SWEN90010 — High Integrity Systems Engineering

Lecture 11 - Ada Programming Language

Hira Syeda
hira.syeda@unimelb.edu.au
Level 2, Melbourne Connect

11th April, 2025

Second Half of the Subject

we began at the high level to gather requirements to make sure that the systems you're building are safe and secure

And then we talked about how can we design (specify) our systems to make sure that they're safe and secure

Second Half of the Subject

we began at the high level to gather requirements to make sure that the systems you're building are safe and secure

And then we talked about how can we design (specify) our systems to make sure that they're safe and secure

having talked about requirements and design,
What else should we talk about now?

Second Half of the Subject

we began at the high level to gather requirements to make sure that the systems you're building are safe and secure

And then we talked about how can we design (specify) our systems to make sure that they're safe and secure

How do we write code (programs) to ensure our system behaves as intended by its specifications and requirements?

Programs of/for High Integrity Systems

we want our code to be unambiguous as well, so that we can reason about its functionality

It turns out that for most ordinary programming languages, doing that is not all that straightforward

for most programming languages, its difficult to even take a simple piece of code and be able to predict exactly how that code is going to behave in all situations

Introducing **Ada**

Let’s write **Hello World!** in Ada

Augusta Ada King (née Byron)

Augusta Ada King
(née Byron)

aka **Ada**, Countess of **Lovelace**

Augusta Ada King
(née Byron)

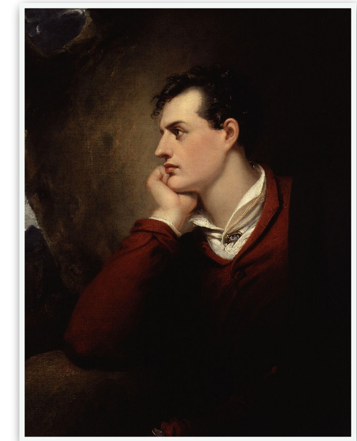
aka **Ada**, Countess of **Lovelace**

10 Dec 1815 - 27 Nov 1852

Augusta Ada King
(née Byron)

aka **Ada**, Countess of **Lovelace**

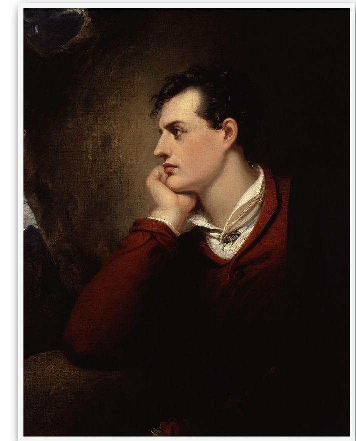
10 Dec 1815 - 27 Nov 1852



Augusta Ada King
(née Byron)

aka **Ada**, Countess of **Lovelace**

10 Dec 1815 - 27 Nov 1852

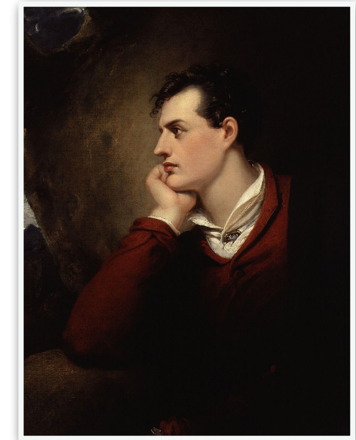


Augusta Ada King
(née Byron)

aka **Ada**, Countess of **Lovelace**

10 Dec 1815 - 27 Nov 1852

16 Jan 1816: Lord Byron departs



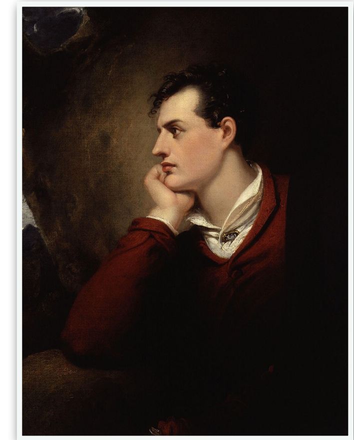
Augusta Ada King (née Byron)

aka **Ada**, Countess of **Lovelace**

10 Dec 1815 - 27 Nov 1852

16 Jan 1816: Lord Byron departs

Studied math at early age, tutored by De Morgan



Augusta Ada King
(née Byron)

