null & NullReferenceException

`null`

- represents... nothing. Like, we can point at something, or at nothing.

`null`

- represents... nothing. Like, we can point at something, or at nothing.
- invented by Tony Hoare, an outstanding computer scientist (among many others, HE invented for example, *quick sort*)

`null`

- represents... nothing. Like, we can point at something, or at nothing.
- invented by Tony Hoare, an outstanding computer scientist (among many others, HE invented for example, *quick sort*)
- years later Tony Hoare apologized for inventing the null reference (😅):
  - *"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."*

`null`

- represents... nothing. Like, we can point at something, or at nothing.
- invented by Tony Hoare, an outstanding computer scientist (among many others, HE invented for example, *quick sort*)
- years later Tony Hoare apologized for inventing the null reference (😅):
  - *"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."*

Part of "memory safety":

`null`

- represents... nothing. Like, we can point at something, or at nothing.
- invented by Tony Hoare, an outstanding computer scientist (among many others, HE invented for example, *quick sort*)
- years later Tony Hoare apologized for inventing the null reference (😅):
  - *"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."*

Part of "memory safety":

- it's better to be hit by `NullReferenceException` than by random error few hours later

`null`

- represents... nothing. Like, we can point at something, or at nothing.
- invented by Tony Hoare, an outstanding computer scientist (among many others, HE invented for example, *quick sort*)
- years later Tony Hoare apologized for inventing the null reference (😅):
  - *"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."*

Part of "memory safety":

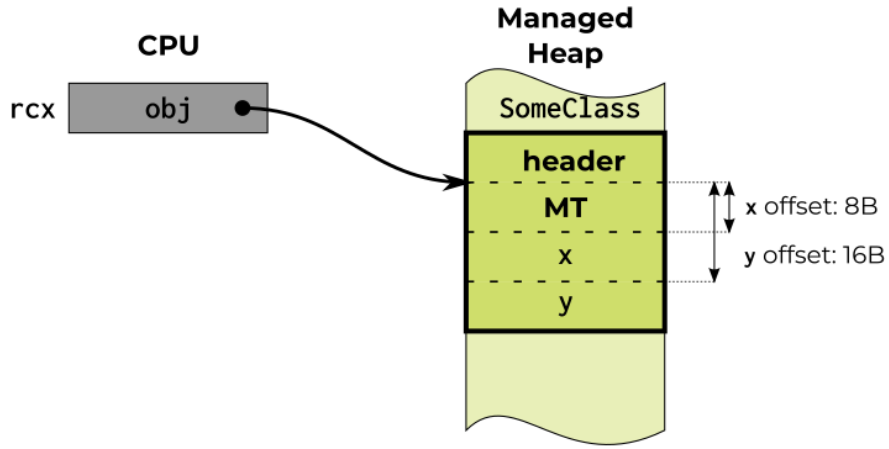
- it's better to be hit by `NullReferenceException` than by random error few hours later
- but... it is even better to have language that does not allow to have non-initialized/zeroed references
  - like F#
  - like C# trying to be with *nullable reference types*

null

```
public class C {
  static public void M(SomeClass obj) {
    Console.WriteLine(obj.Y);
    ...
  }
}

public class SomeClass {
  public long X;
  public long Y;
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```



- **rcx** is the first argument - reference (address) to the **obj**
- **ldfld** is access to field **Y** - we need to add **Y** field offset (**0x10** is **16**)

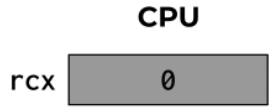
```
C.M(SomeClass)
    mov rcx, [rcx+0x10]
    jmp System.Console.WriteLine(Int64)
```

null

```
public class C {
  static public void M(SomeClass obj) {
    Console.WriteLine(obj.Y);
    ...
  }
}

public class SomeClass {
  public long X;
  public long Y;
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```



- if **obj** is **null** - reference (address) to the **obj** is just **0**
- **ldfld** is access to field **Y** - we need to add **Y** field offset (**0x10** is **16**) to... **0**
- we will access address **0x0000000000000010**

```
C.M(SomeClass)
  mov rcx, [rcx+0x10]
  jmp System.Console.WriteLine(Int64)
```

- this is handled by operating system as **invalid access to the protected page** - first 4kB of every process
- .NET runtime wraps it to re-throw as **NullReferenceException**

null

What exception it will generate?

```
Unsafe.Read<byte>((void*)0);
```

null

What exception it will generate?

```
Unsafe.Read<byte>((void*)0);
```

NullReferenceException - as we've just learnt, we are trying to access **the first, protected** page.

null

What exception it will generate?

```
Unsafe.Read<byte>((void*)0);
```

NullReferenceException - as we've just learnt, we are trying to access **the first, protected** page.

