High-level Abstractions, Safety, or Performance

Mateusz Pusz
March 15, 2023

Disclaimer

I am not claiming that C++ is the best programming language out there. Each of the programming languages **has its use and areas of applicability where it proves the best**.

Disclaimer

I am not claiming that C++ is the best programming language out there. Each of the programming languages **has its use and areas of applicability where it proves the best**.

Let us not start the holy wars here 😉

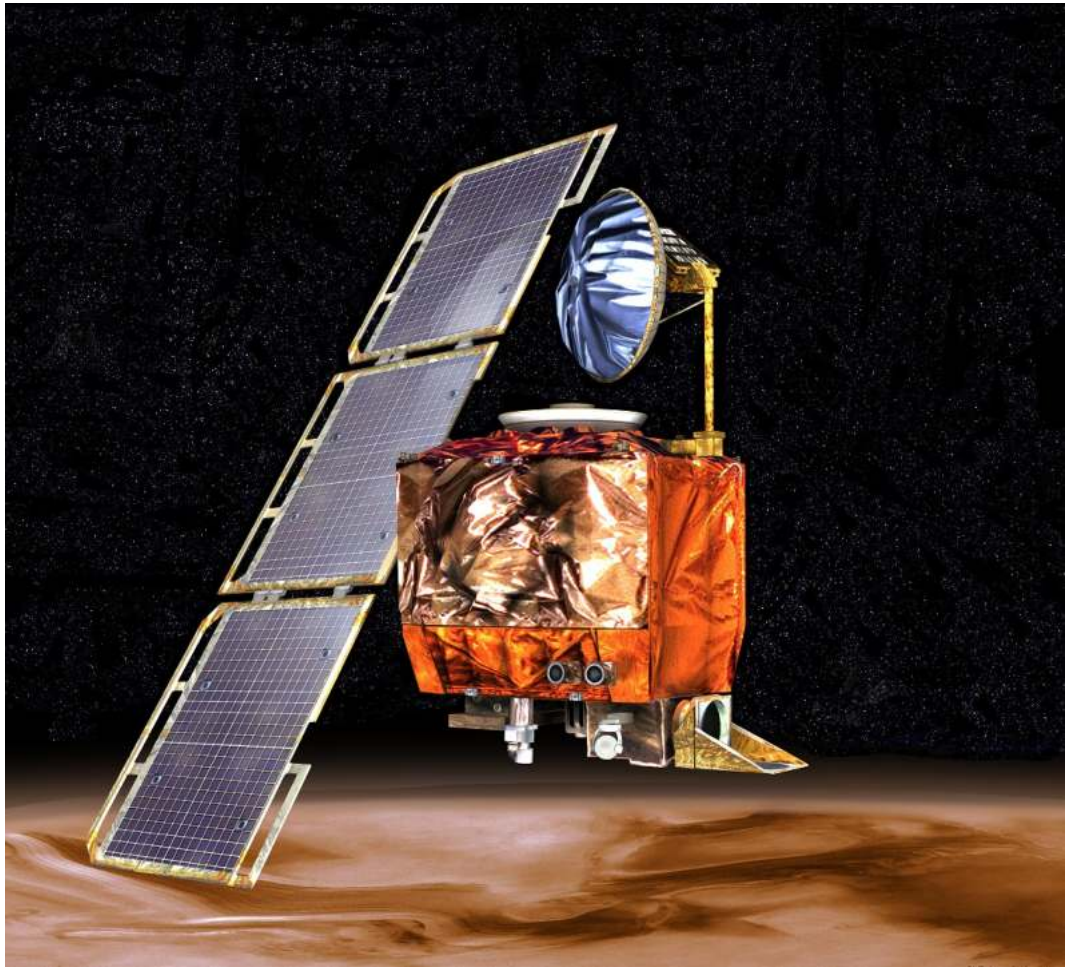
Agenda

1. Motivation
2. Physical Unit Libraries
3. High-level abstractions
4. Safety
5. Efficiency

MOTIVATION

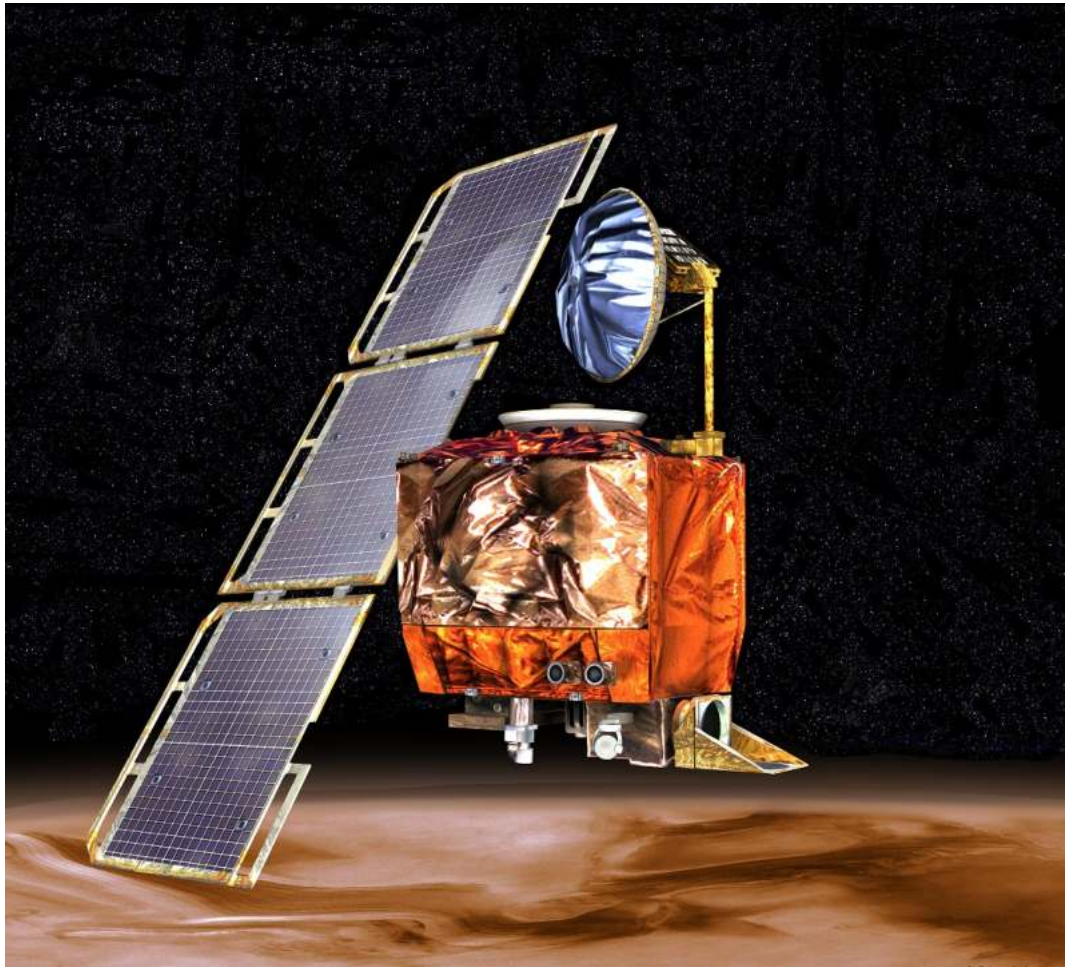
The Mars Climate Orbiter

- **Robotic space probe** launched by NASA on December 11, 1998



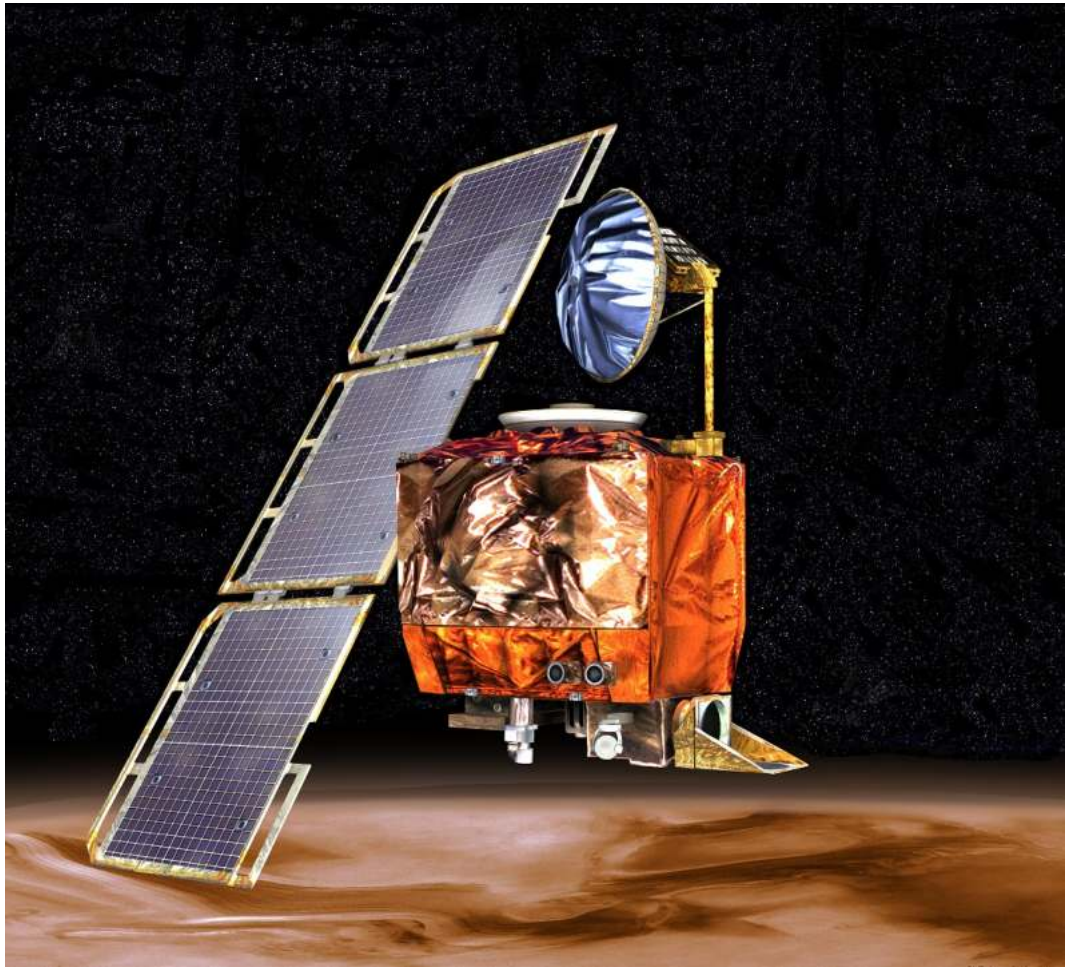
The Mars Climate Orbiter

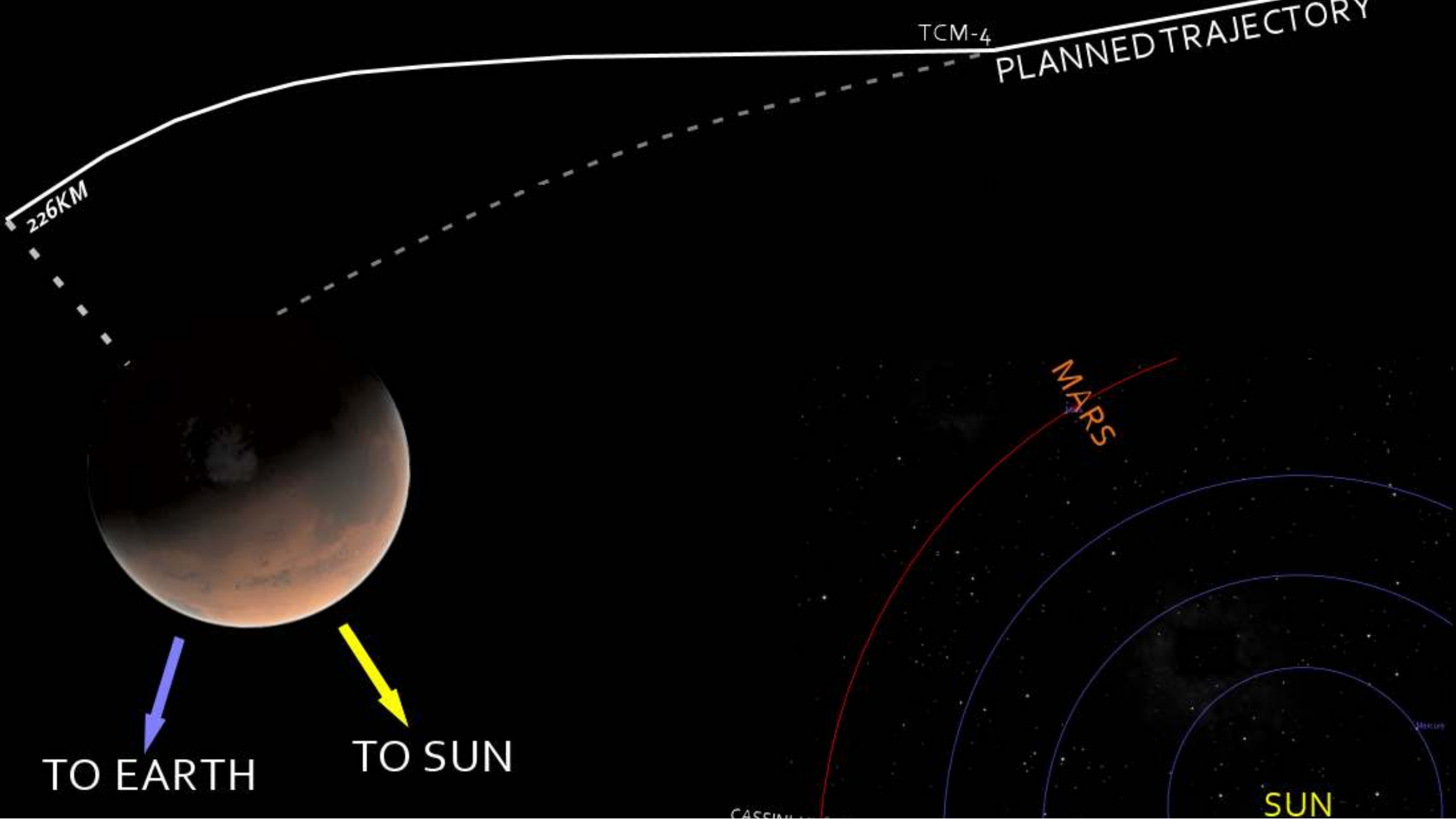
- **Robotic space probe** launched by NASA on December 11, 1998
- Project costs: **$327.6 million**
  - spacecraft development: $193.1 million
  - launching it: $91.7 million
  - mission operations: $42.8 million

