C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 10: Typing, binding, scope

Ryerson University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Type Systems

Type System

- A set of rules that assigns a property called ***type*** to constructs of a program.
- These constructs include variables, functions, expressions, etc.

The whole point is to reduce bugs.

- For example, if a pattern of 32 bits has been encoded using 2s complement, we don't want to read it using IEEE 754
- And we *can* do this in many languages!

```c
#include <stdio.h>
#include <windows.h>

int main(void)
{
    unsigned long long a = 4607182418800017408;

    printf("as integer:  %llu\n", a);
    printf("as double:   %lf\n", a);

    system("pause");
}
```

Declare large 64-bit integer

Print as int, print as double

D:\GoogleDrive\Teaching - Humber\ATMN 253\Visual S

```
as integer:   4607182418800017408
as double:    1.000000
Press any key to continue . . .
```

- The 2s comp bit pattern was read as an IEEE 754 double.
- (The integer constant was deliberately picked to produce a bit pattern that would yield 1.000000 as double)

Type Checking

Clearly, type checking isn't performed in the context of a **`printf`** statement in C++

- Think of type checking as trying to fit puzzle pieces together.
- Does the output type of a function match the variable we're trying to store it in?
- Do the input arguments to a function match the types indicated in the parameter list?
- If no, will we allow implicit conversion?

Static VS Dynamic

When are types checked?

Statically typed languages perform type checking at *compile time*

- Checked while converting source code to machine (or byte) code

Dynamically typed languages perform type checking at *run-time*

- Checked on the fly while instructions are being executed.

Statically Typed languages: C/C++, Java, Haskell, Rust

Dynamically Typed languages: Python, Smalltalk, Elixir

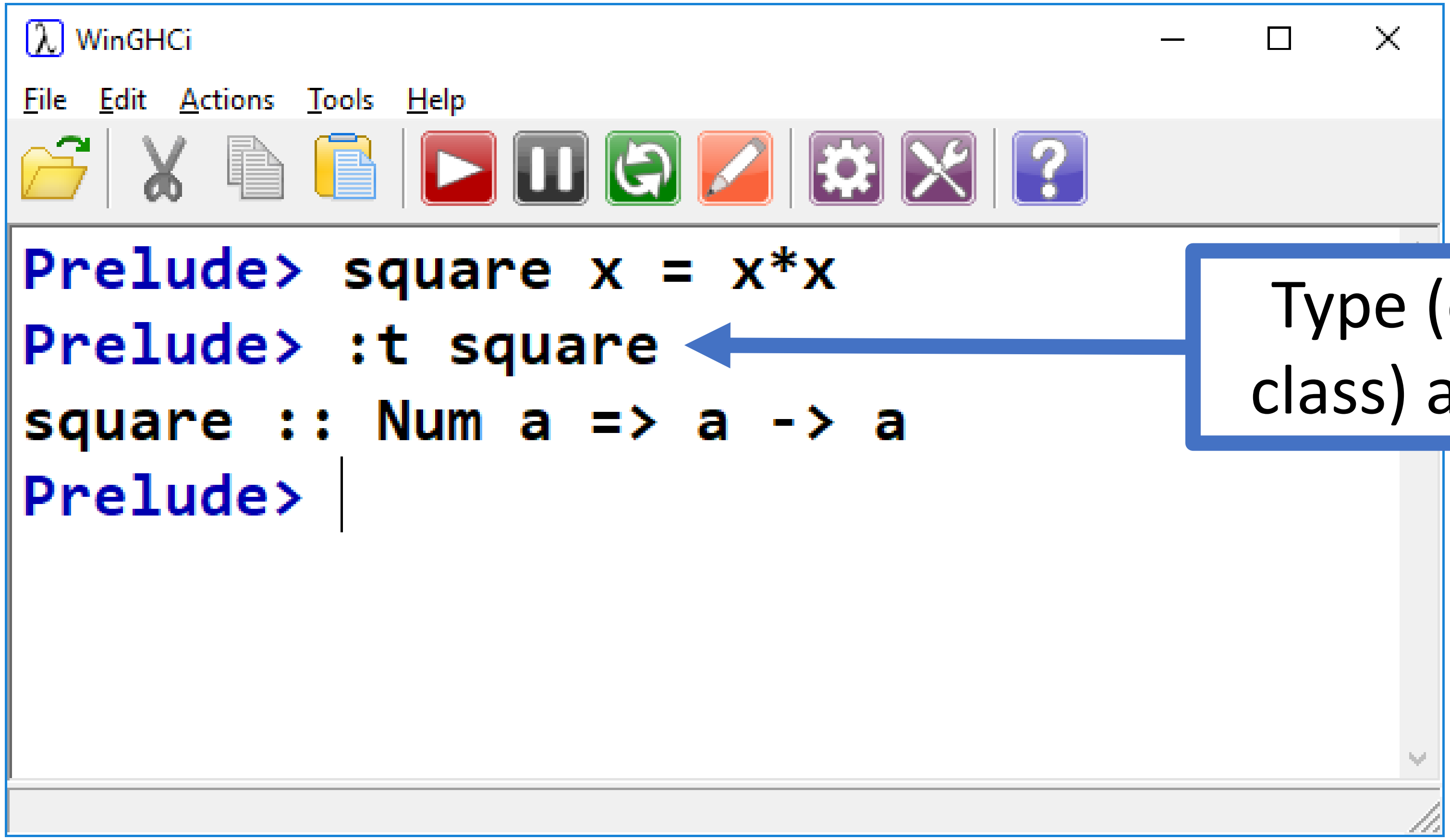
Static Type Checking

```
public class MethodTester
{
    public static void main(String[] args)
    {
        String s = "Hello";

        System.out.println(Math.sqrt(s));
    }
}
```