What exception it will generate?

```
Unsafe.Read<byte>((void*)0x1_001);
```

null

What exception it will generate?

```
Unsafe.Read<byte>((void*)0);
```

`NullReferenceException` - as we've just learnt, we are trying to access **the first, protected** page.

What exception it will generate?

```
Unsafe.Read<byte>((void*)0x1_001);
```

Well, this trying to access **the second page** (**`0x1_000`** is 4kB) which is not special. But it is probably not allocated to our process.

null

What exception it will generate?

```
Unsafe.Read<byte>((void*)0);
```

NullReferenceException - as we've just learnt, we are trying to access **the first, protected** page.

What exception it will generate?

```
Unsafe.Read<byte>((void*)0x1_001);
```

Well, this trying to access **the second page** (**0x1_000** is 4kB) which is not special. But it is probably not allocated to our process. So, this is also invalid access to the page, but for every other than the first page, it is wrapped by .NET runtime into **AccessViolationException**.

null

Careful watcher may ask - but **what if object is bigger than 4kB**?! Trying to access some distant fields (or array elements) will try to access beyond the first, specially protected page:

```
public class SomeClass
{
  public long Field0;
  public long Field1;
  ...
  public long Field8230;
}

public static void Test(SomeClass obj)
{
  Console.WriteLine(obj.Field8000); // if obj is null, throws AccessViolationException...?!
}
```

null

Careful watcher may ask - but **what if object is bigger than 4kB**?! Trying to access some distant fields (or array elements) will try to access beyond the first, specially protected page:

```
public class SomeClass
{
  public long Field0;
  public long Field1;
  ...
  public long Field8230;
}

public static void Test(SomeClass obj)
{
  Console.WriteLine(obj.Field8000); // if obj is null, throws AccessViolationException...?!
}
```

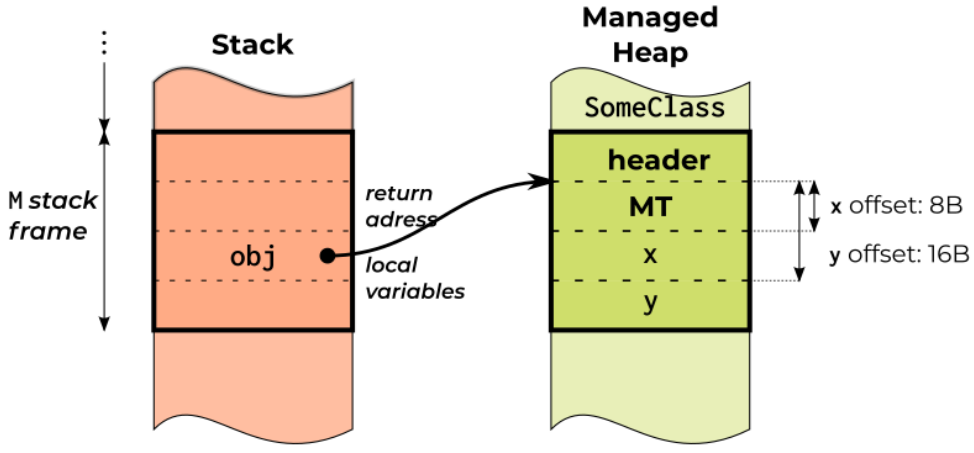
But, JIT is clever! In such case, it adds null checking of the entire object **before** field access:

```
C.Test(SomeClass)
  ...
  cmp [rcx], ecx          ; Invalid address access handled as NullReferenceException
  mov rcx, [rcx+0xfa08]
  call System.Console.WriteLine(Int64)
  ...
```

