Types and Parallelism in Java

- What is a type?
  1. predefined or user-defined data structures
     - e.g. `enum {A, B, C};`
  2. the set of things the compiler knows about a value
  3. set of values (type $\equiv$ set)
  4. set of values + associated operations

```
2.
int n = __;
this is 32 bits long, the range is [-2^31, 2^31 - 1]
what is the range of n + 1? is it [-2^31 + 1, 2^31 - 1]?
no, this is undefined
------------------------------------------------------------
for (int i = INT_MAX - 1; <= INT_MAX; ++)
    print(i); <- UNDEFINED

def f(x,y):
    return x < y
^ there are no types check statically in python, this will compile but not run
```

primitive - built into the language (int, char, bool, float, double,$\ldots$)

- important differences in implementations
- limits in size based on platform
- `int = math_int (no limits on size) or int C float`
  constructed

Float in programming languages IEEE-794 F.P

representation:

- s - 0 (pos) or 1 (neg)
- $\pm 2^{e-127} \times 1.f$
- $0 < e < 255$
- f - $1.010011\ldots 10_2$

1	8	23
s	e	f

```
float x, y;
...
if (x != y)
    print(x - y); <- should always print nonzero
```

largest finite number:

s	e	f
$\varnothing$	254	$1111111...111...1$

outcomes:

- trap
- diverge to infinity (default)

```
float x = 1;
float y = 0;
return x/y;;
```

$$\frac{1}{-\varnothing} = -\infty$$

$\infty - \infty =$ `NaN`, not $\varnothing$, 1, ∞, or $-\infty$

$\pm$ `x == y && memcp(&x, &y, sizeof(x))`

`NaN x != y && ~memcp(...)`

`NaN != x for all x`

`NaN !- NaN`

Uses of types

- annotations
    - info for humans
    - info for compiler
    - can have effect on execution
        - `float x;`

        - `(int) x;`
- inference
    - `int i = ...; float f = i * f;`
- checking
    - prevents errors from having more serious consequences
    - static
        - guarantee: no type errors while running
        - faster execution
    - dynamic
        - more flexible
        - forgiving
    - compromise: do both
        - Java
            - mostly static
            - casts - `Object o = ...; (string) o;`
    - strongly typed
        - OCaml - statically
        - python - dynamically
    - not strongly typed
        - C, C++

Type Equivalence $T = U$

name equivalence:

```
struct s {int val; struct s *next}
struct t {int val; struct t * next}

struct s v;
struct t w;
v = w; <- wont work, diff. names
```

structural equivalence:

```
typedef int s;
typedef int t;
s x = ...;
```

```
t w = ...;
x = w; ,_ works, same internally
```

abstract vs exposed types:

- abstract - flexibility, modularity
- exposed - efficiency

Subtypes $T \subseteq U$

2 options:

- like subsets
- like subclass - has more operations than the supertype

```
type day = (Sun, Mon,...,Sat);
type weekday = Mon...Fri
int f(d: day)...;
weekday w = __;
f(w);
```

Polymorphism `f(x)`

-"function" whose implementation has many forms depending on types

```
FORTRAN:
cos(x)
cos($f) <- float
cos($d) <- double
^ function overloading

x - float
i - int
x * i <- wont normally work
x * (float(i)) <- done automatically
^ coercion
```

overloading:

- single name for multiple functions
  coercion:

- automatic or implicit conversions done by compiler

```
trouble c coercion:
uid_t = -1;
if (x < 0)
    print("ouch");

since it is being compared with an int, it is converted to UINT_MAX = 2^31 - 1 != 1

arctan(y, x) float, float | double, double
arctan(3.0, 5.9f) (double, float) | coercion + overloading

overloading:
int f(float, int); <- different
int f(int, float); <- different

which do we pick to convert to float or int?
```

Parametric polymorphism

- types have type parameters
  - `'a list`
- traditional Java
  - type `Object` is the root of all types
  - `Collection c __;` e.g. collection of strings

```
for(Iterator i = c.iterator; i.hasNext();)
    if ((String)(i.next()).length() == 1)
        i.remove();
```

Generic Types in Java

```
Collection <String> c = __;
for (Iterator i = c.iterator(); i.hasNext())
    if (i.next().length() == 1)
        i.remove();
```

generic types:

- OCaml, Java,. . .
- check and them compile
- cleaner
- less flexible

templates:

- C++,. . .
- instantiates and compiles a template, then does type checking
- every compile has different machine code