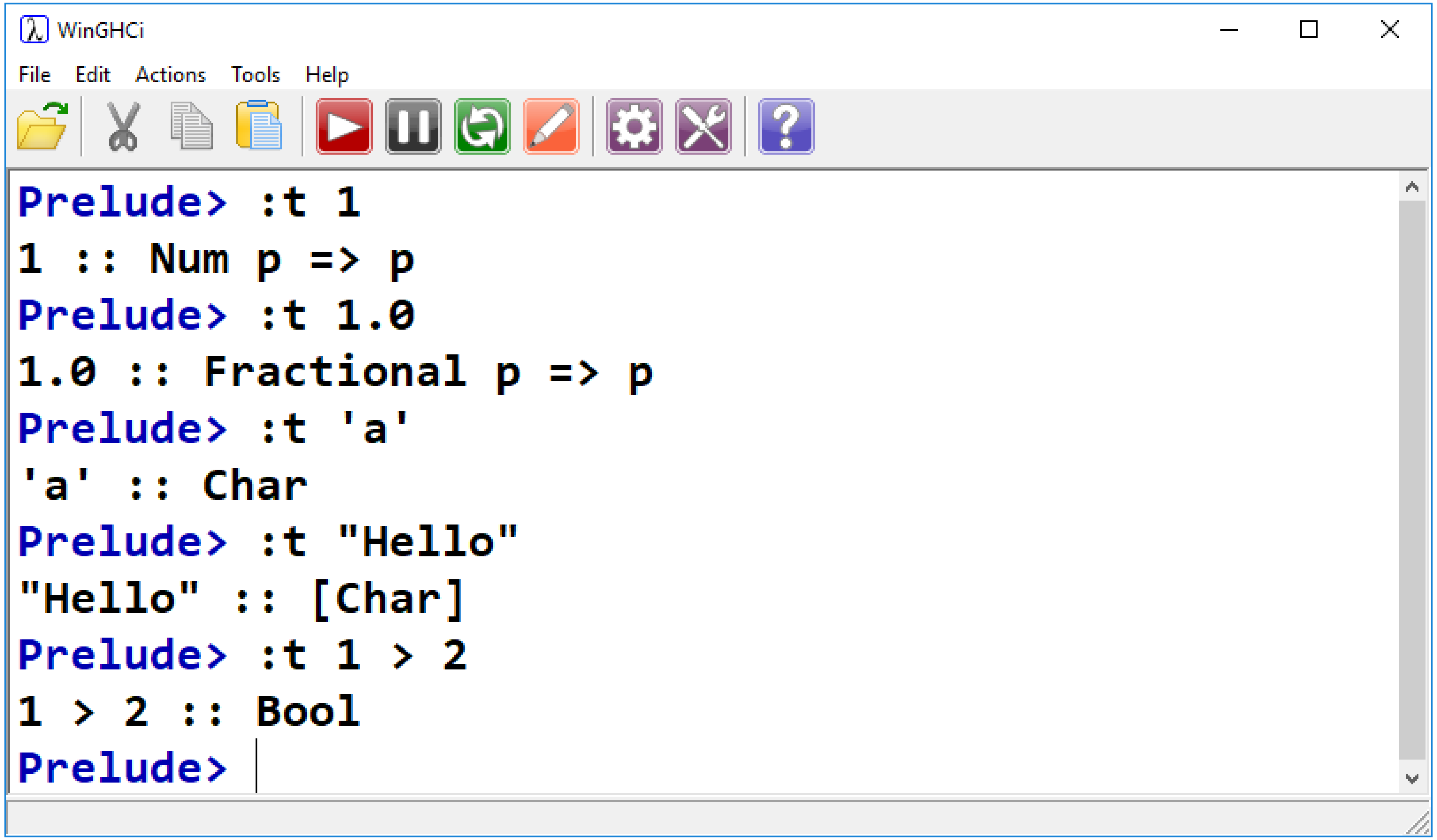
```
incompatible types: java.lang.String cannot be
converted to double
```

Static Type Checking



Type (or in this case type class) assigned to function

Static Type Checking



Static Type Checking

Rust is statically typed, but supports type inference like Haskell:

```
fn main() {
    let x = 7;
    println!("value: {}", x);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
value: 7

C:\_RustCode>_
```

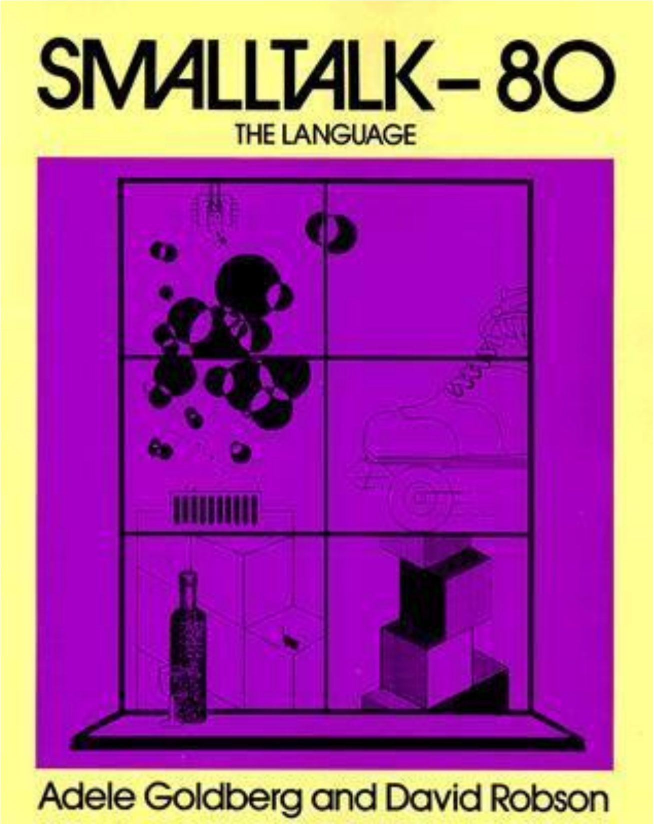
main.rs ×

```
fn main() {
    let x: u8 = 3;
    let y: i64 = 5;
    let z: isize = 999;
    println!("x: {}", x);
    println!("y: {}", y);
    println!("z: {}", z);
}
```

Dynamic Type Checking

- In dynamically typed languages, every operation knows the types for which it is valid.
- Providing invalid arguments or operands will yield a run-time error which may or may not be recoverable
- Such things can be anticipated and mitigated in various ways, such as verifying type explicitly

Dynamic Type Checking




```
factorial: n
  | fac |
  fac := 1.

  n isInteger
    ifTrue: [1 to: n do:
      [:a | fac := fac*a.].
       ^fac
    ]
    ifFalse: [
      ^'Bad input’
    ].
```

- In Java, the parameter would be defined as **int**
- Compile error if arg isn't **int**, or can't be implicitly cast as an **int**.

- Of course, polymorphism in Java complicates this.
- Still statically typed.

Dynamic Type Checking…?

```
#(1 2 3 4) + 18.2
```

- Does Smalltalk have type errors in the strict sense?
- Different objects understand different messages.
- A “type error” occurs when an object doesn’t have a method to handle a particular message.
- “Type” errors in Smalltalk are as a result of not finding a method (DNU, Did Not Understand).
- Above, the error occurs because the Array class doesn’t have an instance ***method*** for the message **#+**
- Smalltalk enthusiasts debate this.

Dynamic Type Checking

```
defmodule UserMath do
  def fib(n) when not is_integer(n) or n < 0 do
    :error
  end
  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n), do: fib(n-2) + fib(n-1)

  def fac(n) when not is_integer(n) or n < 0 do
    :error
  end
  def fac(0), do: 1
  def fac(n), do: n*fac(n-1)

end
```

Static VS Dynamic

Advantages? Disadvantages?

Static:

- Reliably find errors at compile time.
- Code will execute faster if types are assumed to be correct at run time.
- Type-specific optimization can be performed at compile time.
- I.e., integer arithmetic is typically faster than floating point

Dynamic:

- Compilers run faster
- Interpreters can dynamically load new code
  - Smalltalk, MATLAB, iex
- Easier code reuse

Static VS Dynamic

Advantages? Disadvantages?