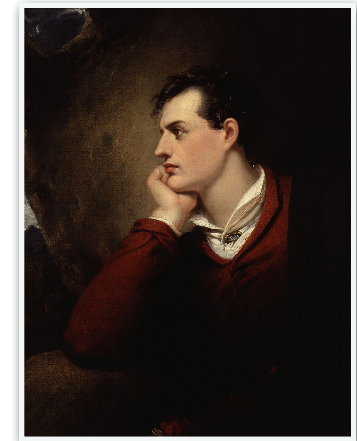
aka **Ada**, Countess of **Lovelace**

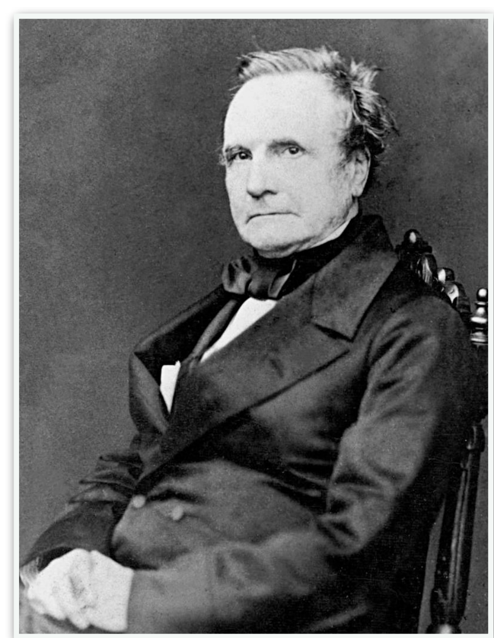
10 Dec 1815 - 27 Nov 1852

16 Jan 1816: Lord Byron departs

Studied math at early age,
tutored by De Morgan

June 1833 (age 17)





Augusta Ada King (née Byron)

aka **Ada**, Countess of **Lovelace**

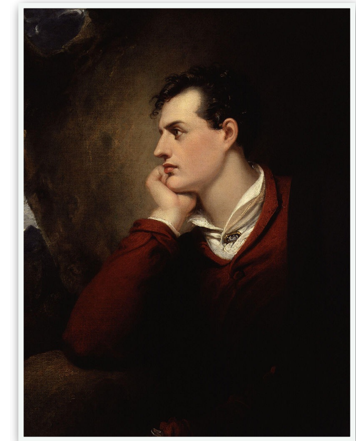
10 Dec 1815 - 27 Nov 1852

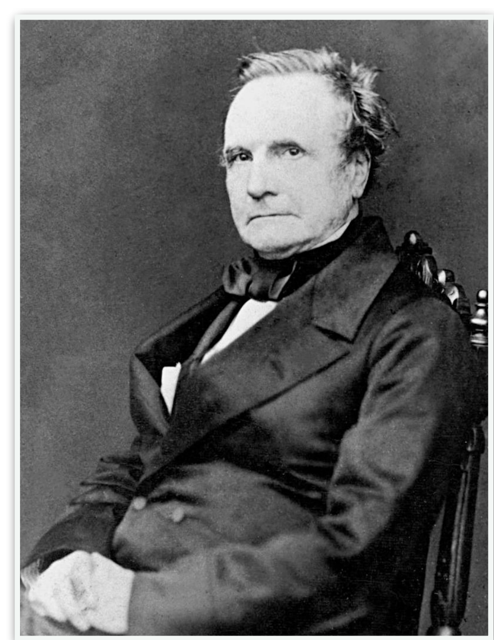
16 Jan 1816: Lord Byron departs

Studied math at early age, tutored by De Morgan

June 1833 (age 17)