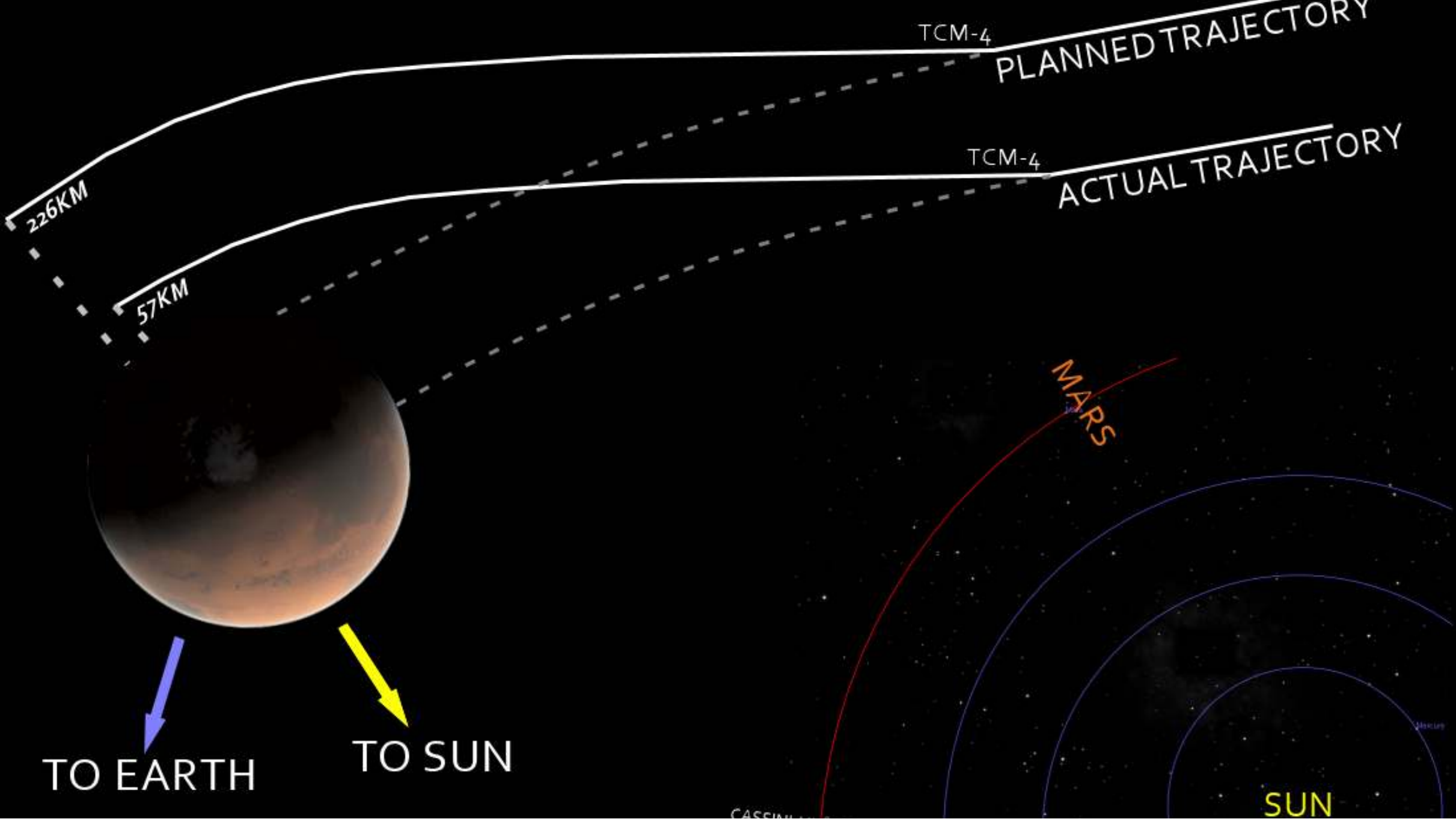


The Mars Climate Orbiter

- **Robotic space probe** launched by NASA on December 11, 1998
- Project costs: **$327.6 million**
  - spacecraft development: $193.1 million
  - launching it: $91.7 million
  - mission operations: $42.8 million
- Mars Climate Orbiter began the planned *orbital insertion maneuver* on September 23, 1999 at 09:00:46 UTC




TCM-4
PLANNED TRAJECTORY
226KM
MARS
TO EARTH
TO SUN
SUN


TCM-4
PLANNED TRAJECTORY
TCM-4
ACTUAL TRAJECTORY
226KM
57KM
MARS
TO EARTH
TO SUN
SUN

The Mars Climate Orbiter

- Space probe went **out of radio contact** when it passed behind Mars at 09:04:52 UTC, *49 seconds* earlier than expected
- Communication was never reestablished
- The **spacecraft disintegrated** due to atmospheric stresses

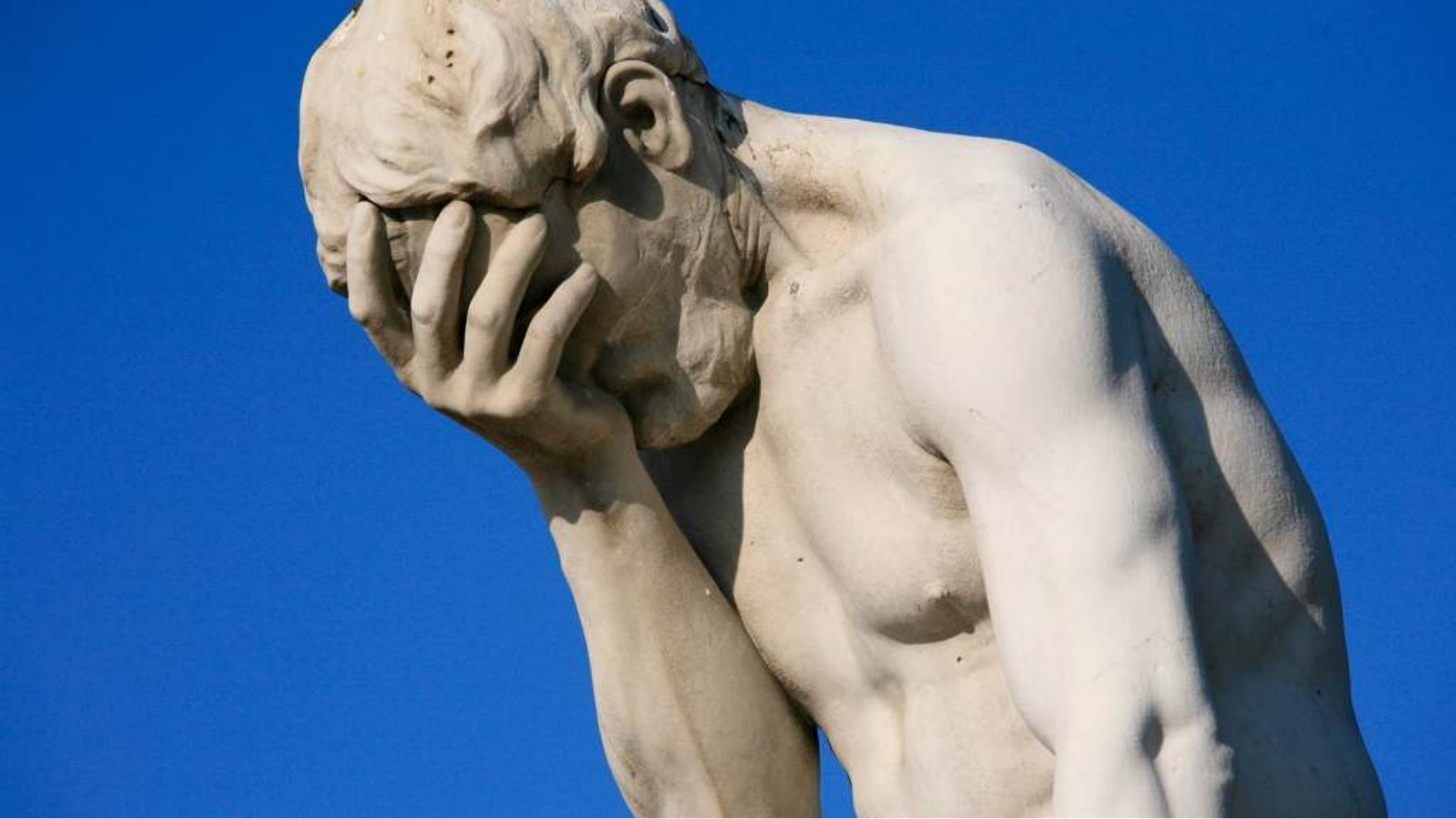
What went wrong?

What went wrong?

- The **primary cause** of this discrepancy was that
  - one piece of ground software supplied by Lockheed Martin produced results in a *United States customary unit*, **contrary to its Software Interface Specification** (SIS)
  - second system, supplied by NASA, expected those results to be in *SI units*, **in accordance** with the SIS

What went wrong?

- The **primary cause** of this discrepancy was that
  - one piece of ground software supplied by Lockheed Martin produced results in a *United States customary unit*, **contrary to its Software Interface Specification** (SIS)
  - second system, supplied by NASA, expected those results to be in *SI units*, **in accordance** with the SIS
- Specifically
  - software that calculated the total impulse produced by thruster firings calculated results in **pound-seconds**
  - the trajectory calculation software then used these results to update the predicted position of the spacecraft and expected it to be in **newton-seconds**



A long time ago in a galaxy far far away...

A long time ago in a galaxy far far away…

Tactical Flight Computer

Tactical Flight Computer

What is the correct order?

```
void DistanceBearing(double lat1, double lon1,
                     double lat2, double lon2,
                     double *Distance, double *Bearing);
```

What is the correct order?

```
void DistanceBearing(double lat1, double lon1,
                     double lat2, double lon2,
                     double *Distance, double *Bearing);
```

```
void FindLatitudeLongitude(double Lat, double Lon,
                           double Bearing, double Distance,
                           double *lat_out, double *lon_out);
```

What is the correct order?

```
void DistanceBearing(double lat1, double lon1,
                     double lat2, double lon2,
                     double *Distance, double *Bearing);
```

```
void FindLatitudeLongitude(double Lat, double Lon,
                           double Bearing, double Distance,
                           double *lat_out, double *lon_out);
```

`double` - an ultimate type to express quantity

```
double GlidePolar::MacCreadyAltitude(double emcready,
                                     double Distance,
                                     const double Bearing,
                                     const double WindSpeed,
                                     const double WindBearing,
                                     double *BestCruiseTrack,
                                     double *VMacCready,
                                     const bool isFinalGlide,
                                     double *TimeToGo,
                                     const double AltitudeAboveTarget,
                                     const double cruise_efficiency,
                                     const double TaskAltDiff);
```

We shouldn't write the code like this anymore

```
// Air Density(kg/m3) from relative humidity(%),
// temperature(°C) and absolute pressure(Pa)
double AirDensity(double hr, double temp, double abs_press)
{
  return (1/(287.06*(temp+273.15))) *
         (abs_press - 230.617 * hr * exp((17.5043*temp)/(241.2+temp)));
}
```

Now it is more important then ever

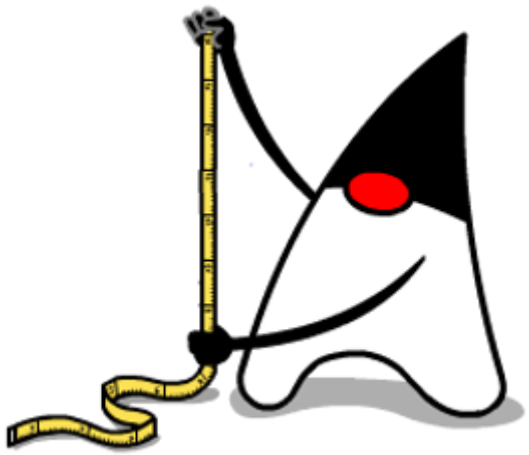
PHYSICAL UNIT LIBRARIES

Pint

- **Python** package to define, operate and manipulate physical quantities
- It allows **arithmetic operations** between them and **conversions from and to different units**
- It is distributed with a comprehensive **list of physical units, prefixes and constants**

JSR 385

- **The Unit of Measurement API**
- Provides a set of **Java** language programming interfaces for handling units and quantities
  - checking of **unit compatibility**
  - **expression of a quantity** in various units
  - **arithmetic operations** on units



mp-units

- **Modern C++** library
- Provides **compile-time dimensional analysis and unit/quantity manipulation**
  - no runtime execution or memory storage space cost is introduced
- Supports arbitrary
  - **systems of quantities**
  - **systems of units**
  - quantity number **representation types**
- Provide affine space quantity type abstractions
- Considered for the ISO standardization as a part of the ~~C++23~~ ~~C++26~~ **C++29** (???)