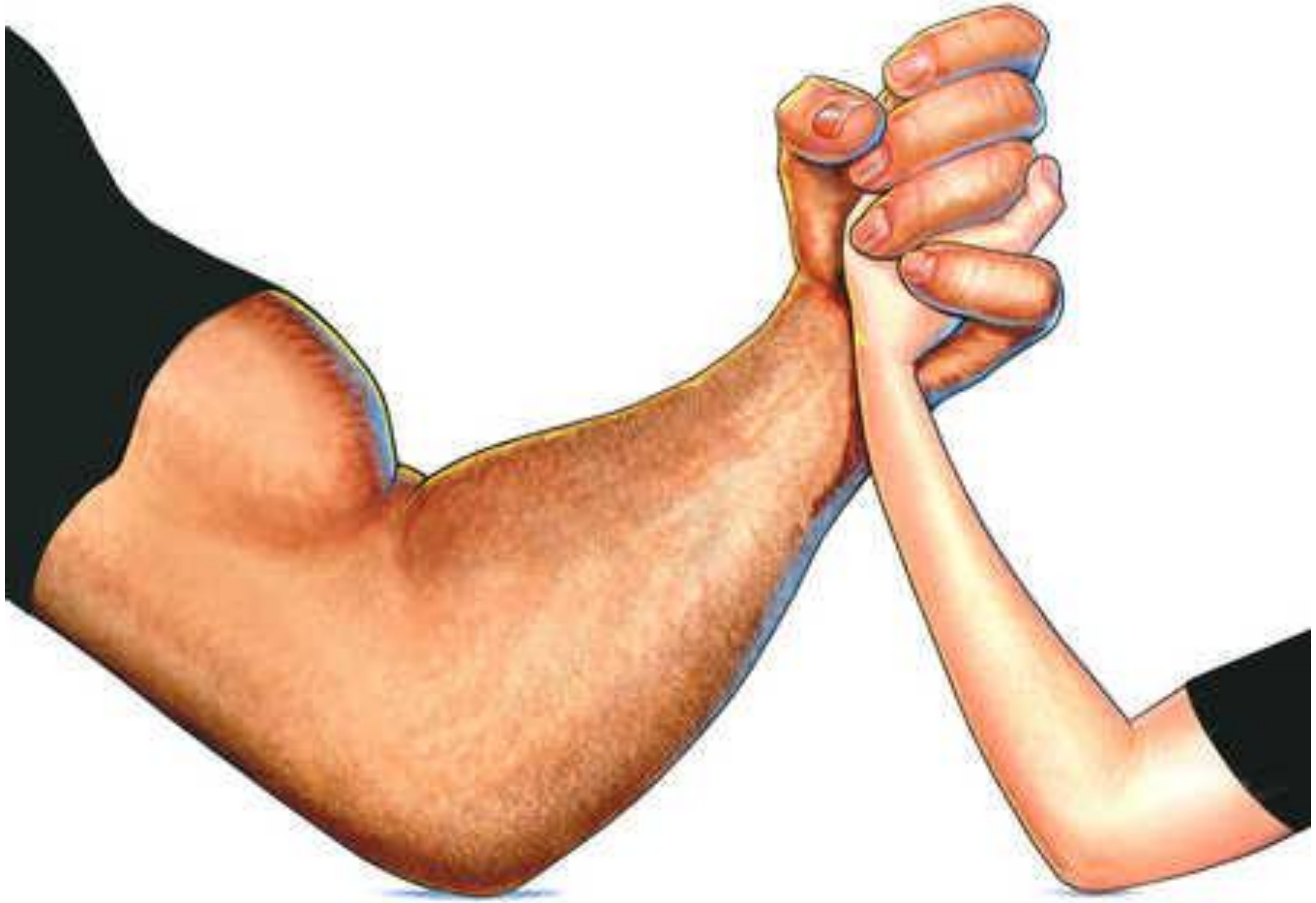
- There is much disagreement among programmers about just how much of a problem type errors are in the grand scheme of things.
- Does the added cost of developing in a statically typed language make sense if type-related bugs are but a tiny fraction?
- Of the type-related bugs that occur, what proportion of those would have been solved by a type checker anyway?
- They aren't perfect after all.

Untyped?

Machine Code:

- At the end of the day, everything is zeros and ones.
  - At the physical level there is no conception of type
  - *In fact, even 0 and 1 are abstractions for voltage levels…*
- A machine instruction simply tells the CPU –
  - “*Perform <u>this</u> operation on the bits in <u>that</u> register*”.
- The CPU moves and manipulates bits (*electrical signals*).
- There is no “type” in any sense of the word.

Strong VS Weak Typing



Strong VS Weak (or *Loose*)

Refers to how strict statically typed languages are at compile time

There is no universally accepted definition of what constitutes strong or weak typing

Of **strongly** typed languages:

1974: “Whenever an object is passed from a calling function to a called function, its type must be ***compatible*** with the type declared in the called function."

Compatible is open to interpretation. Is float compatible with double? Integer with short integer?

Strong VS Weak (or *Loose*)

Refers to how strict statically typed languages are at compile time

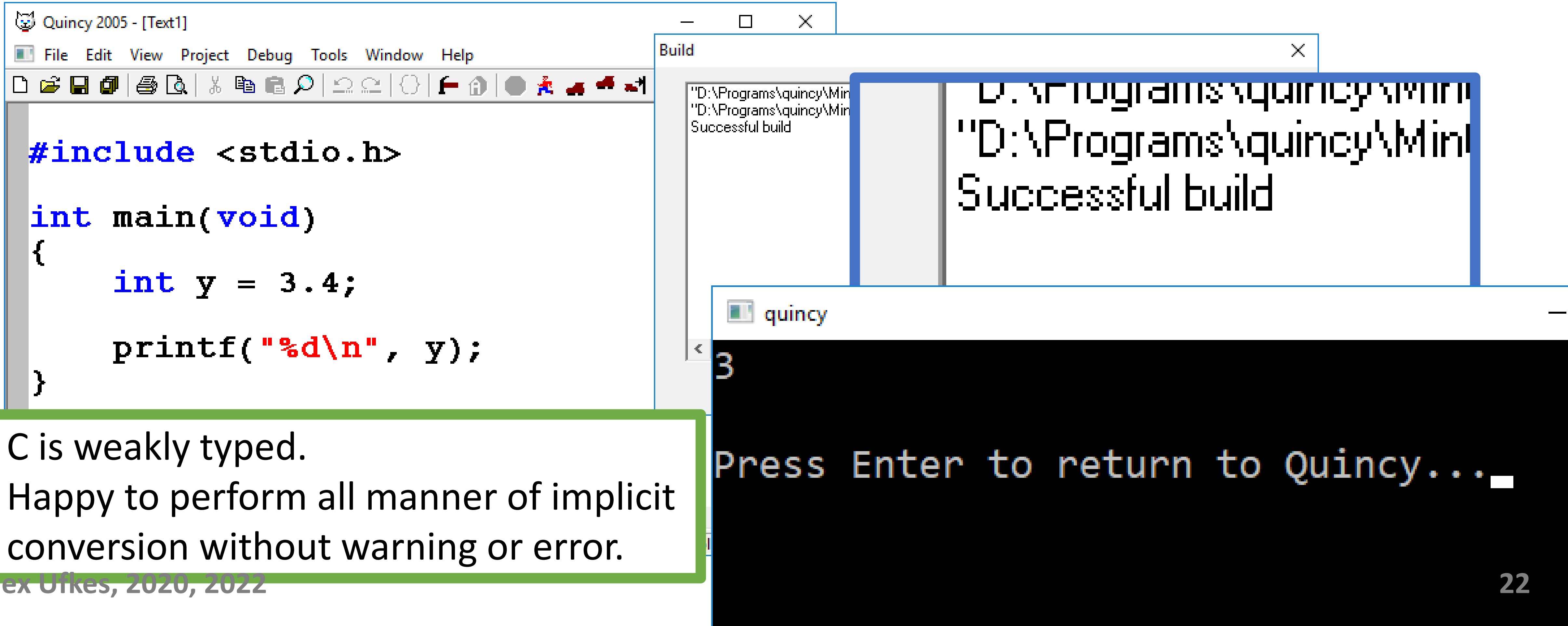
1974: “Whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function."