1842-43: Analytical Engine translation plus notes





Augusta Ada King
(née Byron)

aka **Ada**, Countess of **Lovelace**

10 Dec 1815 - 27 Nov 1852

16 Jan 1816: Lord Byron departs

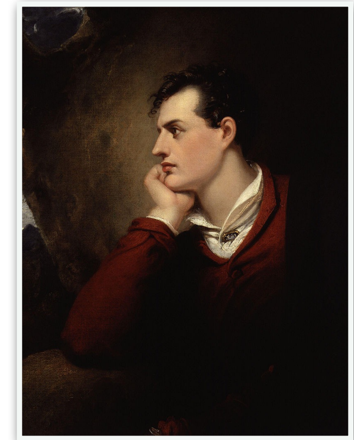
Studied math at early age,
tutored by De Morgan

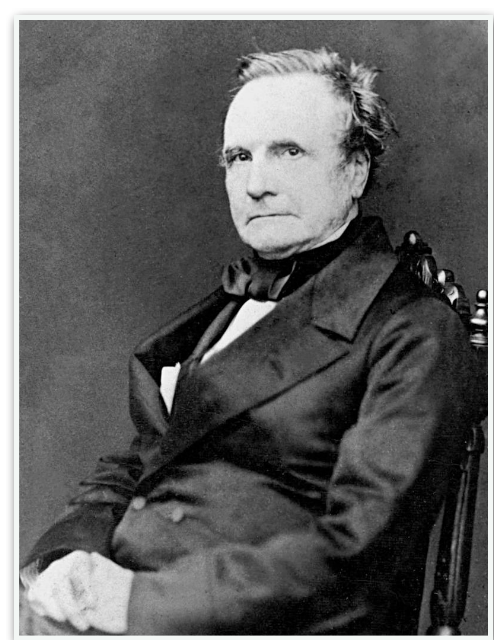
June 1833 (age 17)

1842-43: Analytical Engine
translation plus notes

$$B_m^+ = 1 - \sum_{k=0}^{m-1} \binom{m}{k} \frac{B_k^+}{m-k+1}$$

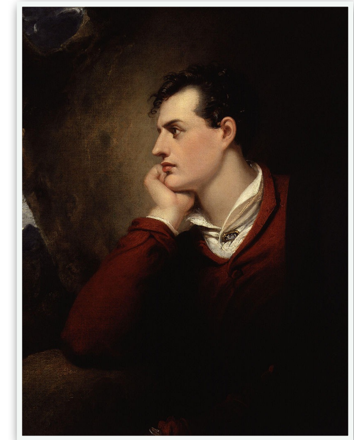
Bernoulli Numbers





Augusta Ada King (née Byron)

THE UNIVERSITY OF MELBOURNE




aka **Ada**, Countess of **Lovelace**

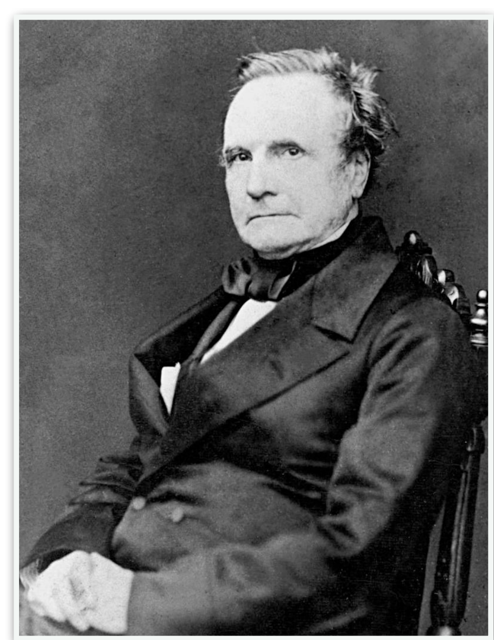
10 Dec 1815 - 27 Nov 1852

16 Jan 1816: Lord Byron departs

Studied math at early age, tutored by De Morgan

June 1833 (age 17)

1842-43: Analytical Engine translation plus notes



$$B_m^+ = 1 - \sum_{k=0}^{m-1} \binom{m}{k} \frac{B_k^+}{m-k+1}$$

Bernoulli Numbers

Analytical Engine as Logical Symbol Manipulation Engine

Ada

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

10 Dec 1980: MIL-STD-**1815** approved

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

10 Dec 1980: MIL-STD-**1815** approved

1983: ANSI/MIL-STD-1815A ("Ada 83")