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    obj = DoSomething();
    Console.WriteLine(obj.Y);
    ...
  }
}
```

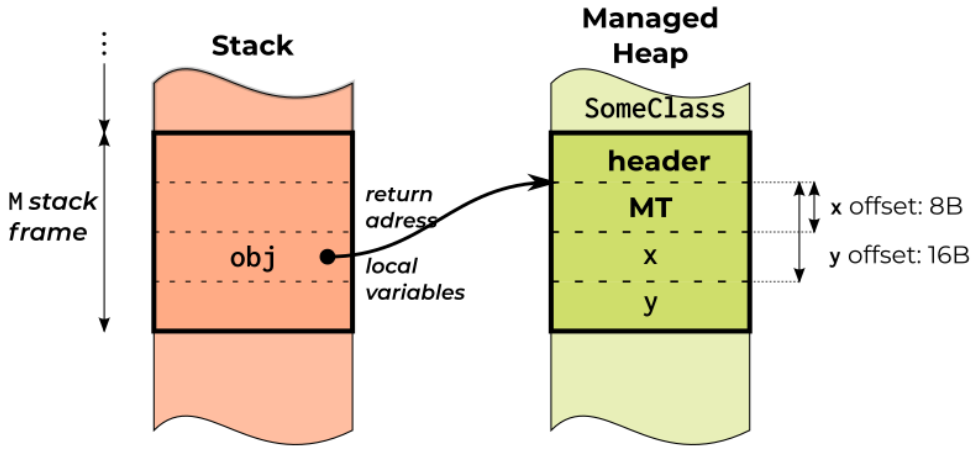
```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```



null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    obj = DoSomething();
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```

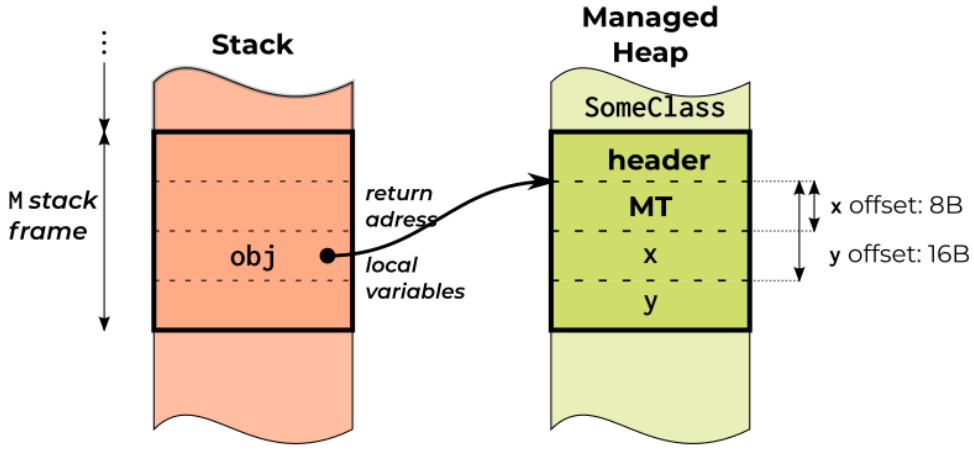



```
C.M(SomeClass)
  ...
  call C.DoSomething()
  mov rcx, [rax+0x10]
```

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    obj = DoSomething();
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```



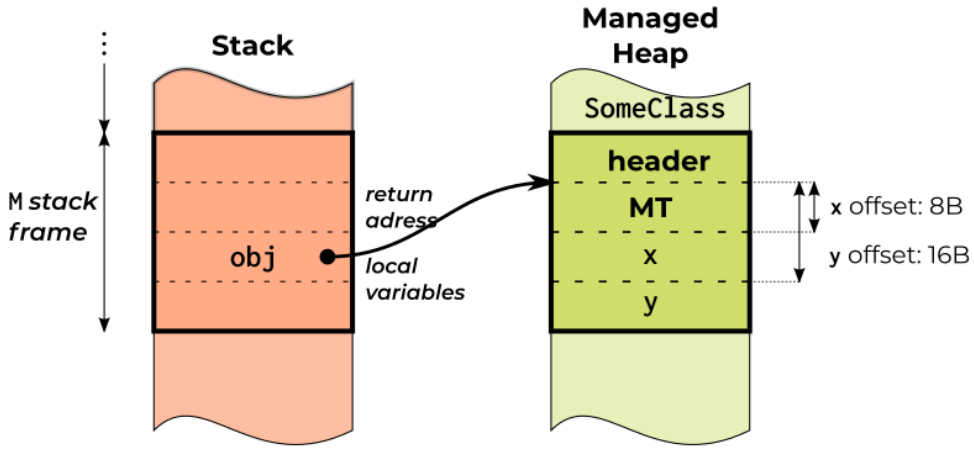

```
C.M(SomeClass)
  ...
  call C.DoSomething()
  mov rcx, [rax+0x10]
```