1977: “In a strongly typed language each data area will have a distinct type and each process will state its ***communication requirements*** in terms of these types."

Parameter lists, return types, etc.

Strong VS Weak (or *Loose*)

To what degree does a statically typed language allow implicit type conversion?



- C is weakly typed.
- Happy to perform all manner of implicit conversion without warning or error.

Strong VS Weak (or *Loose*)

In C, pointer arithmetic can be used to ***completely <u>bypass</u>*** the type system:

Quincy 2005 - [Text1]

```c
#include <stdio.h>

int main(void)
{
    int x = 3357986;

    short a = *((short*)&x) + 1;

    printf("%d\n", a);
}
```

Press F1 for help | Ln 13, Col 1

- We're using pointer arithmetic to read the first 2 bytes of an **int** as a **short**.
- This can be done with any two types.
- We can read the rightmost 4 bytes of a **double** as an **int**, etc.
- We can treat memory any way we want

quincy

```
15651

Press Enter to return to Quincy...
```

Strong VS Weak (or *Loose*)

C++ compilers will often give warnings, but programs still compile and run:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    int x = 57.99;

    print

    syste
}
```

```
guration: Debug Win32 ------

warning C4244: 'initializing' : conversion from 'double' to 'int', possible loss of data
est\Debug\test.exe
-to-date, 0 skipped ==========
```

```
C:\Users\aufke\Desktop\test\Debug\test.exe
57
Press any key to continue . . .
```

Strong VS Weak (or *Loose*)

Java will throw compile errors when a ***loss of precision*** occurs:

```
public class MethodTester
{
    public static void main(String[] args)
    {
        int a = 7.7;
        float b = 7.7;
        float c = 7.7f;
        double d = 7.7;
    }
}
```

- No implicit truncation from floating point to integer
- Floating point constants are double precision
- Need to indicate single precision explicitly

Java will throw compile errors when a loss of precision occurs:

<u>Careful!</u> Loss of precision does not ***only*** occur when going from floating point type to integer type!

`int` is 32 bits two's complement.
`float` is a 23-bit mantissa and an 8-bit exponent.

32-bit:

0   1 0 0 0 0 0 1 0   0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

sign bit — Exponent (8 bits) — Mantissa (23 bits)

```java
public class MethodTester
{
    public static void main(String[] args)
    {
        int a = 2111111111;
        System.out.println(a);

        float b = a;
        a = (int) b;

        System.out.println(a);
    }
}
```

BlueJ: Terminal Window - HelloWorld

Options

```
2111111111
2111111168
```

Can only ent

Imprecision of Floating Point

- Integers are represented *precisely.* The integer 42 is **exactly** 42.
- The single-precision (32 bits) floating point value 0.1 is ***actually*** 0.100000001490116119384765625
- *Double*-precision (64 bit) floating point values are more accurate, but still not perfect.

But why?

- Floating point values exist on an infinite continuum.
- Between any two floating point values are an ***infinite*** number of additional floating-point values.
- Integers are discrete. Between any two integers are a ***finite*** number of integers.

```
In [35]: 0.1 + 0.1 + 0.1
Out[35]: 0.30000000000000004

In [36]: 0.3
Out[36]: 0.3

In [37]:
```

Huh?

- Adding `0.1` three times accumulates rounding/representation errors.
- Echoing `0.3` on its own hasn't accumulated those errors.
- ***Even still:*** `0.3` is not precise in binary!
- The interactive shell just doesn't show all the trailing digits.

Imprecision of Floating Point

- A double-precision float is represented using 64 bits.
- A ***finite*** number of bits cannot represent an ***infinite*** number of floating point values.

0100101110000010100010100001010100011010101011000110001110000011

- There are **2^64** ways to arrange 64 bits. A large number to be sure, but certainly not infinite.

Infinite Integers?

But there are an infinite number of integers!

- 100% correct. We can't represent every possible integer either.
- Rather, there is a range. A standard 32-bit integer has a range of −2,147,483,648 to 2,147,483,647.
- Every integer within this range is represented precisely.
- Anything outside this range can't be represented using 32 bits
- If we try, we overflow.

Overflow

YouTube

Shared publicly - Dec 1, 2014

We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!

Strong VS Weak (or *Loose*)

Haskell uses type classes to achieve a level of type polymorphism:

WinGHCi

File Edit Actions Tools Help

```
Prelude> bigger x y = x < y
Prelude> :t bigger
bigger :: Ord a => a -> a -> Bool
Prelude> square x = x*x
Prelude> :t square
square :: Num a => a -> a
Prelude> square True

<interactive>:11:1: error:
    • No instance for (Num Bool) arising from a use of ‘square’
    • In the expression: square True
      In an equation for ‘it’: it = square True
Prelude>
```

- Rather than assign concrete types at compile time, assign a type class instead.
- If a function expects two args of type Ord, we can pass in any of the concrete types that implement Ord.

No mixed expressions once we assign concrete types!

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke> ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> x = 5::Float
Prelude> y = 5::Int
Prelude> x+y

<interactive>:4:3: error:
    * Couldn't match expected type `Float' with actual type `Int'
    * In the second argument of `(+)', namely `y'
      In the expression: x + y
      In an equation for `it': it = x + y
Prelude>
```

Strong VS Weak (or *Loose*)

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0277]: cannot add a float to an integer --> main.rs:2:16
  |
2 |     let r1 = 3 + 4.0;
  |                ^ no implementation for `{integer} + {float}`
  |
  = help: the trait `std::ops::Add<{float}>` is not implemented for `{integer}`