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

10 Dec 1980: MIL-STD-**1815** approved

1983: ANSI/MIL-STD-1815A ("Ada 83")

1995: ISO-8652:1995 ("Ada 95")

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

10 Dec 1980: MIL-STD-**1815** approved

1983: ANSI/MIL-STD-1815A ("Ada 83")

1995: ISO-8652:1995 ("Ada 95")

USAF funds GNAT Compiler, now in GCC

Ada

mid 1970s: US DoD has too many languages for myriad embedded devices;
want: safe, modular, cross platform, high level

1977: solicit proposals for new language

1979: Honeywell's proposal chosen; christened **Ada**

10 Dec 1980: MIL-STD-**1815** approved

1983: ANSI/MIL-STD-1815A ("Ada 83")

1995: ISO-8652:1995 ("Ada 95")

USAF funds GNAT Compiler, now in GCC

2012: ISO/IEC 8652:2012 ("Ada 2012")

Strong, Static Typing

Strong, Static Typing

```
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

Strong, Static Typing

```ada
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

```
$ gnatmake StrongTyping.adb
gcc -c strongtyping.adb
strongtyping.adb:7:09: expected type "Standard.Integer"
strongtyping.adb:7:09: found type "Standard.Float"
gnatmake: "strongtyping.adb" compilation error
$
```

Strong, Static Typing

```
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

```
$ gnatmake StrongTyping.adb
gcc -c strongtyping.adb
strongtyping.adb:7:09: expected type "Standard.Integer"
strongtyping.adb:7:09: found type "Standard.Float"
gnatmake: "strongtyping.adb" compilation error
$
```

Prevents e.g.
loss of precision

Strong, Static Typing

THE UNIVERSITY OF MELBOURNE

```
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

```
int main(void){
  int i = 0;
  float f = 0.0;
  i = f;
  return 0;
}
```

```
$ gnatmake StrongTyping.adb
gcc -c strongtyping.adb
strongtyping.adb:7:09: expected type "Standard.Integer"
strongtyping.adb:7:09: found type "Standard.Float"
gnatmake: "strongtyping.adb" compilation error
$
```

Prevents e.g.
loss of precision

Strong, Static Typing

THE UNIVERSITY OF MELBOURNE

```
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

```
int main(void){
  int i = 0;
  float f = 0.0;
  i = f;
  return 0;
}
```

```
$ gnatmake StrongTyping.adb
gcc -c strongtyping.adb
strongtyping.adb:7:09: expected type "Standard.Integer"
strongtyping.adb:7:09: found type "Standard.Float"
gnatmake: "strongtyping.adb" compilation error
$
```

Prevents e.g.
loss of precision

```
$ gcc strong_typing.c -o strong_typing
$
```

Strong, Static Typing

```
procedure StrongTyping is
   I : Integer := 0;
   J : Float := 0.0;
begin
   I := J;
end StrongTyping;
```

```
int main(void){
  int i = 0;
  float f = 0.0;
  i = f;
  return 0;
}
```

```
$ gnatmake StrongTyping.adb
gcc -c strongtyping.adb
strongtyping.adb:7:09: expected type "Standard.Integer"
strongtyping.adb:7:09: found type "Standard.Float"
gnatmake: "strongtyping.adb" compilation error
$
```

Prevents e.g.
loss of precision

```
$ gcc strong_typing.c -o strong_typing
$
```

C: weak, static typing

Module System

Module System

```
package ModuleTwo is
   procedure DoNothing;
end ModuleTwo;
```

```
package body ModuleTwo is
   procedure DoNothing is
   begin
      null;
   end DoNothing;
end ModuleTwo;
```

Module System

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

```
with ModuleTwo;

package body ModuleOne is
   procedure DoNothing is
   begin
      ModuleTwo.DoNothing;
   end DoNothing;
end ModuleOne;
```