HIGH-LEVEL ABSTRACTIONS

Comparison Scenario #1: avg_speed

- Implement **`avg_speed`** function that
  - gets **length** and **time** quantities as parameters
  - divide **length** by **time** and ensure we actually got a quantity of **speed**
  - returns **speed** in the unit derived from the units of function arguments
- Calculate **`avg_speed(220 km, 2 h)`** and print the result in **km/h** and **m/s**
- Calculate **`avg_speed(140 mi, 2 h)`** and print the result in **mi/h** and **m/s**

Pint

```
ureg = UnitRegistry()
ureg.default_format = "~P"

@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d / t

s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
s2 = avg_speed(140 * ureg.mile, 2 * ureg.hour)

print(s1)
print(s2)
print(s1.to("metre/second"))
print(s2.to_base_units())
```

Pint

```
ureg = UnitRegistry()
ureg.default_format = "~P"

@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d / t

s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
s2 = avg_speed(140 * ureg.mile, 2 * ureg.hour)

print(s1)
print(s2)
print(s1.to("metre/second"))
print(s2.to_base_units())
```

```
110.0 km/hr
70.0 mi/hr
30.555555555555557 m/s
31.292800000000003 m/s
```

JSR 385

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
    return length.divide(time).asType(Speed.class);
}
```

```
final Quantity<Speed> s1 = avg_speed(Quantities.getQuantity(220, KILO(Units.METRE)),
                                     Quantities.getQuantity(2, Units.HOUR));
final Quantity<Speed> s2 = avg_speed(Quantities.getQuantity(140, MILE),
                                     Quantities.getQuantity(2, Units.HOUR));

System.out.println(s1);
System.out.println(s2);
System.out.println(s1.to(Units.METRE_PER_SECOND));
System.out.println(s2.toSystemUnit());
```

JSR 385

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
    return length.divide(time).asType(Speed.class);
}
```

```
final Quantity<Speed> s1 = avg_speed(Quantities.getQuantity(220, KILO(Units.METRE)),
                                     Quantities.getQuantity(2, Units.HOUR));
final Quantity<Speed> s2 = avg_speed(Quantities.getQuantity(140, MILE),
                                     Quantities.getQuantity(2, Units.HOUR));

System.out.println(s1);
System.out.println(s2);
System.out.println(s1.to(Units.METRE_PER_SECOND));
System.out.println(s2.toSystemUnit());
```

```
110 km/h
70 mi/h
30.555555555555555555555555555556 m/s
31.2928 m/s
```

mp-units: avg_speed

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d / t;
}
```

```
QuantityOf<isq::speed> auto s1 = avg_speed(220 * km, 2 * h);
QuantityOf<isq::speed> auto s2 = avg_speed(140 * mi, 2 * h);

std::print("{}\n", s1);
std::print("{}\n", s2);
std::print("{}\n", value_cast<double>(s1)[m / s]);
std::print("{}\n", value_cast<double>(s2)[m / s]);
```

mp-units: avg_speed

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d / t;
}
```

```
QuantityOf<isq::speed> auto s1 = avg_speed(220 * km, 2 * h);
QuantityOf<isq::speed> auto s2 = avg_speed(140 * mi, 2 * h);

std::print("{}\n", s1);
std::print("{}\n", s2);
std::print("{}\n", value_cast<double>(s1)[m / s]);
std::print("{}\n", value_cast<double>(s2)[m / s]);
```

```
110 km/h
70 mi/h
30.5556 m/s
31.2928 m/s
```

Comparison Scenario #2 ???

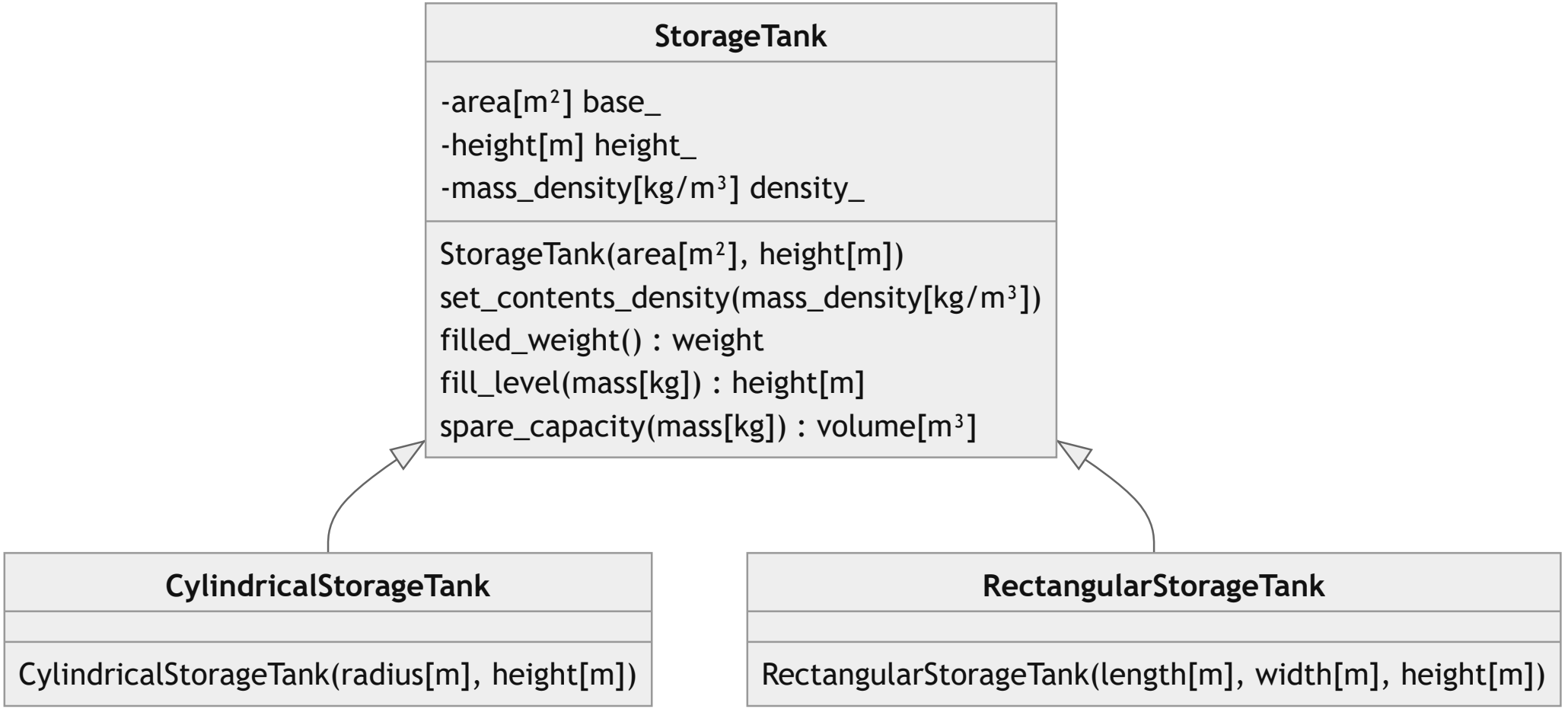
Comparison Scenario #2: Tanks???

Comparison Scenario #2: Storage Tanks

Comparison Scenario #2: Storage Tanks

Comparison Scenario #2: Storage Tanks

Comparison Scenario #2: Storage Tanks



Pint: Storage Tank

```
air_density = 1.225 * ureg.kg / ureg.m**3
g = ureg.standard_gravity

class StorageTank:
    # ...
```

Pint: Storage Tank

```
air_density = 1.225 * ureg.kg / ureg.m**3
g = ureg.standard_gravity

class StorageTank:
    # ...
```

```
class CylindricalStorageTank(StorageTank):
    @ureg.check(None, "[length]", "[length]")
    def __init__(self, radius, height):
        super().__init__(math.pi * radius * radius, height)


class RectangularStorageTank(StorageTank):
    @ureg.check(None, "[length]", "[length]", "[length]")
    def __init__(self, length, width, height):
        super().__init__(length * width, height)
```

Pint: Storage Tank

```
class StorageTank:
    @ureg.check(None, "[area]", "[length]")
    def __init__(self, base, height):
        self._base = base
        self._height = height
        self._density = air_density

    @ureg.check(None, "[density]")
    def set_contents_density(self, density):
        assert density > air_density
        self._density = density

    def filled_weight(self):
        volume = self._base * self._height
        mass = self._density * volume
        return mass * g

    @ureg.check(None, "[mass]")
    def fill_level(self, measured_mass):
        return self._height * measured_mass * g / self.filled_weight()

    @ureg.check(None, "[mass]")
    def spare_capacity(self, measured_mass):
        return (self._height - self.fill_level(measured_mass)) * self._base
```

Pint: Storage Tank

```
height = 200 * ureg.mm
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
tank.set_contents_density(1000 * ureg.kg / ureg.m**3)

fill_time = 200 * ureg.s
measured_mass = 20 * ureg.kg

fill_level = tank.fill_level(measured_mass)
spare_capacity = tank.spare_capacity(measured_mass)
filled_weight = tank.filled_weight()

input_flow_rate = measured_mass / fill_time
float_rise_rate = fill_level / fill_time
fill_time_left = (height / fill_level - 1) * fill_time
fill_ratio = fill_level / height
```

Pint: Storage Tank

```
print("fill height at {} = {} ({} full)".format(
    fill_time, fill_level.to("m"), fill_ratio.to_reduced_units()))
print("fill weight at {} = {} ({:.3~P})".format(
    fill_time, filled_weight.to_reduced_units(), filled_weight.to("N")))
print("spare capacity at {} = {}".format(fill_time, spare_capacity.to(ureg.m**3)))
print("input flow rate = {}".format(input_flow_rate))
print("float rise rate = {}".format(float_rise_rate.to("m/s")))
print("tank full E.T.A. at current flow rate = {}".format(fill_time_left))
```

Pint: Storage Tank

```
print("fill height at {} = {} ({} full)".format(
    fill_time, fill_level.to("m"), fill_ratio.to_reduced_units()))
print("fill weight at {} = {} ({:.3~P})".format(
    fill_time, filled_weight.to_reduced_units(), filled_weight.to("N")))
print("spare capacity at {} = {}".format(fill_time, spare_capacity.to(ureg.m**3)))
print("input flow rate = {}".format(input_flow_rate))
print("float rise rate = {}".format(float_rise_rate.to("m/s")))
print("tank full E.T.A. at current flow rate = {}".format(fill_time_left))
```

```
fill height at 200 s = 0.04 m (0.1999999999999998 full)
fill weight at 200 s = 100.0 g_0·kg (9.81×10² N)
spare capacity at 200 s = 0.08 m³
input flow rate = 0.1 kg/s
float rise rate = 0.0002 m/s
tank full E.T.A. at current flow rate = 800.0 s
```

JSR 385: Storage Tank

```
public class StorageTank {
    private static final Quantity<Density> airDensity = Quantities
            .getQuantity(1.225, Units.KILOGRAM.divide(Units.CUBIC_METRE)).asType(Density.class);
    private static final Quantity<Acceleration> g = Quantities.getQuantity(1, NonSI.STANDARD_GRAVITY);

    private final Quantity<Area> base;
    private final Quantity<Length> height;
    private Quantity<Density> density = airDensity;
// ...
}
```

JSR 385: Storage Tank

```
public class StorageTank {
    private static final Quantity<Density> airDensity = Quantities
            .getQuantity(1.225, Units.KILOGRAM.divide(Units.CUBIC_METRE)).asType(Density.class);
    private static final Quantity<Acceleration> g = Quantities.getQuantity(1, NonSI.STANDARD_GRAVITY);

    private final Quantity<Area> base;
    private final Quantity<Length> height;
    private Quantity<Density> density = airDensity;
// ...
}
```

```
public class CylindricalStorageTank extends StorageTank {
    public CylindricalStorageTank(Quantity<Length> radius, Quantity<Length> height) {
        super(radius.multiply(radius).multiply(Math.PI).asType(Area.class), height);
    }
}
```