error: aborting due to previous

For more information about this

C:\_RustCode>_
```

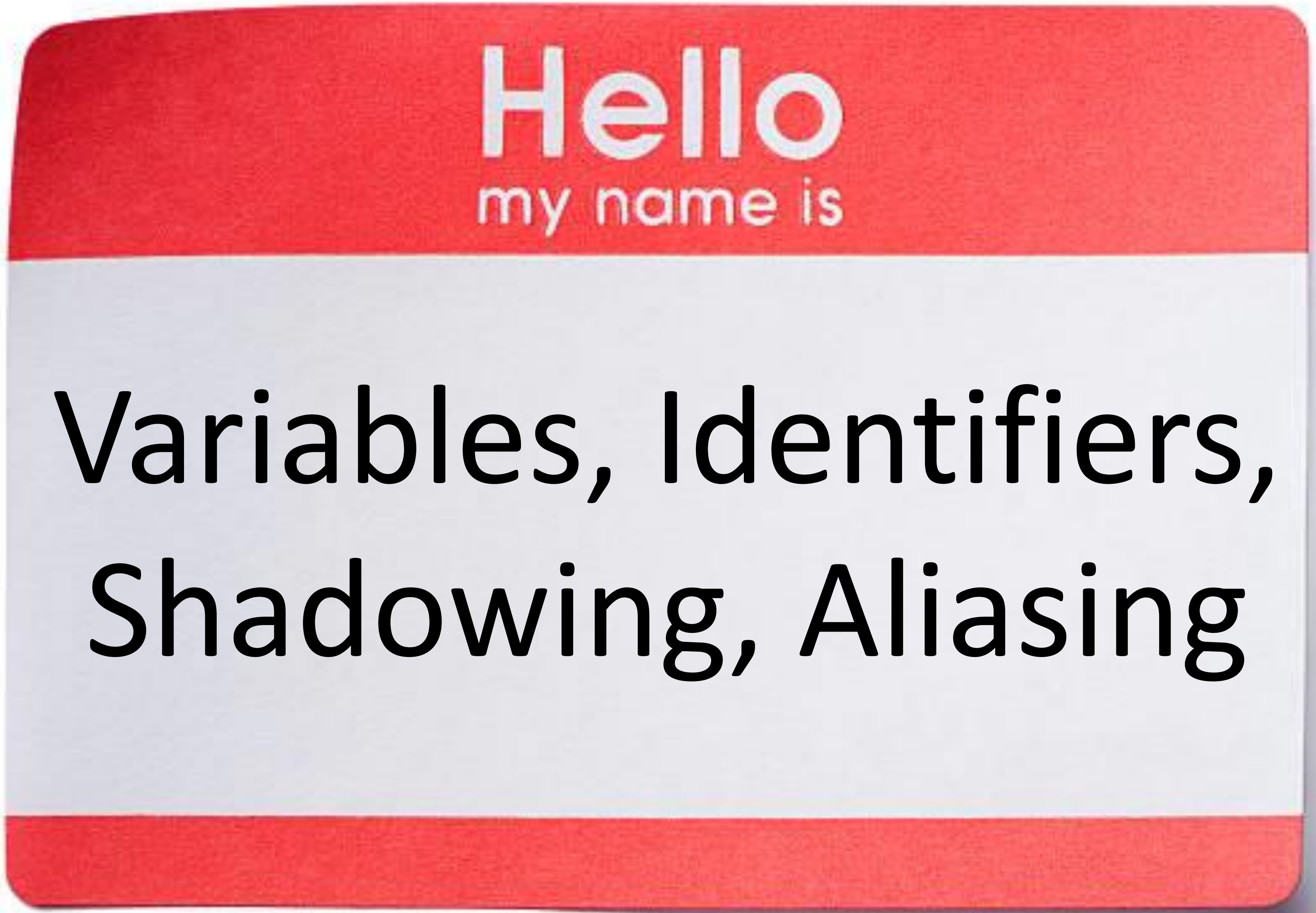
main.rs

```rust
fn main() {
    let r1 = 3 + 4.0;
    println!("r1: {}", r1);
}
```

Strong VS Weak (or *Loose*)

- Rust achieves type polymorphism using ***Traits***
- Directly inspired by type classes in Haskell
- Not inferred, must be indicated explictly.

```
fn max_val<T: PartialOrd + Copy> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```



Variables, Identifiers, Shadowing, Aliasing

Shadowing? Aliasing?

- A variable is a location associated with a name (identifier)
- When two names are associated with the same memory location, we call this ***aliasing***.
  - A variable has multiple names

- **Shadowing:** assign the same name to a different location in memory.
- Many languages allow this, but only when scope is different in some way.
- In Rust, we can "shadow" in the same scope.

Shadowing in C/C++

```c
#include <stdio.h>

int main(void)
{
	int x = 2;

	{
		int x = 5;
		printf("%d\n", x);
	}

	printf("%d\n", x);
}
```

```
5
2

Press Enter to return to Quincy...
```

- In C/C++, overlapping scope is fine.
- We will simply access the variable that is closest in scope to the statement doing the accessing.

Aliasing in C/C++

Quincy 2005 - [Text1]

```c
#include <stdio.h>

int main(void)
{
	int x = 2;
	int *a = &x;
	int *b = &x;

	printf("%d\n", *a);
	printf("%d\n", *b);
}
```

Press F1 for help

- Using pointers
- Both pointers refer to the same location.
- No problem using each pointer in the same scope at the same time.

quincy

```
2
2

Press Enter to return to Quincy...
```

Shadowing in Java

```java
public class Scope
{
    public static void main(String[] args)
    {
        int x = 2;

        {
            int x = 5;
            Syst
        }

        System.out.println(x);
    }
}
```

```
variable x is already defined in method
main(java.lang.String[])
```

- In Java, within the same method, we can only shadow if scopes do not overlap
- However, object instance variables can be shadowed by method local variables

Problem?

Shadowing in Java

```
public class Person
{
    public String name;
    public int age;

    public Person(String name, int age)
    {
        name = name;
        age = age;
    }
}
```

- Parameter names shadow instance variable names
- Not a compile error!
- Constructor is just pointlessly assigning the parameters to themselves.

this

Shadowing in Java

```java
public class Person
{
    public String name;
    public int age;

    public Person(String name,
    {
        name = name;
        age = age;
    }
}
```

```java
public class Person
{
    public String name;
    public int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

this lets us refer to the object instance variable rather than the parameter.

How about this?

Shadowing in Java

```
if (x >= 0)
{
    double r = Math.sqrt(x);
}
else
{
    float r = 0;
}
int r;
r = 0;
```

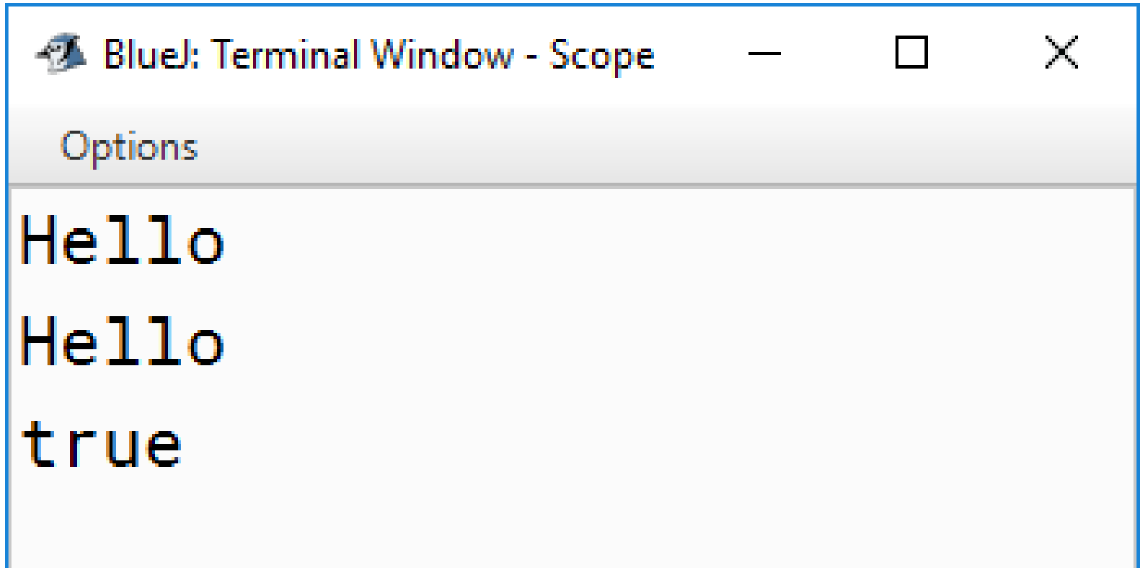
Scopes do not overlap!

Aliasing in Java

```java
public static void main(String[] args)
{
    String s1 = "Hello";
    String s2 = s1;

    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s1 == s2);
}
```

- No problem
- Two references, same object
- Both are valid
- Check equality to verify



Shadowing in Rust

```
fn main()
{
    let x = 7;      // immutable variable, can still shadow
    let x = "Hi";   // can even shadow to a different type!

    let mut y = 8;  // mutable variable, can reassign value
    y = y * 2;      // new value must match type

    // always immutable, must indicate type
    const V: u8 = 10;
}
```

- In Rust, we can shadow in the same scope!
- This is similar to rebinding in Elixir
- But, the old value of x is lost. What about…

Shadowing in Rust

```
fn main()
{
    let x = 7;

    {
        let x = "Hi";
        println!("{}", x);
    }

    println!("{}", x);
}
```

Shadowing over different scopes:

- Each scope preserves its own value
- Just like C/C++

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
Hi
7

C:\_RustCode>_
```

Aliasing in Rust

Aliasing is very different in Rust.