```
package ModuleTwo is
   procedure DoNothing;
end ModuleTwo;
```

```
package body ModuleTwo is
   procedure DoNothing is
   begin
      null;
   end DoNothing;
end ModuleTwo;
```

Module System

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

```
with ModuleTwo;

package body ModuleOne is
   procedure DoNothing is
   begin
      ModuleTwo.DoNothing;
   end DoNothing;
end ModuleOne;
```

```
$ gnatmake ModuleOne.adb
gcc -c moduleone.adb
gcc -c moduletwo.adb
$
```

```
package ModuleTwo is
   procedure DoNothing;
end ModuleTwo;
```

```
package body ModuleTwo is
   procedure DoNothing is
   begin
      null;
   end DoNothing;
end ModuleTwo;
```

Module System

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

```
package ModuleTwo is
   procedure DoNothing;
end ModuleTwo;
```

```
with ModuleTwo;

package body ModuleOne is
   procedure DoNothing is
   begin
      ModuleTwo.DoNothing;
   end DoNothing;
end ModuleOne;
```

```
package body ModuleTwo is
   procedure DoNothing is
   begin
      null;
   end DoNothing;
end ModuleTwo;
```

```
$ gnatmake ModuleOne.adb
gcc -c moduleone.adb
gcc -c moduletwo.adb
$
```

Automatic dependency resolution

Module System

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

```
package ModuleTwo is
   procedure DoNothing;
end ModuleTwo;
```

```
with ModuleTwo;

package body ModuleOne is
   procedure DoNothing is
   begin
      ModuleTwo.DoNothing;
   end DoNothing;
end ModuleOne;
```

```
package body ModuleTwo is
   procedure DoNothing is
   begin
      null;
   end DoNothing;
end ModuleTwo;
```

```
$ gnatmake ModuleOne.adb
gcc -c moduleone.adb
gcc -c moduletwo.adb
$
```

Automatic dependency resolution

Information Hiding, Encapsulation

Portable

Portable

If you try to perform an operation that produces a value outside the allowed range of a signed integer type, Ada raises a **Constraint_Error** to prevent incorrect results”

Portable

If you try to perform an operation that produces a value outside the allowed range of a signed integer type, Ada raises a **Constraint_Error** to prevent incorrect results”

```
procedure Overflow is
   I : Integer := 0;
begin
   loop
      I := I + 1;
   end loop;
end Overflow;
```

Portable

THE UNIVERSITY OF MELBOURNE

If you try to perform an operation that produces a value outside the allowed range of a signed integer type, Ada raises a **Constraint_Error** to prevent incorrect results"

```
procedure Overflow is
   I : Integer := 0;
begin
   loop
      I := I + 1;
   end loop;
end Overflow;
```

```
$ gnatmake Overflow.adb
gcc -c overflow.adb
gnatbind -x overflow.ali
gnatlink overflow.ali
$ ./overflow

raised CONSTRAINT_ERROR : overflow.adb:5 overflow check failed
$
```

Portable (c.f. C)

Portable (c.f. C)

```c
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

Portable (c.f. C)

```c
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Portable (c.f. C)

```
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Is this any safer?

Portable (c.f. C)

(Cost of Runtime Checking)

```c
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Is this any safer?

Portable (c.f. C)

```
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Is this any safer?

(Cost of Runtime Checking)

```
$ time ./overflow

raised CONSTRAINT_ERROR
real  0m4.775s
user  0m4.770s
sys   0m0.003s
```

Portable (c.f. C)

```
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Is this any safer?

(Cost of Runtime Checking)

```
$ time ./overflow

raised CONSTRAINT_ERROR
real  0m4.775s
user  0m4.770s
sys   0m0.003s
```

```
$ time ./overflow_c

real  0m4.072s
user  0m4.066s
sys   0m0.004s
```

Portable (c.f. C)

THE UNIVERSITY OF MELBOURNE

```
int main(void){
  int i = 0;
  while(i >= 0){
    i++;
  }
  return 0;
}
```

```
$ gcc overflow.c -o overflow
$ ./overflow
$
```

Is this any safer?

(Cost of Runtime Checking)