JSR 385: Storage Tank

```
public class StorageTank {
    private static final Quantity<Density> airDensity = Quantities
            .getQuantity(1.225, Units.KILOGRAM.divide(Units.CUBIC_METRE)).asType(Density.class);
    private static final Quantity<Acceleration> g = Quantities.getQuantity(1, NonSI.STANDARD_GRAVITY);

    private final Quantity<Area> base;
    private final Quantity<Length> height;
    private Quantity<Density> density = airDensity;
// ...
}
```

```
public class CylindricalStorageTank extends StorageTank {
    public CylindricalStorageTank(Quantity<Length> radius, Quantity<Length> height) {
        super(radius.multiply(radius).multiply(Math.PI).asType(Area.class), height);
    }
}
```

```
public class RectangularStorageTank extends StorageTank {
    public RectangularStorageTank(Quantity<Length> length, Quantity<Length> width, Quantity<Length> height) {
        super(length.multiply(width).asType(Area.class), height);
    }
}
```

JSR 385: Storage Tank

```
public class StorageTank {
    // ...
    public StorageTank(Quantity<Area> base, Quantity<Length> height) {
        this.base = base;
        this.height = height;
    }

    public void setContentsDensity(ComparableQuantity<Density> density) {
        assert density.isGreaterThan(airDensity);
        this.density = density;
    }

    public Quantity<Force> filledWeight() {
        final Quantity<Volume> volume = base.multiply(height).asType(Volume.class);
        final Quantity<Mass> mass = density.multiply(volume).asType(Mass.class);
        return mass.multiply(g).asType(Force.class);
    }

    public Quantity<Length> fillLevel(Quantity<Mass> measuredMass) {
        return height.multiply(measuredMass).multiply(g).divide(filledWeight()).asType(Length.class);
    }

    public Quantity<Volume> spareCapacity(Quantity<Mass> measuredMass) {
        return height.subtract(fillLevel(measuredMass)).multiply(base).asType(Volume.class);
    }
}
```

JSR 385: Storage Tank

```
final Quantity<Length> height = Quantities.getQuantity(200, MILLI(Units.METRE));
StorageTank tank = new RectangularStorageTank(Quantities.getQuantity(1000, MILLI(Units.METRE)),
        Quantities.getQuantity(500, MILLI(Units.METRE)), height);
tank.setContentsDensity(
        Quantities.getQuantity(1000, Units.KILOGRAM.divide(Units.CUBIC_METRE)).asType(Density.class));

final Quantity<Time> fillTime = Quantities.getQuantity(200, Units.SECOND);
final Quantity<Mass> measuredMass = Quantities.getQuantity(20, Units.KILOGRAM);

final Quantity<Length> fillLevel = tank.fillLevel(measuredMass);
final Quantity<Volume> spareCapacity = tank.spareCapacity(measuredMass);
final Quantity<Force> filledWeight = tank.filledWeight();

final Quantity<MassFlowRate> inputFlowRate = measuredMass.divide(fillTime).asType(MassFlowRate.class);
final Quantity<Speed> floatRiseRate = fillLevel.divide(fillTime).asType(Speed.class);
Quantity<Dimensionless> one = Quantities.getQuantity(1, AbstractUnit.ONE);
Quantity<Dimensionless> scalar = height.divide(fillLevel).asType(Dimensionless.class);
final Quantity<Time> fillTimeLeft = fillTime.multiply(scalar.subtract(one).getValue());
final Quantity<?> fillRatio = fillLevel.divide(height);

// ...
```

JSR 385: Storage Tank

```
System.out.println("fill height at " + fillTime + " = " + fillLevel.to(Units.METRE) + " ("
        + fillRatio.asType(Dimensionless.class).to(Units.PERCENT) + " full)");
System.out.println("fill weight at " + fillTime + " = " + filledWeight + "("
        + filledWeight.to(Units.NEWTON) + ")");
System.out.println("spare capacity at " + fillTime + " = " + spareCapacity.to(Units.CUBIC_METRE));
System.out.println("input flow rate = " + inputFlowRate);
System.out.println("float rise rate = " + floatRiseRate.to(Units.METRE_PER_SECOND));
System.out.println("tank full E.T.A. at current flow rate = " + fillTimeLeft);
```

JSR 385: Storage Tank

```
System.out.println("fill height at " + fillTime + " = " + fillLevel.to(Units.METRE) + " ("
        + fillRatio.asType(Dimensionless.class).to(Units.PERCENT) + " full)");
System.out.println("fill weight at " + fillTime + " = " + filledWeight + "("
        + filledWeight.to(Units.NEWTON) + ")");
System.out.println("spare capacity at " + fillTime + " = " + spareCapacity.to(Units.CUBIC_METRE));
System.out.println("input flow rate = " + inputFlowRate);
System.out.println("float rise rate = " + floatRiseRate.to(Units.METRE_PER_SECOND));
System.out.println("tank full E.T.A. at current flow rate = " + fillTimeLeft);
```

```
fill height at 200 s = 0.04 m (20 % full)
fill weight at 200 s = 100000000000 kg·mm³·g\u2099/m³(980.665 N)
spare capacity at 200 s = 0.08 m³
input flow rate = 0.1 kg/s
float rise rate = 0.0002 m/s
tank full E.T.A. at current flow rate = 800000000000 s
```

JSR 385: Storage Tank

```
System.out.println("fill height at " + fillTime + " = " + fillLevel.to(Units.METRE) + " ("
        + fillRatio.asType(Dimensionless.class).to(Units.PERCENT) + " full)");
System.out.println("fill weight at " + fillTime + " = " + filledWeight + "("
        + filledWeight.to(Units.NEWTON) + ")");
System.out.println("spare capacity at " + fillTime + " = " + spareCapacity.to(Units.CUBIC_METRE));
System.out.println("input flow rate = " + inputFlowRate);
System.out.println("float rise rate = " + floatRiseRate.to(Units.METRE_PER_SECOND));
System.out.println("tank full E.T.A. at current flow rate = " + fillTimeLeft);
```

```
fill height at 200 s = 0.04 m (20 % full)
fill weight at 200 s = 100000000000 kg·mm³·g\u2099/m³(980.665 N)
spare capacity at 200 s = 0.08 m³
input flow rate = 0.1 kg/s
float rise rate = 0.0002 m/s
tank full E.T.A. at current flow rate = 800000000000 s
```

`fillTimeLeft` should be **`800 s`**. We found a bug in arithmetics on dimensionless quantities in the JSR 385 ;-)

mp-units: Storage Tank

```
inline constexpr auto g = 1 * si::standard_gravity;
inline constexpr auto air_density = isq::mass_density(1.225 * (kg / m3));

class StorageTank;
```

mp-units: Storage Tank

```
inline constexpr auto g = 1 * si::standard_gravity;
inline constexpr auto air_density = isq::mass_density(1.225 * (kg / m3));

class StorageTank;
```

```
struct CylindricalStorageTank : StorageTank {
  constexpr CylindricalStorageTank(const quantity<isq::radius[m]>& radius,
                                   const quantity<isq::height[m]>& height) :
      StorageTank(std::numbers::pi * radius * radius, height)
  {}
};

struct RectangularStorageTank : StorageTank {
  constexpr RectangularStorageTank(const quantity<isq::length[m]>& length,
                                   const quantity<isq::width[m]>& width,
                                   const quantity<isq::height[m]>& height) :
      StorageTank(length * width, height)
  {}
};
```

mp-units: Storage Tank

```
class StorageTank {
  quantity<isq::area[m2]> base_;
  quantity<isq::height[m]> height_;
  quantity<isq::mass_density[kg / m3]> density_ = air_density;
public:
  constexpr StorageTank(const quantity<isq::area[m2]>& base, const quantity<isq::height[m]>& height) :
      base_(base), height_(height)
  {}

  constexpr void set_contents_density(const quantity<isq::mass_density[kg / m3]>& density) {
    assert(density_in > air_density);
    density_ = density;
  }

  [[nodiscard]] constexpr QuantityOf<isq::weight> auto filled_weight() const {
    const QuantityOf<isq::mass> auto mass = density_ * isq::volume(base_ * height_);
    return isq::weight(mass * g);
  }

  [[nodiscard]] constexpr quantity<isq::height[m]> fill_level(const quantity<isq::mass[kg]>& measured_mass) const {
    return height_ * measured_mass * g / filled_weight();
  }

  [[nodiscard]] constexpr quantity<isq::volume[m3]> spare_capacity(const quantity<isq::mass[kg]>& measured_mass) const {
    return (height_ - fill_level(measured_mass)) * base_;
  }
};
```

mp-units: Storage Tank

```
const auto height = isq::height(200 * mm);
auto tank = RectangularStorageTank(isq::length(1000 * mm), isq::width(500 * mm), height);
tank.set_contents_density(1000 * isq::mass_density[kg / m3]);

const auto fill_time = 200 * s;        // time since starting fill
const auto measured_mass = 20. * kg;  // measured mass at fill_time

const auto fill_level = tank.fill_level(measured_mass);
const auto spare_capacity = tank.spare_capacity(measured_mass);
const auto filled_weight = tank.filled_weight();

const QuantityOf<isq::mass_change_rate> auto input_flow_rate = measured_mass / fill_time;
const QuantityOf<isq::speed> auto float_rise_rate = fill_level / fill_time;
const QuantityOf<isq::time> auto fill_time_left = (height / fill_level - 1) * fill_time;

const auto fill_percent = (fill_level / height)[percent];
```

mp-units: Storage Tank

```
std::print("fill height at {} = {} ({} full)\n", fill_time, fill_level, fill_percent);
std::print("fill weight at {} = {} ({})\n", fill_time, filled_weight, filled_weight[N]);
std::print("spare capacity at {} = {}\n", fill_time, spare_capacity);
std::print("input flow rate = {}\n", input_flow_rate);
std::print("float rise rate = {}\n", float_rise_rate);
std::print("tank full E.T.A. at current flow rate = {}\n", fill_time_left[s]);
```

mp-units: Storage Tank

```
std::print("fill height at {} = {} ({} full)\n", fill_time, fill_level, fill_percent);
std::print("fill weight at {} = {} ({})\n", fill_time, filled_weight, filled_weight[N]);
std::print("spare capacity at {} = {}\n", fill_time, spare_capacity);
std::print("input flow rate = {}\n", input_flow_rate);
std::print("float rise rate = {}\n", float_rise_rate);
std::print("tank full E.T.A. at current flow rate = {}\n", fill_time_left[s]);
```

```
fill height at 200 s = 0.04 m (20 % full)
fill weight at 200 s = 100 [g₀] kg (980.665 N)
spare capacity at 200 s = 0.08 m³
input flow rate = 0.1 kg/s
float rise rate = 0.0002 m/s
tank full E.T.A. at current flow rate = 800 s
```

Class members

PYTHON

```
class StorageTank:
    @ureg.check(None, "[area]", "[length]")
    def __init__(self, base, height):
        self._base = base
        self._height = height
        self._density = air_density
    # ...
```

Class members

PYTHON

```python
class StorageTank:
    @ureg.check(None, "[area]", "[length]")
    def __init__(self, base, height):
        self._base = base
        self._height = height
        self._density = air_density
    # ...
```

JAVA

```java
public class StorageTank {
    private final Quantity<Area> base;
    private final Quantity<Length> height;
    private Quantity<Density> density =
                                    airDensity;

    // ...
};
```

Class members

PYTHON

```python
class StorageTank:
    @ureg.check(None, "[area]", "[length]")
    def __init__(self, base, height):
        self._base = base
        self._height = height
        self._density = air_density
    # ...
```

JAVA

```java
public class StorageTank {
    private final Quantity<Area> base;
    private final Quantity<Length> height;
    private Quantity<Density> density =
                                    airDensity;
    // ...
};
```

C++

```cpp
class StorageTank {
  quantity<isq::area[m2]> base_;
  quantity<isq::height[m]> height_;
  quantity<isq::mass_density[kg / m3]> density_ = air_density;
  // ...
};
```

Return types

PYTHON

```python
class StorageTank:
    def filled_weight(self):
        # ...
    @ureg.check(None, "[mass]")
    def fill_level(self, measured_mass):
        # ...
```

Return types

PYTHON

```python
class StorageTank:
    def filled_weight(self):
        # ...
    @ureg.check(None, "[mass]")
    def fill_level(self, measured_mass):
        # ...
```

JAVA

```java
public class StorageTank {
    public Quantity<Force> filledWeight() {
      // ...
    }
    public Quantity<Length> fillLevel(Quantity<Mass> m) {
      // ...
    }
    // ...
}
```

Return types

PYTHON

```python
class StorageTank:
    def filled_weight(self):
        # ...
    @ureg.check(None, "[mass]")
    def fill_level(self, measured_mass):
        # ...
```

JAVA

```java
public class StorageTank {
    public Quantity<Force> filledWeight() {
      // ...
    }
    public Quantity<Length> fillLevel(Quantity<Mass> m) {
      // ...
    }
    // ...
}
```