If the type implements trait Copy, we get a copy:

```
fn main()
{
    let x = 7;
    let mut y = x;
    y = 2;

    println!("{}, {}", x, y);
}
```

- 7 is i32, i32 implements trait Copy
- Thus, y is a new variable whose value is copied from x.
- Changing y does not affect x.

Command Prompt

```
C:\_RustCode>main
7, 2

C:\_RustCode>
```

Aliasing in Rust

Aliasing is very different in Rust, as we saw.

If the type does not implement trait Copy, we move ownership!

```rust
fn main()
{
    let s1 = String::from
    let s2 = s1;

    println!("{}", s1);
    println!("{}", s2);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0382]: use of moved value: `s1`
 --> main.rs:6:20
  |
4 |     let s2 = s1;
  |              -- value moved here
5 |
6 |     println!("{}", s1);
  |                    ^^ value used here after move
  |
```

Functions,

Methods,

Procedures

Practically speaking, there is little difference.

Purists will tell you otherwise, so let's see how they might be considered different

The difference comes mainly from the context in which they are used.

Functions VS Procedures

A “**function**” refers to a sub-program that returns *at least one value*.

- Comes from the mathematical notion of a function.
- Calculates a return value based on its inputs.

A “**procedure**” refers to a sub-program that executes commands.

- It effectively acts via side effect.
- Thus, a *procedure* in a *functional* language makes as much sense as a void function. That is, very little.

A function can be pure, a pure procedure would be useless by definition.

```rust
fn main()
{
    pure_procedure(7, 8);
}

fn pure_procedure (x: i32, y: i32)
{
    let z = x + y;
}
```

- A procedure returns nothing.
- For it to be pure, it can have no side effects.
- With no return value, and no side effects, a procedure is completely pointless.

- Some languages (Pascal) treat functions and procedures as distinct entities, which behave differently with respect to the language syntax.
- In C-like languages, there is no distinction. A procedure and a void function are the same thing.

Methods?

We can make a more meaningful distinction between functions and methods

- A method usually refers to a subroutine that is associated with an object in the OO paradigm.
- Thus, the concept of a method doesn't make sense in a language without objects, such as C.

- Rust, on the other hand, does not implement classes
- However, it makes a distinction between methods and functions.
- Methods are implemented over types (structs, enums, etc.), and carry in implicit reference parameter to that value.

Argument Passing

Many different strategies:
Call-by-value

- This is most common
- Values get copied into new variables
- Function can’t modify original argument
- Even when passing references (Java), we get a *copy* of the reference.
- Though they will point to the same object in memory.
- C, C++, Java, Smalltalk (technically)

Argument Passing

Many different strategies:

Call-by-value

Smalltalk (technically):

- Everything in Smalltalk is an object, thus we’re always passing references.
- It’s hard to think of this as pass-by-value.
- We’re still creating a copy of a reference, it’s still the same object in memory.
- Call-by-reference is different from passing a reference by value.

Argument Passing

Many different strategies:
Call-by-value
Call-by-reference

- Many languages support call-by-reference in some capacity
- Fortran II defaults to pass-by-reference. Modifying parameters affects original argument
- C++ defaults to call by value, but offers special syntax to pass by reference (&)
- C can simulate call-by-reference using pointers.
- Rust offers call by reference, but defaults to immutable. Special syntax allows us to pass mutable references.

Pass by Reference **VS** Passing a Reference by Value

```java
public class RefTester
{
    public static void funny(Point p1, Point p2)
    {
        p1.x = 10;    p1.y = 10;
        Point temp = p1;
        p1 = p2;
        p2 = temp;
    }

    public static void main(String[] args)
    {
        Point p1 = new Point(0,0);
        Point p2 = new Point(0,0);
        funny(p1, p2);
        System.out.println("x1: " + p1.x + " y1: " + p1.y);
        System.out.println("x2: " + p2.x + " y2: " + p2.y);
    }
}
```

Use the copied reference to modify the original object.

Swap which object each reference points to....?

- x1,y1 should be 0,0?
- x2,y2 should be 10,10?

BlueJ: Terminal Window - Scope

Options

```
x1: 10 y1: 10
x2: 0 y2: 0
```

```cpp
class Pt { public: int x, y; };

void swap(Pt & p1, Pt & p2)
{
    Pt temp = p1;
    p1 = p2;
    p2 = temp;
}

int main(void)
{
    Pt p1;  p1.x = 2;  p1.y = 2;
    Pt p2;  p2.x = 5;  p2.y = 5;

    printf("P1 = %d, %d\n", p1.x, p1.y);
    printf("P2 = %d, %d\n\n", p2.x, p2.y);

    swap(p1, p2);

    printf("P1 = %d, %d\n", p1.x, p1.y);
    printf("P2 = %d, %d\n\n", p2.x, p2.y);

    system("pause");
}
```

Compare to pass-by-reference in C++:

D:\GoogleDrive\Teaching - Humber\ATMN 253\

```
P1 = 2, 2
P2 = 5, 5

P1 = 5, 5
P2 = 2, 2

Press any key to continue . . .
```

Parameter Passing

Many different strategies:
Call-by-value
Call-by-reference
Call-by-name

- Arguments are not evaluated until function is called.
- Rather, they are substituted directly into the body of the function.
- Think #define macro style in C++
- If an argument is not used in the function body, it is never actually evaluated.
- Early example was ALGOL 60
- Consider:

```c
#include <stdio.h>
#include <windows.h>

void func (int x, int y)
{
    printf("%d", x);
}

int main(void)
{
    func(1 + 2 + 3 + 4 + 5, 8 - 7 * 9);

    system("pause");
}
```

If this was using call-by-name:

- Only the first argument is evaluated
- The expression 1+2+3+4+5 gets substituted into the function body.
- Downside? We might evaluate it more than once if it’s used more than once.
- Upside? Unused arguments are never evaluated.
- Argument might only be used in one branch of a selection structure, for example.
- One way to do lazy evaluation!

Parameter Passing

Many different strategies:
Call-by-value
Call-by-reference
Call-by-name
Call-by-need
Etc.

- In call-by-name, an expression might be evaluated multiple times.
- Call by need is just like call by name, except when an argument is evaluated once, that result is shared if the argument is used again.
- Requires more overhead behind the scenes.

