional
- Setter
- Creator
Checking attachment of an expression (and its type)
Assignment to a local
  Local is not declared and only available in one branch

name: detachable STRING

capitalize_name
do
  if attached name as l_name then
    l_name.to_upper
  end
ensure
  attached name as n and then n.is_upper
end
Object test demo

- Object test in body
- Object test in assertion
- Object test to test for type
Stable attributes

- Detachached attributes which are never set to void
- They are initially void, but once attached will stay so
- The basic CAPs work for them as well

```
stable name: detachable STRING

capitalize_name
do
  if name /= Void then
    name.to_upper
  end
end
```
Generics

Will be discussed when generics are introduced
Void-safety in other languages: Spec#

- Research variant of C#
- Adds contracts and non-null types (and more)
- Non-null types are marked with !

```csharp
String s = null;

String! s = "abc";

String! s = null;
```
Void-safety in other languages: JML

- Research variant of Java
- Adds contracts and non-null types (and more)
- Types (except locals) are non-null per default

```java
String s = null;

String s = "abc";

/**< @ nullable @*/ String s = null;
```