C++

```cpp
class StorageTank {
public:
  [[nodiscard]] constexpr QuantityOf<isq::weight> auto filled_weight() const;
  [[nodiscard]] constexpr quantity<isq::height[m]> fill_level(const quantity<isq::mass[kg]>& m) const;
  // ...
};
```

Object instance creation

PYTHON

```
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
```

Object instance creation

PYTHON

```
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
```

JAVA

```
StorageTank tank = new RectangularStorageTank(Quantities.getQuantity(1000, MILLI(Units.METRE)),
                                              Quantities.getQuantity(500, MILLI(Units.METRE)), height);
```

Object instance creation

PYTHON

```
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
```

JAVA

```
StorageTank tank = new RectangularStorageTank(Quantities.getQuantity(1000, MILLI(Units.METRE)),
                                              Quantities.getQuantity(500, MILLI(Units.METRE)), height);
```

C++

```
auto tank = RectangularStorageTank(isq::length(1000 * mm), isq::width(500 * mm), height);
```

Object instance creation

PYTHON

```
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
```

JAVA

```
StorageTank tank = new RectangularStorageTank(Quantities.getQuantity(1000, MILLI(Units.METRE)),
                                              Quantities.getQuantity(500, MILLI(Units.METRE)), height);
```

C++

```
auto tank = RectangularStorageTank(isq::length(1000 * mm), isq::width(500 * mm), height);
```

```
auto tank = std::make_unique<RectangularStorageTank>(isq::length(1000 * mm), isq::width(500 * mm), height);
```

Object instance creation

PYTHON

```
tank = RectangularStorageTank(1000 * ureg.mm, 500 * ureg.mm, height)
```

JAVA

```
StorageTank tank = new RectangularStorageTank(Quantities.getQuantity(1000, MILLI(Units.METRE)),
                                              Quantities.getQuantity(500, MILLI(Units.METRE)), height);
```

C++

```
auto tank = RectangularStorageTank(isq::length(1000 * mm), isq::width(500 * mm), height);
```

```
auto tank = std::make_unique<RectangularStorageTank>(isq::length(1000 * mm), isq::width(500 * mm), height);
```

```
template<typename T>
using ptr_at = std::unique_ptr<T, decltype([](auto* p){ std::destroy_at(p); })>;

RectangularStorageTank* raw_storage = ...;
ptr_at<RectangularStorageTank> tank(std::construct_at(raw_storage, isq::length(1000 * mm), isq::width(500 * mm), height));
```

Arithmetics

PYTHON

```
fill_time_left = (height / fill_level - 1) * fill_time
```

Arithmetics

PYTHON

```
fill_time_left = (height / fill_level - 1) * fill_time
```

JAVA

```
Quantity<Dimensionless> one = Quantities.getQuantity(1, AbstractUnit.ONE);
Quantity<Dimensionless> scalar = height.divide(fillLevel).asType(Dimensionless.class);
final Quantity<Time> fillTimeLeft = fillTime.multiply(scalar.subtract(one).getValue());
```

Arithmetics

PYTHON

```
fill_time_left = (height / fill_level - 1) * fill_time
```

JAVA

```
Quantity<Dimensionless> one = Quantities.getQuantity(1, AbstractUnit.ONE);
Quantity<Dimensionless> scalar = height.divide(fillLevel).asType(Dimensionless.class);
final Quantity<Time> fillTimeLeft = fillTime.multiply(scalar.subtract(one).getValue());
```

C++

```
const auto fill_time_left = (height / fill_level - 1) * fill_time;
```

Arithmetics

PYTHON

```
fill_time_left = (height / fill_level - 1) * fill_time
```

JAVA

```
Quantity<Dimensionless> one = Quantities.getQuantity(1, AbstractUnit.ONE);
Quantity<Dimensionless> scalar = height.divide(fillLevel).asType(Dimensionless.class);
final Quantity<Time> fillTimeLeft = fillTime.multiply(scalar.subtract(one).getValue());
```

C++

```
const auto fill_time_left = (height / fill_level - 1) * fill_time;
```

```
const QuantityOf<isq::time> auto fill_time_left = (height / fill_level - 1) * fill_time;
```

SAFETY

Comparison Scenario

- Ensure that calling **`avg_speed`** with reordered arguments returns an error
- Ensure that constructing **`rectangular_storage_tank`** with reordered arguments returns an error
- Ensure that an error is reported when **`avg_speed`** returns the result of an invalid calculation
- Ensure that assigning incorrect quantity type to a local variable returns an error
- How much is `1 Hz + 1 Bq + 1 Bd`?

Pint – reordered arguments

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d / t

s = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
```

Pint – reordered arguments

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d / t

s = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
```

```
Traceback (most recent call last):
  File "safety_1.py", line 13, in <module>
    s1 = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 350, in wrapper
    raise DimensionalityError(value, "a quantity of", val_dim, dim)
pint.errors.DimensionalityError: Cannot convert from '2 hour' ([time]) to 'a quantity of' ([length])
```

Pint – reordered arguments

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d / t

s = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
```

```
Traceback (most recent call last):
  File "safety_1.py", line 13, in <module>
    s1 = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 350, in wrapper
    raise DimensionalityError(value, "a quantity of", val_dim, dim)
pint.errors.DimensionalityError: Cannot convert from '2 hour' ([time]) to 'a quantity of' ([length])
```

Runtime Error

JSR 385 – reordered arguments

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.divide(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(2., Units.HOUR),
                                      Quantities.getQuantity(220., KILO(Units.METRE)));
}
```

JSR 385 – reordered arguments

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.divide(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(2., Units.HOUR),
                                      Quantities.getQuantity(220., KILO(Units.METRE)));
}
```

```
The method avg_speed(Quantity<Length>, Quantity<Time>) in the type Safety_1 is not applicable for the arguments
(ComparableQuantity<Time>, ComparableQuantity<Length>)
```

JSR 385 – reordered arguments

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.divide(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(2., Units.HOUR),
                                      Quantities.getQuantity(220., KILO(Units.METRE)));
}
```

```
The method avg_speed(Quantity<Length>, Quantity<Time>) in the type Safety_1 is not applicable for the arguments
(ComparableQuantity<Time>, ComparableQuantity<Length>)
```

Compile-time Error

mp-units – reordered arguments

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d / t;
}

auto s = avg_speed(2 * h, 220 * km);
```

mp-units – reordered arguments

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d / t;
}

auto s = avg_speed(2 * h, 220 * km);
```

```
error: no matching function for call to ‘avg_speed(mp_units::quantity<mp_units::si::hour(), int>,
       mp_units::quantity<mp_units::si::kilo_<mp_units::si::metre()>(), int>)’
    44 |   constexpr auto v3 = avg_speed(2 * h, 220 * km);
       |                       ~~~~~~~~~^~~~~~~~~~~~~~~~~
note: candidate: ‘template<class auto:77, class auto:78>  requires (QuantityOf<auto:77, {}>) && (QuantityOf<auto:78, {}>)
      constexpr auto [requires mp_units::QuantityOf<<placeholder>, {}>] avg_speed(auto:77, auto:78)’
    32 | constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d, QuantityOf<isq::time> auto t)
       |                                       ^~~~~~~~~
note:   template argument deduction/substitution failed:
note: constraints not satisfied
...
```

mp-units – reordered arguments

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d / t;
}

auto s = avg_speed(2 * h, 220 * km);
```

```
error: no matching function for call to ‘avg_speed(mp_units::quantity<mp_units::si::hour(), int>,
       mp_units::quantity<mp_units::si::kilo_<mp_units::si::metre()>(), int>)’
    44 |   constexpr auto v3 = avg_speed(2 * h, 220 * km);
       |                       ~~~~~~~~~^~~~~~~~~~~~~~~~~
note: candidate: ‘template<class auto:77, class auto:78>  requires (QuantityOf<auto:77, {}>) && (QuantityOf<auto:78, {}>)
      constexpr auto [requires mp_units::QuantityOf<<placeholder>, {}>] avg_speed(auto:77, auto:78)’
    32 | constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d, QuantityOf<isq::time> auto t)
       |                                       ^~~~~~~~~
note:   template argument deduction/substitution failed:
note: constraints not satisfied
...
```

Compile-time Error

Pint – reordered arguments

```
tank = RectangularStorageTank(height, 1000 * ureg.mm, 500 * ureg.mm)
```

Pint – reordered arguments

```
tank = RectangularStorageTank(height, 1000 * ureg.mm, 500 * ureg.mm)
```

No error!

JSR 385 – reordered arguments

```
StorageTank tank = new RectangularStorageTank(height,
        Quantities.getQuantity(1000, MILLI(Units.METRE)),
        Quantities.getQuantity(500, MILLI(Units.METRE)));
```

JSR 385 – reordered arguments

```
StorageTank tank = new RectangularStorageTank(height,
        Quantities.getQuantity(1000, MILLI(Units.METRE)),
        Quantities.getQuantity(500, MILLI(Units.METRE)));
```

No error!

mp-units – reordered arguments

```
auto tank = RectangularStorageTank(height, isq::length(1000 * mm), isq::width(500 * mm));
```

mp-units – reordered arguments

```
auto tank = RectangularStorageTank(height, isq::length(1000 * mm), isq::width(500 * mm));
```

```
error: no matching function for call to ‘{anonymous}::RectangularStorageTank::RectangularStorageTank(
  const mp_units::quantity<mp_units::reference<mp_units::isq::height(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>&,
  mp_units::quantity<mp_units::reference<mp_units::isq::length(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>,
  mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>)’
   106 |   auto tank = RectangularStorageTank(height, isq::length(1000 * mm), isq::width(500 * mm));
       |                                                                                 ^
note: candidate: ‘constexpr {anonymous}::RectangularStorageTank::RectangularStorageTank(
  const mp_units::quantity<mp_units::reference<mp_units::isq::length(), mp_units::si::metre()>()>&,
  const mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::metre()>()>&,
  const mp_units::quantity<mp_units::reference<mp_units::isq::height(), mp_units::si::metre()>()>&)’
    90 |   constexpr RectangularStorageTank(const quantity<isq::length[m]>& length, const quantity<isq::width[m]>& width,
       |             ^~~~~~~~~~~~~~~~~~~~~~
note:   no known conversion for argument 2 from ‘mp_units::quantity<mp_units::reference<mp_units::isq::length(),
  mp_units::si::milli_<mp_units::si::metre()>()>(), int>’ to
  ‘const mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::metre()>()>&’
    90 |   constexpr RectangularStorageTank(const quantity<isq::length[m]>& length, const quantity<isq::width[m]>& width,
       |                                                                             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~
...
```

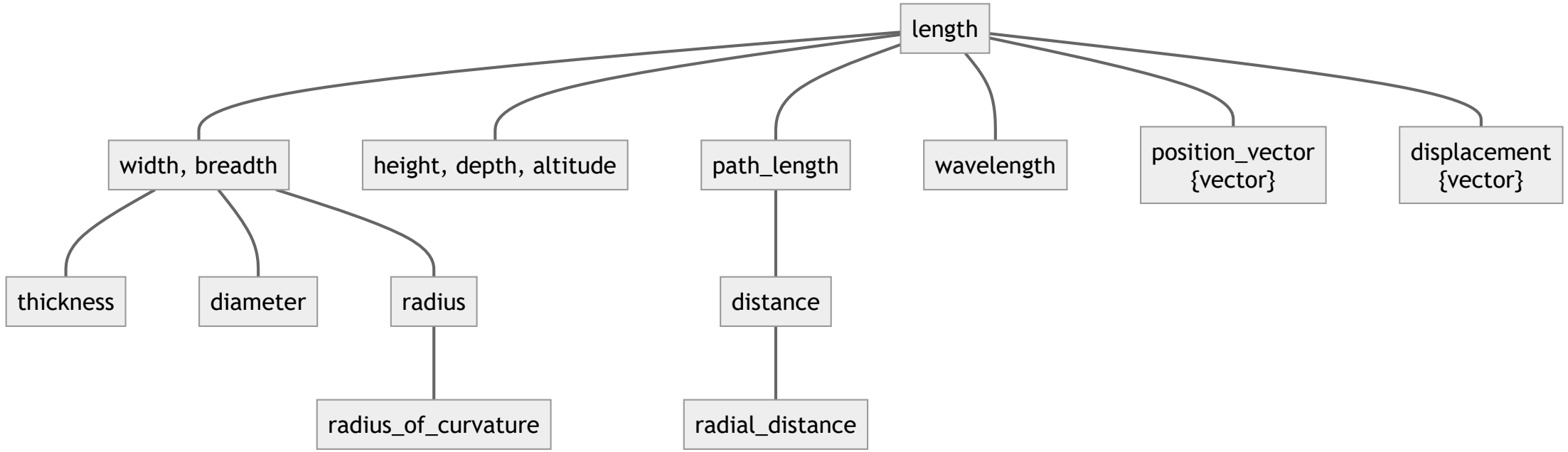
mp-units – reordered arguments

```
auto tank = RectangularStorageTank(height, isq::length(1000 * mm), isq::width(500 * mm));
```