```
$ time ./overflow

raised CONSTRAINT_ERROR
real  0m4.775s
user  0m4.770s
sys   0m0.003s
```

```
$ time ./overflow_c

real  0m4.072s
user  0m4.066s
sys   0m0.004s
```

But stronger static checking can **improve** performance too. (see later)

Readability

Readability

Purposeful Verbosity

```
procedure Overflow is
   I : Integer := 0;
begin
   loop
      I := I + 1;
   end loop;
end Overflow;
```

```
int main(void){
  int i = 0;
  while(i >= 0)
    i++;
  return 0;
}
```

Readability

Purposeful Verbosity

```
procedure Overflow is
   I : Integer := 0;
begin
   loop
      I := I + 1;
   end loop;
end Overflow;
```

```
int main(void){
   int i = 0;
   while(i >= 0)
      i++;
   return 0;
}
```

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

Intended Redundancy

Readability

Purposeful Verbosity

```
procedure Overflow is
   I : Integer := 0;
begin
   loop
      I := I + 1;
   end loop;
end Overflow;
```

```
int main(void){
  int i = 0;
  while(i >= 0)
    i++;
  return 0;
}
```

```
package ModuleOne is
   procedure DoNothing;
end ModuleOne;
```

Intended Redundancy

```
$ time ./overflow

raised CONSTRAINT_ERROR
real  0m4.775s
user  0m4.770s
sys   0m0.003s
```

Unambiguous Semantics

Ada Prefers Readability

THE UNIVERSITY OF MELBOURNE

Make **reading** programs **easier** than **writing** them.

Ada Prefers Readability

Make **reading** programs **easier** than **writing** them.

Why?

Ada Prefers Readability

Make **reading** programs **easier** than **writing** them.

Why?

Code may be read > 10 times,
but written only once.

Ada Prefers Readability

THE UNIVERSITY OF MELBOURNE

Make **reading** programs **easier** than **writing** them.

Why?

Code may be read > 10 times,
but written only once.

“First, optimise the task that contributes most to overall cost.”

Ada Prefers Readability

Make **reading** programs **easier** than **writing** them.

Why?

Code may be read > 10 times,
but written only once.

"First, optimise the task that
contributes most to overall cost."

c.f. Ruby, Javascript, Perl which do the opposite

Ada Prefers Readability

Make **reading** programs **easier** than **writing** them.

Why?

Code may be read > 10 times, but written only once.

"First, optimise the task that contributes most to overall cost."

JavaScript:

```
>>> false == 'false';
false
>>> false == '0';
true
>>> '' == '0';
false
>>> 0 == '';
true
>>> 0 == '0';
true
```

c.f. Ruby, Javascript, Perl which do the opposite

Ada Prefers Readability

Make **reading** programs **easier** than **writing** them.

Why?

Code may be read > 10 times, but written only once.

"First, optimise the task that contributes most to overall cost."

This includes **programs** that read your code

JavaScript:

```
>>> false == 'false';
false
>>> false == '0';
true
>>> '' == '0';
false
>>> 0 == '';
true
>>> 0 == '0';
true
```

c.f. Ruby, Javascript, Perl which do the opposite

Semantics and Safety

Semantics and Safety

Runtime Checks

Signed overflow

Array index out-of-bounds

Accessing unallocated memory

Range errors (see later) etc.

Semantics and Safety

Runtime Checks

Signed overflow

Array index out-of-bounds

Accessing unallocated memory

Range errors (see later) etc.

Turns **heisenbugs** into **crashes**

Semantics and Safety

Runtime Checks

Signed overflow

Array index out-of-bounds

Accessing unallocated memory

Range errors (see later) etc.

Turns **heisenbugs** into **crashes**

Implications for safety of real-time embedded, and critical systems?

Application Areas

THE UNIVERSITY OF MELBOURNE

Mostly real-time embedded

Avionics

Railway

Defence

Space

Robotics

Crypto(graphy)

Muen verified Separation Kenrel

Application Areas

THE UNIVERSITY OF MELBOURNE

Mostly real-time embedded

Avionics

Railway

Defence

Space

Robotics

Crypto(graphy)

Muen verified Separation Kenrel

Lots of these
with formal verification
via SPARK