Late VS Early Binding

Dynamic/late binding VS static/early binding

Early binding
- Method to be called is found at compile time
- Method not found = compile error
- More efficient at runtime

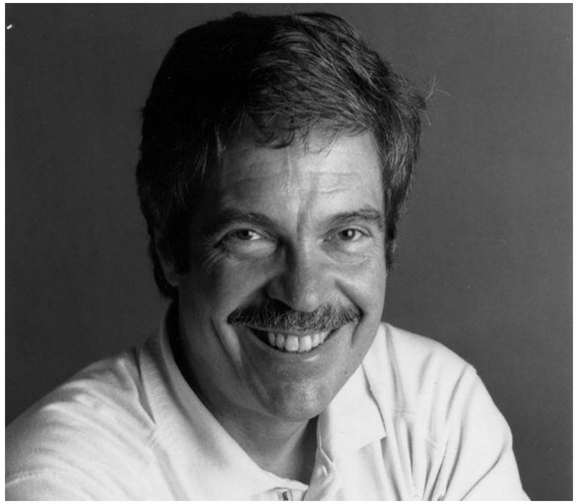
Late binding
- Method is looked up at runtime
- Often as simple is searching name
- Symbol comparison in Smalltalk
- Method not found = runtime error
- Costlier at runtime

Polymorphism is implemented through dynamic binding

Double dispatch is implemented through dynamic binding

Late VS Early Binding

Dynamic/late binding VS static/early binding



According to Kay, the essential ingredients of OOP are:

1. ***Message passing***
2. ***Encapsulation***
3. ***Dynamic binding***

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.."

Dynamic Binding

The difference between dynamic and static binding is easiest to appreciate through an example.

Consider…

AccountTester - BankAccount

Class Edit Tools Options

AccountTester ✕ BankAccount ✕ RRSP ✕

Compile Undo Cut Copy Paste Find... Close Source Code

```
public class AccountTester
{
    public static void main (String[] args)
    {
        RRSP acc1 = new RRSP();
        BankAccount acc2 = new BankAccount();
        acc1.deposit(10.00);

        atm(acc1, 2);
        atm(acc2, 3);
    }

    public static void atm(BankAccount b, double amt)
    {
        b.deposit(amt);
    }
}
```

```
public class BankAccount
{
    public double balance;
    public BankAccount() {
        balance = 0;
    }
    public void deposit(double balance) {
        this.balance += balance;
    }
}
```

```
public class RRSP extends BankAccount
{
    private static final double maxAnnual
        = 20000;

    @Override
    public void deposit(double amt) {
        if (super.balance + amt
                <= maxAnnual)
            super.deposit(amt);
    }
}
```

Class compiled - no syntax errors

saved

```java
public class AccountTester
{
    public static void main (String[] args)
    {
        RRSP acc1 = new RRSP();
        BankAccount acc2 = new BankAccount();
        acc1.deposit(10.00);

        atm(acc1, 2);
        atm(acc2, 3);
    }

    public static void atm(BankAccount b, double amt)
    {
        b.deposit(amt);
    }
}
```

What's going on here?

- BankAccount defines a deposit() method.
- Thus, subclasses of BankAccount do also.
- Could be overridden, could be inherited.
- However! Our atm() method can ***only*** invoke methods that exist in BankAccount.
- RRSP might define a new method not present in BankAccount.
- This could ***not*** be invoked from atm()

This gets confusing. Always remember the ***kind-of*** relationship!

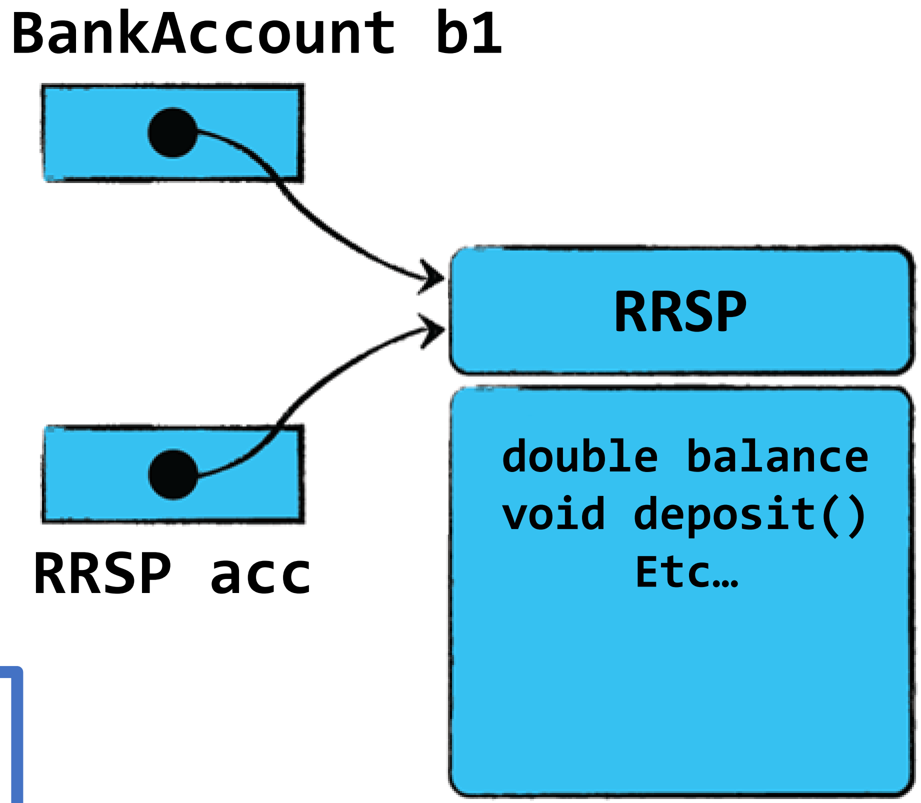
- An RRSP **is a** BankAccount, so any method designed for a BankAccount will/should also work on an RRSP.
- A BankAccount is *not necessarily* an RRSP. A method whose parameter is an RRSP will ***NOT*** accept a BankAccount.

Upcasting

```java
public class AccountTester
{
    public static void main (String[] args)
    {
        RRSP acc = new RRSP();
        BankAccount b1 = acc;
```

This is an implicit cast. We could also write:
`BankAccount b1 = (BankAccount) acc;`

```java
    }
```

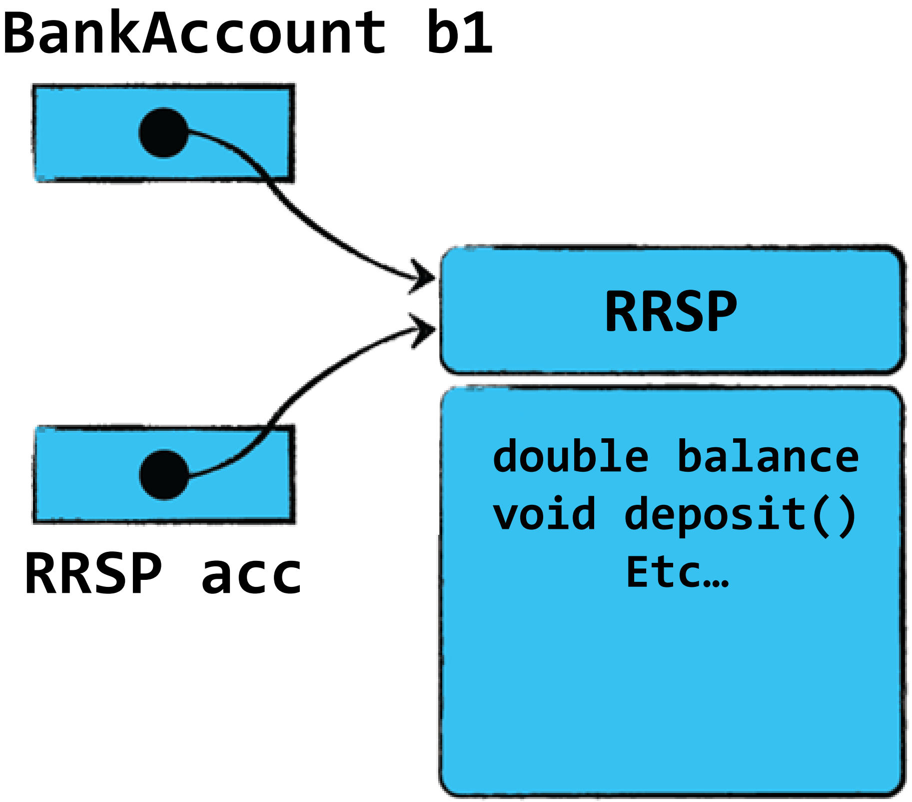


Upcasting

Upcasting: A sub-class object can be treated as an instance of its super-class

Why does this work?