```
error: no matching function for call to ‘{anonymous}::RectangularStorageTank::RectangularStorageTank(
  const mp_units::quantity<mp_units::reference<mp_units::isq::height(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>&,
  mp_units::quantity<mp_units::reference<mp_units::isq::length(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>,
  mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::milli_<mp_units::si::metre()>()>(), int>)’
   106 |   auto tank = RectangularStorageTank(height, isq::length(1000 * mm), isq::width(500 * mm));
       |                                                                                 ^
note: candidate: ‘constexpr {anonymous}::RectangularStorageTank::RectangularStorageTank(
  const mp_units::quantity<mp_units::reference<mp_units::isq::length(), mp_units::si::metre()>()>&,
  const mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::metre()>()>&,
  const mp_units::quantity<mp_units::reference<mp_units::isq::height(), mp_units::si::metre()>()>&)’
    90 |   constexpr RectangularStorageTank(const quantity<isq::length[m]>& length, const quantity<isq::width[m]>& width,
       |             ^~~~~~~~~~~~~~~~~~~~~~
note:   no known conversion for argument 2 from ‘mp_units::quantity<mp_units::reference<mp_units::isq::length(),
  mp_units::si::milli_<mp_units::si::metre()>()>(), int>’ to
  ‘const mp_units::quantity<mp_units::reference<mp_units::isq::width(), mp_units::si::metre()>()>&’
    90 |   constexpr RectangularStorageTank(const quantity<isq::length[m]>& length, const quantity<isq::width[m]>& width,
       |                                                                             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~
...
```

Compile-time Error

ISQ: International System of Quantities (ISO 80000)



Pint – invalid quantity equation

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d * t

s = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

Pint – invalid quantity equation

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  return d * t

s = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

No error!

Pint – invalid quantity equation

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d * t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

Pint – invalid quantity equation

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d * t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

```
Traceback (most recent call last):
  File "safety_2.py", line 14, in <module>
    s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 351, in wrapper
    return func(*args, **kwargs)
  File "safety_2.py", line 10, in avg_speed
    raise RuntimeError("Not a [speed] dimension")
RuntimeError: Not a [speed] dimension
```

Pint – invalid quantity equation

```
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d * t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

```
Traceback (most recent call last):
  File "safety_2.py", line 14, in <module>
    s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 351, in wrapper
    return func(*args, **kwargs)
  File "safety_2.py", line 10, in avg_speed
    raise RuntimeError("Not a [speed] dimension")
RuntimeError: Not a [speed] dimension
```

Runtime Error

JSR 385 – invalid quantity equation

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.multiply(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(220., KILO(Units.METRE)),
                                      Quantities.getQuantity(2., Units.HOUR));
}
```

JSR 385 – invalid quantity equation

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.multiply(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(220., KILO(Units.METRE)),
                                      Quantities.getQuantity(2., Units.HOUR));
}
```

```
Exception in thread "main" java.lang.ClassCastException: The unit: km·h is not compatible with quantities of type interface
javax.measure.quantity.Speed
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:277)
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:89)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:337)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:114)
        at zed2020.Safety_2.avg_speed(Safety_2.java:37)
        at zed2020.Safety_2.main(Safety_2.java:41)
```

JSR 385 – invalid quantity equation

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.multiply(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(220., KILO(Units.METRE)),
                                      Quantities.getQuantity(2., Units.HOUR));
}
```

```
Exception in thread "main" java.lang.ClassCastException: The unit: km·h is not compatible with quantities of type interface
javax.measure.quantity.Speed
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:277)
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:89)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:337)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:114)
        at zed2020.Safety_2.avg_speed(Safety_2.java:37)
        at zed2020.Safety_2.main(Safety_2.java:41)
```

Runtime Error

mp-units – invalid quantity equation

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d * t;
}

auto s = avg_speed(220 * km, 2 * h);
```

mp-units – invalid quantity equation

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d * t;
}

auto s = avg_speed(220 * km, 2 * h);
```

```
In instantiation of ‘constexpr auto [requires mp_units::QuantityOf<<placeholder>, {}>] avg_speed(auto:77, auto:78)
[with auto:77 = mp_units::quantity<mp_units::si::kilo_<mp_units::si::metre()>(), int>;
      auto:78 = mp_units::quantity<mp_units::si::hour(), int>]’:
  required from here
error: deduced return type does not satisfy placeholder constraints
    34 |   return d * t;
       |              ^
note: constraints not satisfied
...
```

mp-units – invalid quantity equation

```
constexpr QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto d,
                                                QuantityOf<isq::time> auto t)
{
  return d * t;
}

auto s = avg_speed(220 * km, 2 * h);
```

```
In instantiation of ‘constexpr auto [requires mp_units::QuantityOf<<placeholder>, {}>] avg_speed(auto:77, auto:78)
[with auto:77 = mp_units::quantity<mp_units::si::kilo_<mp_units::si::metre()>(), int>;
      auto:78 = mp_units::quantity<mp_units::si::hour(), int>]’:
  required from here
error: deduced return type does not satisfy placeholder constraints
    34 |   return d * t;
       |              ^
note: constraints not satisfied
...
```

Compile-time Error

Pint - assigning incorrect quantity type to a local variable

```
filled_mass = tank.filled_weight()
```

Pint - assigning incorrect quantity type to a local variable

```
filled_mass = tank.filled_weight()
```

No Error!

Pint - assigning incorrect quantity type to a local variable

```
filled_mass = tank.filled_weight()
if not filled_mass.check("[mass]"):
    raise RuntimeError("Not a [mass] dimension")
```

Pint - assigning incorrect quantity type to a local variable

```
filled_mass = tank.filled_weight()
if not filled_mass.check("[mass]"):
    raise RuntimeError("Not a [mass] dimension")
```

```
Traceback (most recent call last):
  File "/home/mpusz/repos/zed_2020/python/storage_tank.py", line 84, in <module>
    raise RuntimeError("Not a [mass] dimension")
RuntimeError: Not a [mass] dimension
```

Pint - assigning incorrect quantity type to a local variable

```
filled_mass = tank.filled_weight()
if not filled_mass.check("[mass]"):
    raise RuntimeError("Not a [mass] dimension")
```

```
Traceback (most recent call last):
  File "/home/mpusz/repos/zed_2020/python/storage_tank.py", line 84, in <module>
    raise RuntimeError("Not a [mass] dimension")
RuntimeError: Not a [mass] dimension
```

Runtime Error

JSR 385 – assigning incorrect quantity type to a local variable

```
final Quantity<Mass> filledMass = tank.filledWeight();
```

JSR 385 – assigning incorrect quantity type to a local variable

```
final Quantity<Mass> filledMass = tank.filledWeight();
```

```
Type mismatch: cannot convert from Quantity<Force> to Quantity<Mass>
```

JSR 385 – assigning incorrect quantity type to a local variable

```
final Quantity<Mass> filledMass = tank.filledWeight();
```

```
Type mismatch: cannot convert from Quantity<Force> to Quantity<Mass>
```

Compile-time Error

mp-units - assigning incorrect quantity type to a local variable

```
const QuantityOf<isq::mass> auto filled_mass = tank.filled_weight();
```

mp-units - assigning incorrect quantity type to a local variable

```
const QuantityOf<isq::mass> auto filled_mass = tank.filled_weight();
```

```
error: deduced initializer does not satisfy placeholder constraints
   116 |   const QuantityOf<isq::mass> auto filled_mass = tank.filled_weight();
       |                                                        ^
note: constraints not satisfied
   required for the satisfaction of ‘QuantityOf<auto [requires mp_units::QuantityOf<<placeholder>, {}>], mp_units::isq::mass()>’
   [with auto [requires mp_units::QuantityOf<<placeholder>, {}>] = mp_units::quantity<{}, double>]
note: no operand of the disjunction is satisfied
    58 |   Quantity<Q> && ((Dimension<std::remove_const_t<decltype(V)>> && Q::dimension == V) ||
       |                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
    59 |                    (QuantitySpec<std::remove_const_t<decltype(V)>> && implicitly_convertible_to(Q::quantity_spec, V)));
       |                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: the operand ‘(Dimension<typename std::remove_const<decltype (V1)>::type> && (Q::dimension == V))’ is unsatisfied because
    58 |   Quantity<Q> && ((Dimension<std::remove_const_t<decltype(V)>> && Q::dimension == V) ||
       |                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~
    59 |                    (QuantitySpec<std::remove_const_t<decltype(V)>> && implicitly_convertible_to(Q::quantity_spec, V)));
       |                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

mp-units - assigning incorrect quantity type to a local variable

```
const QuantityOf<isq::mass> auto filled_mass = tank.filled_weight();
```

```
error: deduced initializer does not satisfy placeholder constraints
   116 |   const QuantityOf<isq::mass> auto filled_mass = tank.filled_weight();
       |                                                                ^
note: constraints not satisfied
   required for the satisfaction of ‘QuantityOf<auto [requires mp_units::QuantityOf<<placeholder>, {}>], mp_units::isq::mass()>’
   [with auto [requires mp_units::QuantityOf<<placeholder>, {}>] = mp_units::quantity<{}, double>]
note: no operand of the disjunction is satisfied
    58 |   Quantity<Q> && ((Dimension<std::remove_const_t<decltype(V)>> && Q::dimension == V) ||
       |                  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
    59 |                   (QuantitySpec<std::remove_const_t<decltype(V)>> && implicitly_convertible_to(Q::quantity_spec, V)));
       |                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: the operand ‘(Dimension<typename std::remove_const<decltype (V1)>::type> && (Q::dimension == V))’ is unsatisfied because
    58 |   Quantity<Q> && ((Dimension<std::remove_const_t<decltype(V)>> && Q::dimension == V) ||
       |                  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~
    59 |                   (QuantitySpec<std::remove_const_t<decltype(V)>> && implicitly_convertible_to(Q::quantity_spec, V)));
       |                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

Compile-time Error

Pint – different quantity kinds

```
v = 1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud
print(v)
```

Pint – different quantity kinds

```
v = 1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud
print(v)
```

```
3.0 Hz
```

Pint – different quantity kinds

```
v = 1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud
print(v)
```

```
3.0 Hz
```

No Error!

JSR 385 – different quantity kinds

```
final Quantity<?> v = Quantities.getQuantity(1, Units.HERTZ)
    .add(Quantities.getQuantity(1, Units.BECQUEREL)); // Baud not provided by default
System.out.println(v);
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The method add(Quantity<Frequency>) in the type ComparableQuantity<Frequency> is not applicable for the arguments
	(ComparableQuantity<Radioactivity>)

	at zed2020.Safety_3.main(Safety_3.java:32)
```

JSR 385 – different quantity kinds

```
final Quantity<?> v = Quantities.getQuantity(1, Units.HERTZ)
    .add(Quantities.getQuantity(1, Units.BECQUEREL)); // Baud not provided by default
System.out.println(v);
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The method add(Quantity<Frequency>) in the type ComparableQuantity<Frequency> is not applicable for the arguments
	(ComparableQuantity<Radioactivity>)

	at zed2020.Safety_3.main(Safety_3.java:32)
```

Runtime-time Error

mp-units - different quantity kinds

```
auto q = 1 * Hz + 1 * Bq + 1 * Bd;
std::print("{}\n", q);
```

mp-units - different quantity kinds

```
auto q = 1 * Hz + 1 * Bq + 1 * Bd;
std::print("{}\n", q);
```

```
error: no match for ‘operator+’ (operand types are ‘mp_units::quantity<mp_units::si::hertz(), int>’ and
       ‘mp_units::quantity<mp_units::si::becquerel(), int>’)
    44 |   auto q = 1 * Hz + 1 * Bq + 1 * Bd;
       |            ~~~~~~ ^ ~~~~~~
       |              |       |
       |              |       quantity<mp_units::si::becquerel(),[...]>
       |              quantity<mp_units::si::hertz(),[...]>
...
```

mp-units - different quantity kinds