If **DoSomething** returns (in **rax**) a valid reference - everything is fine.

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    obj = DoSomething();
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```




```
C.M(SomeClass)
  ...
  call C.DoSomething()
  mov rcx, [rax+0x10]
```

If **DoSomething** returns (in **rax**) a valid reference - everything is fine.

If it returns **null** (0) - again we try to access address **0x10** (the first page).

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```

- stack frame **is zeroed** by default - so all references are `null` by default

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```

- stack frame **is zeroed** by default - so all references are **`null`** by default, so it would throw **NRE**:

```
C.M(SomeClass)
...
mov rcx, [rax+0x10]
call System.Console.WriteLine(Int64)
```

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```

- stack frame **is zeroed** by default - so all references are **`null`** by default, so it would throw **`NRE`**:

```
C.M(SomeClass)
...
mov rcx, [rax+0x10]
call System.Console.WriteLine(Int64)
```

- BTW, compilers try to disallow usage of uninitiated variables:

```
error CS0165: Use of unassigned local variable 'obj'
```

null

```
public class C {
  static public void M() {
    SomeClass obj;
    ...
    Console.WriteLine(obj.Y);
    ...
  }
}
```

```
.method public hidebysig static
void M (class SomeClass obj) cil managed
{
  ldarg.0
  ldfld int32 SomeClass::Y
  call void System.Console::WriteLine(int32)
  ret
}
```

- stack frame **is zeroed** by default - so all references are **`null`** by default, so it would throw **`NRE`**:

```
C.M(SomeClass)
...
mov rcx, [rax+0x10]
call System.Console.WriteLine(Int64)
```

- BTW, compilers try to disallow usage of uninitiated variables:

```
error CS0165: Use of unassigned local variable 'obj'
```

- we can disable stack zeroing **and** allow to use uninitialized locals (😱) - we will come to that...

`null` or not to `null`?

There is a popular question - does it make sense to **manually** null references as soon as we don't need them:

```
var x = new X();
x.DoSomething();
x = null;
await DoSomeLondRunningCallAsync();
```

This is nice question...

`null` or not to `null`?

There is a popular question - does it make sense to **manually** null references as soon as we don't need them:

```
var x = new X();
x.DoSomething();
x = null;
await DoSomeLondRunningCallAsync();
```

This is nice question...but we need more GC knowledge to answer. We will return to it in the next modules!

`null` summary

- there are no any special *null-checks* in generated code:

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page
  - thus, *nullable reference types* will not speed up things - we are not getting rid of any null checks (because there are none)

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page
  - thus, *nullable reference types* will not speed up things - we are not getting rid of any null checks (because there are none)
  - unless they are, in special cases (like the "huge object" access)

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page
  - thus, *nullable reference types* will not speed up things - we are not getting rid of any null checks (because there are none)
  - unless they are, in special cases (like the "huge object" access)
- in typical application we should not observe **`AccessViolationException`**

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page
  - thus, *nullable reference types* will not speed up things - we are not getting rid of any null checks (because there are none)
  - unless they are, in special cases (like the "huge object" access)
- in typical application we should not observe **`AccessViolationException`**
  - it is just for super rare cases when memory safety was broken

`null` summary

- there are no any special *null-checks* in generated code:
  - it is just handled by accessing the first, protected page
  - thus, *nullable reference types* will not speed up things - we are not getting rid of any null checks (because there are none)
  - unless they are, in special cases (like the "huge object" access)
- in typical application we should not observe **`AccessViolationException`**
  - it is just for super rare cases when memory safety was broken

    (we will show some of them during the course)