- An RRSP can do everything a BankAccount can do
- If a BankAccount reference points to an RRSP object, there's no danger



Liskov Substitution Principle:
"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it"

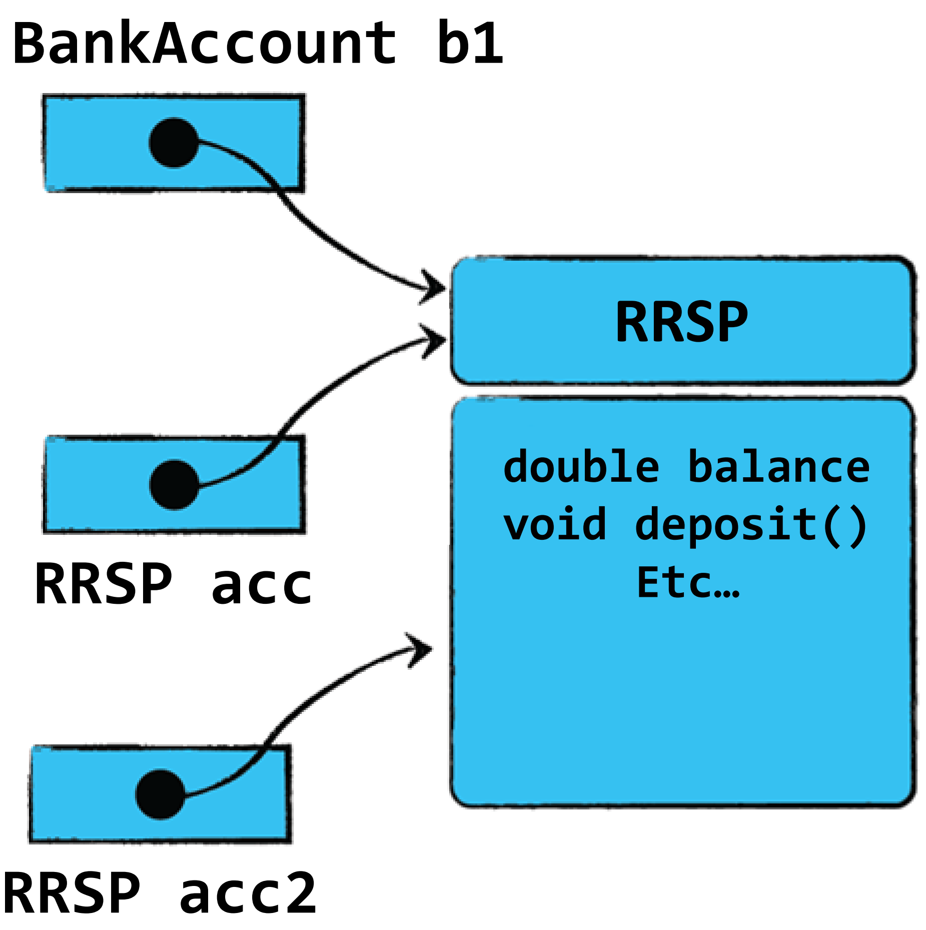
Downcasting?

```java
public class AccountTester
{
    public static void main (String[] args)
    {
        RRSP acc = new RRSP();

        BankAccount b1 = acc;

        RRSP acc2 = (RRSP) b1;
    }
}
```

We *do* have to explicitly cast here. *We* know that b1 references an RRSP object, but the *compiler* doesn't.

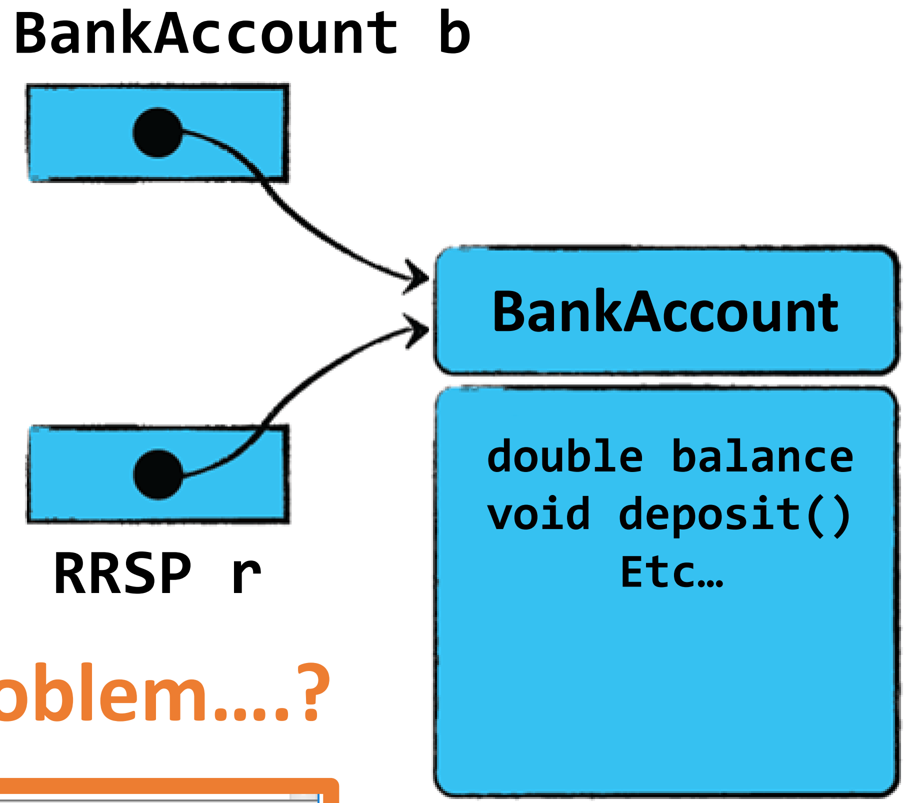


Downcasting?

```
public class AccountTester
{
    public static void main (String[] args)
    {
        BankAccount b = new BankAccount();

        RRSP r = (RRSP) b;

Class compiled - no syntax errors                    saved
    }
```



Problem….?

Downcasting?

```
public class AccountTester
{
    public static void main (String[] args)
    {
        BankAccount b = new BankAccount()
        RRSP r = (RRSP) b;
}
```

- Java accepts that we've cast reference b as an RRSP.
- It's not until *runtime* that Java follows the reference to the object and detects the problem.
- No other choice: the object doesn't actually exist at compile time.

```
Can only enter input while your programming is running
java.lang.ClassCastException: BankAccount cannot be cast to RRSP
        at AccountTester.main(AccountTester.java:8)
```

Moving on…

…to imperative.

Rust is an imperative language. However, we’ll see many cool features that remind us of the functional languages we’ve seen.