```
auto q = 1 * Hz + 1 * Bq + 1 * Bd;
std::print("{}\n", q);
```

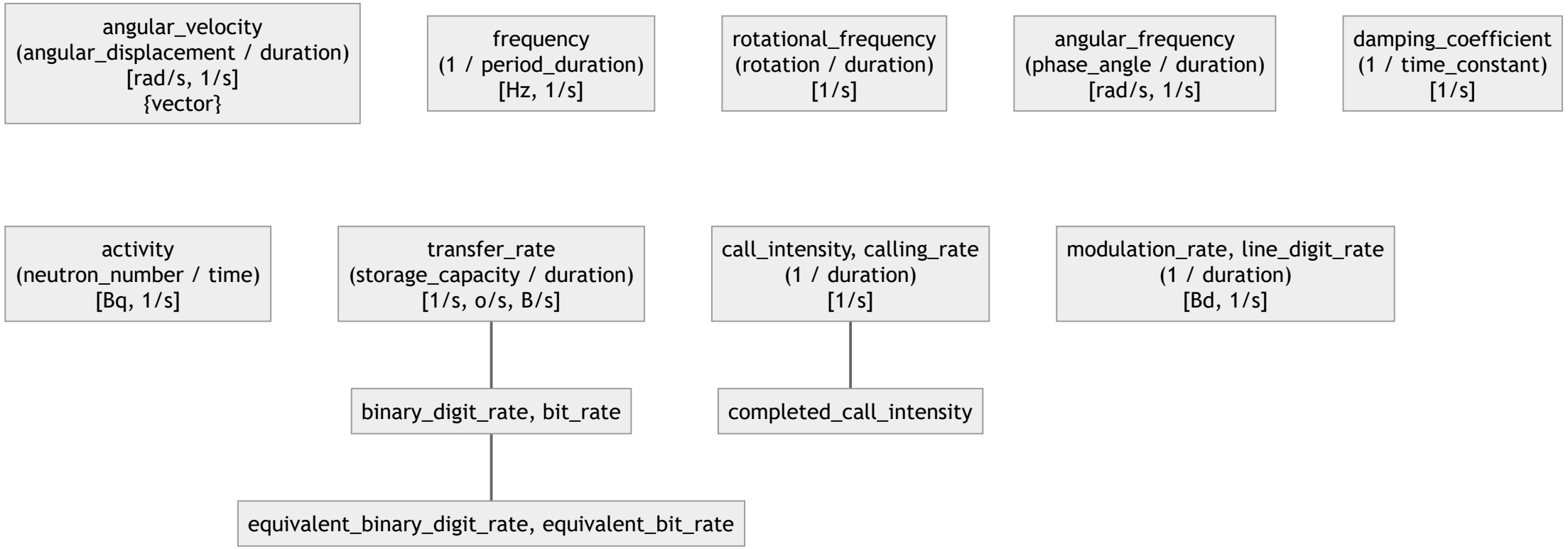
```
error: no match for ‘operator+’ (operand types are ‘mp_units::quantity<mp_units::si::hertz(), int>’ and
       ‘mp_units::quantity<mp_units::si::becquerel(), int>’)
    44 |   auto q = 1 * Hz + 1 * Bq + 1 * Bd;
       |            ~~~~~~ ^ ~~~~~~
       |              |       |
       |              |       quantity<mp_units::si::becquerel(),[...]>
       |              quantity<mp_units::si::hertz(),[...]>
...
```

Compile-time Error

Quantity kinds (ISO 80000)

- Quantities may be grouped together into **categories** of quantities that are **mutually comparable**
- Mutually comparable quantities are called **quantities of the same kind**
- Two or more quantities cannot be **added or subtracted** unless they belong to the same category of mutually comparable quantities
- Quantities of the same kind within a given system of quantities **have the same quantity dimension**
- Quantities of the same dimension are **not necessarily of the same kind**

ISQ quantities of dimension T^{-1}



Quantity kinds in mp-units

mp-units is probably the first library on the Open Source market (in any programming language)

- that properly supports quantity kinds
- provides definitions of all ISQ quantities defined in ISO 80000

Safety: Summary

	PINT	JSR 385	MP-UNITS
Reordered arguments (1)	Runtime	Compile-time	Compile-time
Reordered arguments (2)	No	No	Compile-time
Invalid quantity equation	Runtime (manual check)	Runtime	Compile-time
Assigning incorrect quantity type to a local variable	Runtime (manual check)	Compile-time	Compile-time
Different quantity kinds	No	Runtime	Compile-time

Safety: Summary

	PINT	JSR 385	MP-UNITS
Reordered arguments (1)	Runtime	Compile-time	Compile-time
Reordered arguments (2)	No	No	Compile-time
Invalid quantity equation	Runtime (manual check)	Runtime	Compile-time
Assigning incorrect quantity type to a local variable	Runtime (manual check)	Compile-time	Compile-time
Different quantity kinds	No	Runtime	Compile-time

Runtime errors require a lot of unit tests.
Compile-time errors prevent shipping incorrect product to customers (no product to ship if it does not compile 😉).

EFFICIENCY

Comparison Scenario

- Benchmark the following scenarios both for operations on fundamental/primitive types and on high-level quantity abstractions
  - **Arithmetic** - create quantities of **length** and **time** and divide them to obtain **speed**
  - **Scaling** - create a quantity of **speed** and convert the unit from **km/h** to **m/s**

Pint

Pint can impose a significant performance overhead on computationally-intensive problems. The following are some suggestions for getting the best performance.

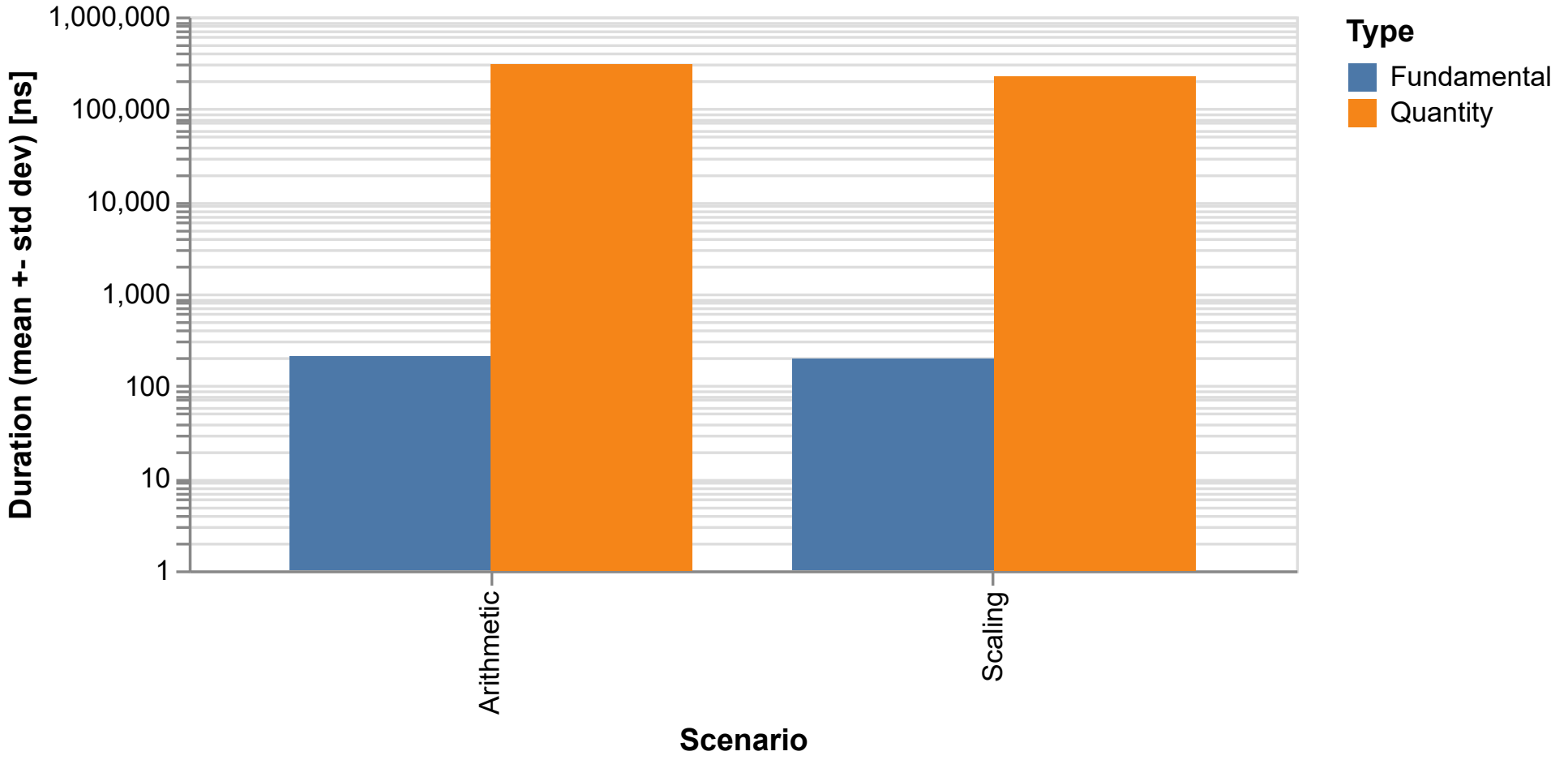
Use magnitudes when possible

It's significantly faster to perform mathematical operations on magnitudes (even though your'e still using pint to retrieve them from a quantity object).

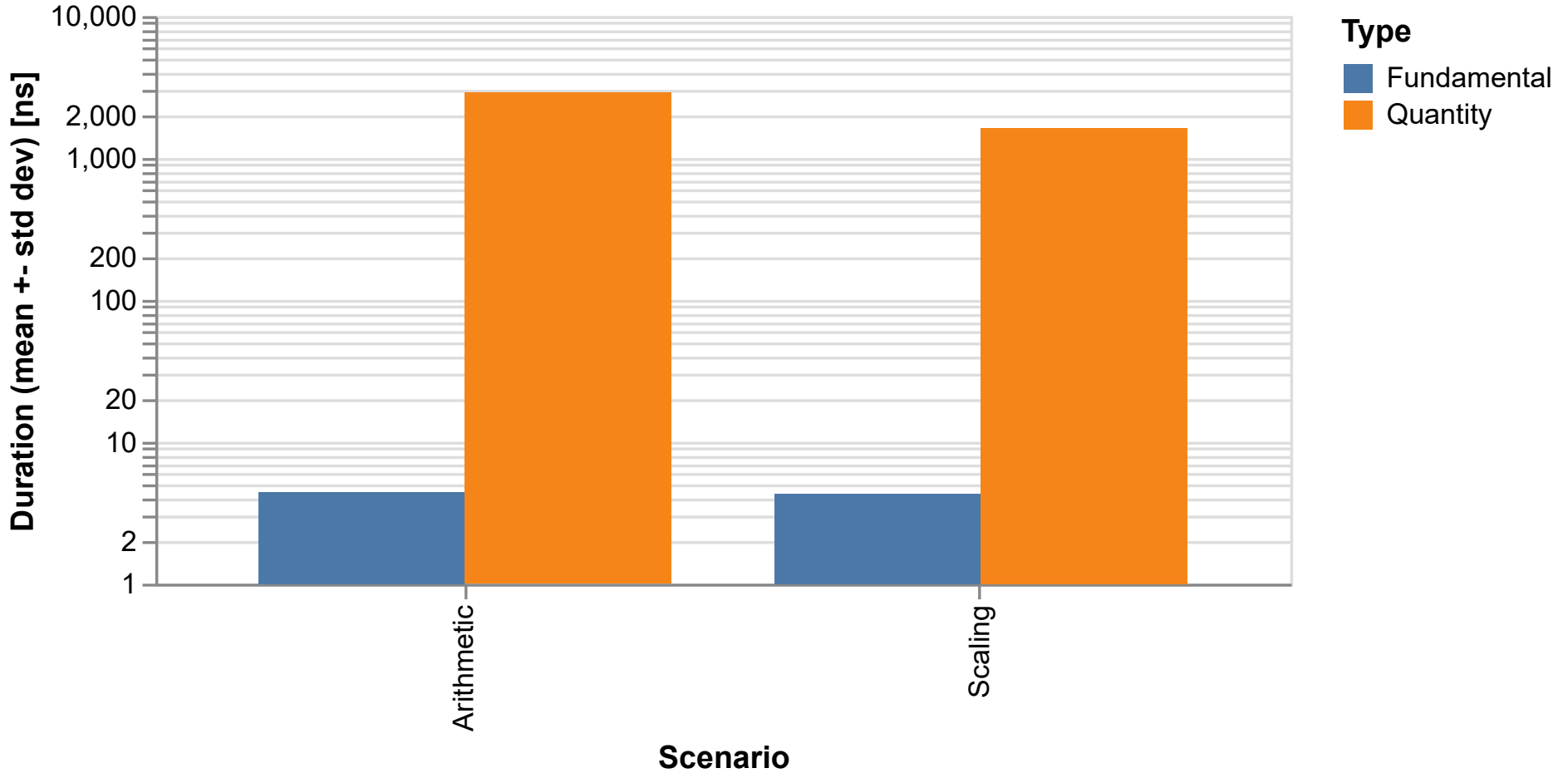
```
In [1]: from pint import UnitRegistry

In [2]: ureg = UnitRegistry()

In [3]: q1 =ureg('1m')

In [5]: q2=ureg('2m')

In [6]: %timeit (q1-q2)
100000 loops, best of 3: 7.9 µs per loop

In [7]: %timeit (q1.magnitude-q2.magnitude)
1000000 loops, best of 3: 356 ns per loop
```

Bear in mind that altering computations like this **loses the benefits of automatic unit conversion**, so use with care.

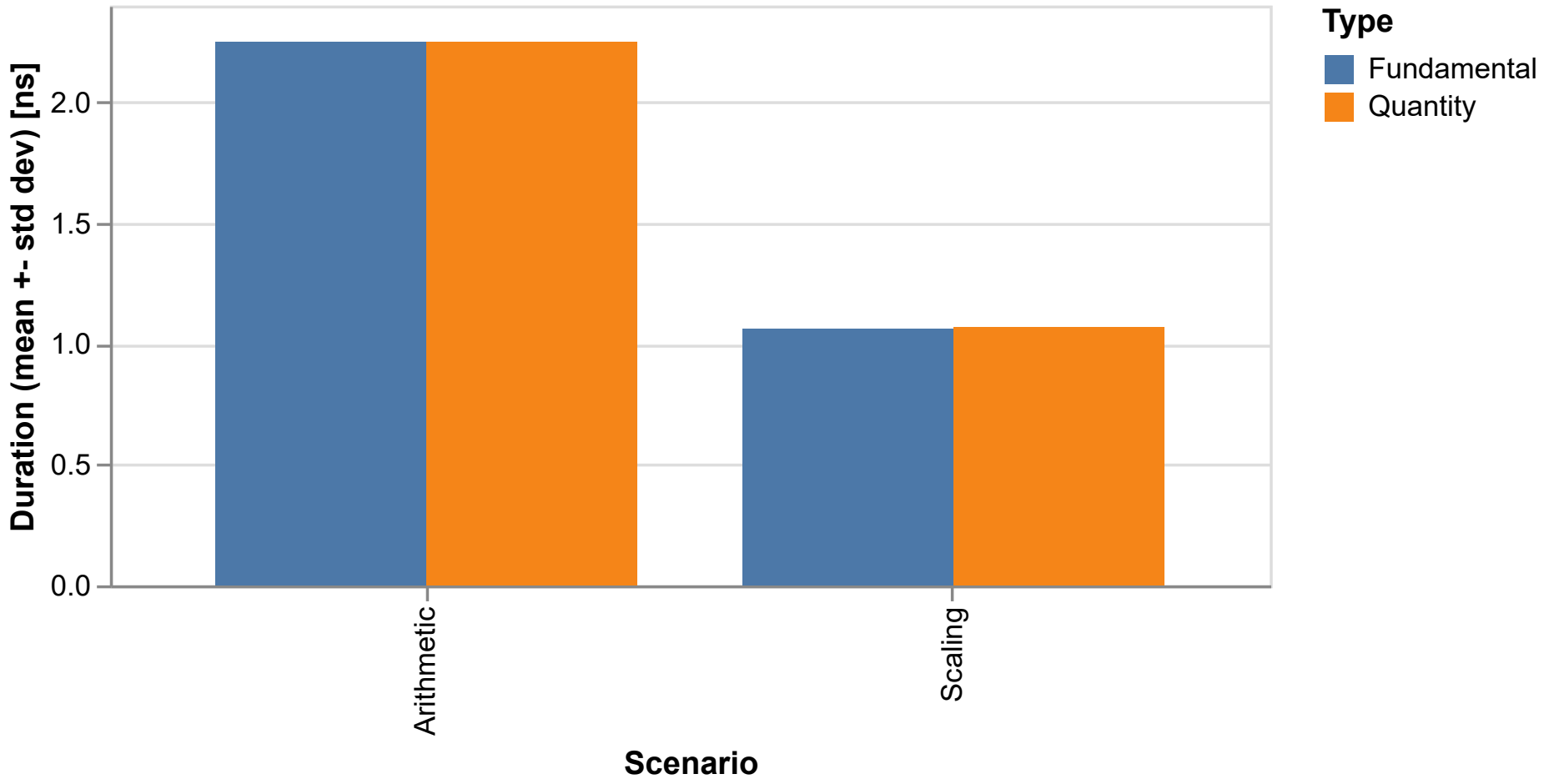
Pint


1,000,000
100,000
10,000
1,000
100
10
1
Duration (mean +- std dev) [ns]
Arithmetic
Scaling
Scenario
Type
Fundamental
Quantity

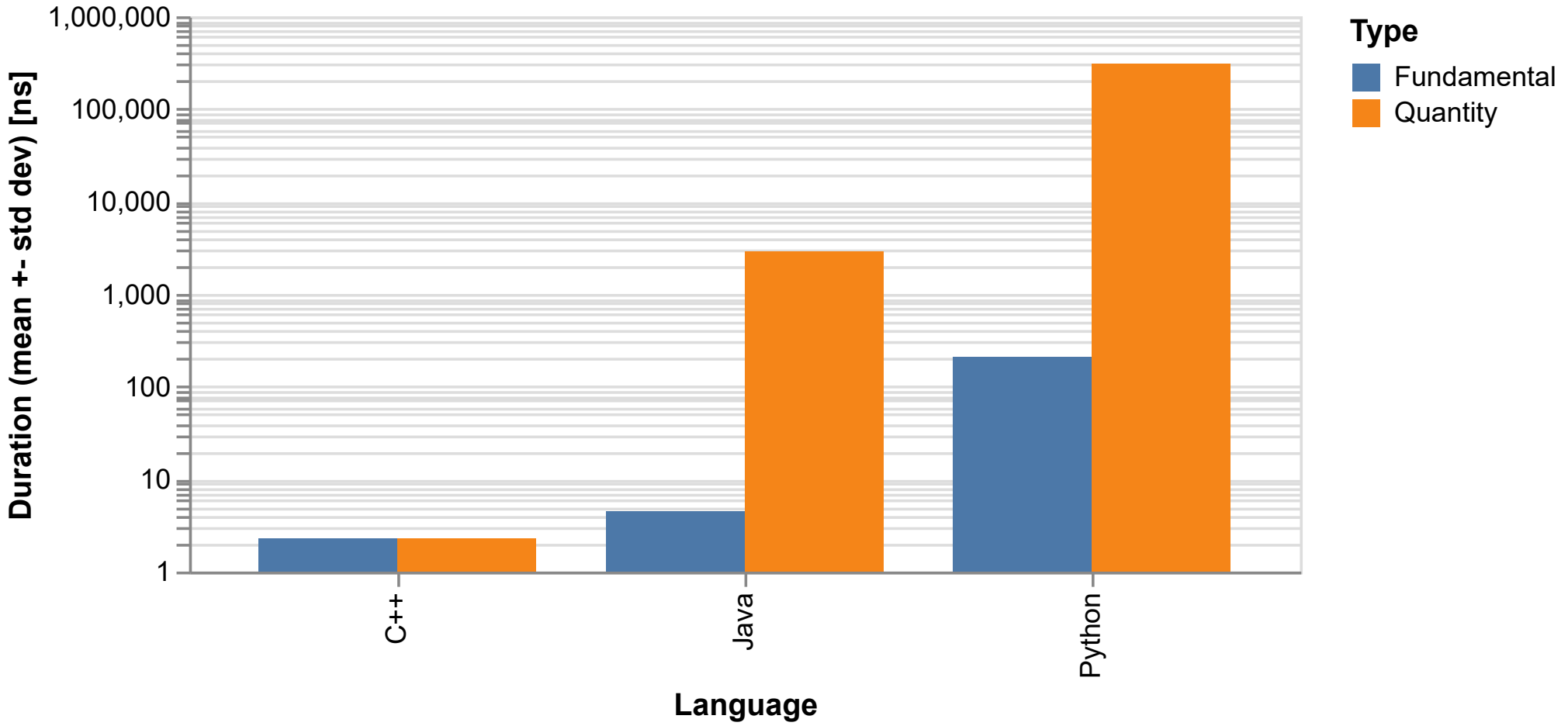
JSR 385


Duration (mean +- std dev) [ns]
10,000
2,000
1,000
200
100
20
10
2
1
Arithmetic
Scaling
Scenario
Type
Fundamental
Quantity

mp-units


Duration (mean +- std dev) [ns]
2.0
1.5
1.0
0.5
0.0
Arithmetic
Scaling
Scenario
Type
Fundamental
Quantity

Comparison (Arithmetic Scenario)



mp-units

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

```
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

mp-units

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

```
constexpr QuantityOf<isq::speed> auto
avg_speed(QuantityOf<isq::length> auto d,
          QuantityOf<isq::time> auto t)
{
  return d / t;
}
```

```
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

```
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

mp-units

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

```
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

```
constexpr QuantityOf<isq::speed> auto
avg_speed(QuantityOf<isq::length> auto d,
          QuantityOf<isq::time> auto t)
{
  return d / t;
}
```

```
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

C++ provides high-level abstractions without sacrificing runtime performance.

Do you agree?

The fastest programs are those that do nothing

Compile-time evaluation when possible

```
constexpr RectangularStorageTank make_tank(const quantity<isq::length[m]>& length,
                                           const quantity<isq::width[m]>& width,
                                           const quantity<isq::height[m]>& height,
                                           const quantity<isq::mass_density[kg / m3]>& density)
{
  auto tank = RectangularStorageTank(length, width, height);
  tank.set_contents_density(density);
  return tank;
}
```

Compile-time evaluation when possible

```
constexpr RectangularStorageTank make_tank(const quantity<isq::length[m]>& length,
                                           const quantity<isq::width[m]>& width,
                                           const quantity<isq::height[m]>& height,
                                           const quantity<isq::mass_density[kg / m3]>& density)
{
  auto tank = RectangularStorageTank(length, width, height);
  tank.set_contents_density(density);
  return tank;
}
```

```
constexpr auto height = isq::height(200 * mm);
constexpr auto tank = make_tank(isq::length(1000 * mm), isq::width(500 * mm),
                                height, 1000 * (kg / m3));
```

Compile-time evaluation when possible

```
constexpr RectangularStorageTank make_tank(const quantity<isq::length[m]>& length,
                                           const quantity<isq::width[m]>& width,
                                           const quantity<isq::height[m]>& height,
                                           const quantity<isq::mass_density[kg / m3]>& density)
{
  auto tank = RectangularStorageTank(length, width, height);
  tank.set_contents_density(density);
  return tank;
}
```

```
constexpr auto height = isq::height(200 * mm);
constexpr auto tank = make_tank(isq::length(1000 * mm), isq::width(500 * mm),
                                height, 1000 * (kg / m3));
```

```
static_assert(tank.fill_level(20. * kg) / height == 20 * percent);
```

Compile-time evaluation when possible

```
constexpr RectangularStorageTank make_tank(const quantity<isq::length[m]>& length,
                                           const quantity<isq::width[m]>& width,
                                           const quantity<isq::height[m]>& height,
                                           const quantity<isq::mass_density[kg / m3]>& density)
{
  auto tank = RectangularStorageTank(length, width, height);
  tank.set_contents_density(density);
  return tank;
}
```

```
constexpr auto height = isq::height(200 * mm);
constexpr auto tank = make_tank(isq::length(1000 * mm), isq::width(500 * mm),
                                height, 1000 * (kg / m3));
```

```
static_assert(tank.fill_level(20. * kg) / height == 20 * percent);
```

Precalculating things at compile-time saves many CPU cycles at runtime.

Compile-time evaluation when possible

```
// simple numeric operations
static_assert(10 * km / 2 == 5 * km);

// unit conversions
static_assert(1 * h == 3600 * s);
static_assert(1 * km + 1 * m == 1001 * m);

// dimension conversions
static_assert(1 * km / (1 * s) == 1000 * (m / s));
static_assert(2 * (km / h) * (2 * h) == 4 * km);
static_assert(2 * km / (2 * (km / h)) == 1 * h);
static_assert(2 * m * (3 * m) == 6 * m2);
static_assert(10 * km / (5 * km) == 2);
static_assert(1000 / (1 * s) == 1 * kHz);
```

Compile-time evaluation when possible

```
// simple numeric operations
static_assert(10 * km / 2 == 5 * km);

// unit conversions
static_assert(1 * h == 3600 * s);
static_assert(1 * km + 1 * m == 1001 * m);

// dimension conversions
static_assert(1 * km / (1 * s) == 1000 * (m / s));
static_assert(2 * (km / h) * (2 * h) == 4 * km);
static_assert(2 * km / (2 * (km / h)) == 1 * h);
static_assert(2 * m * (3 * m) == 6 * m2);
static_assert(10 * km / (5 * km) == 2);
static_assert(1000 / (1 * s) == 1 * kHz);
```

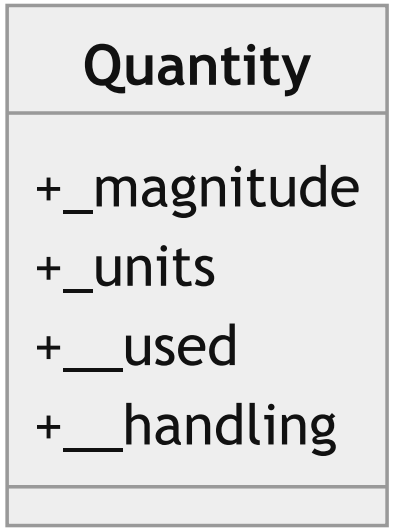
The best unit tests are those that you do not have to run 😉

Efficiency

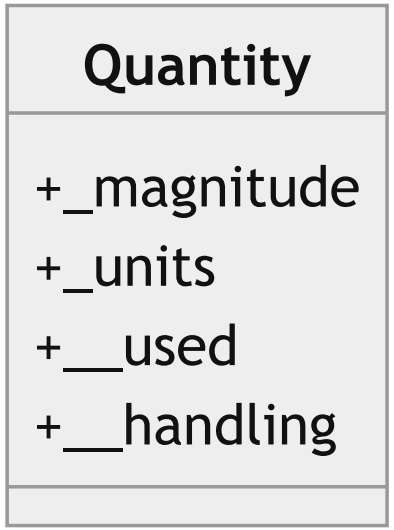
Efficiency is not just running fast or running bigger programs, it's also running using less resources.

-- Bjarne Stroustrup, June 2011

Pint

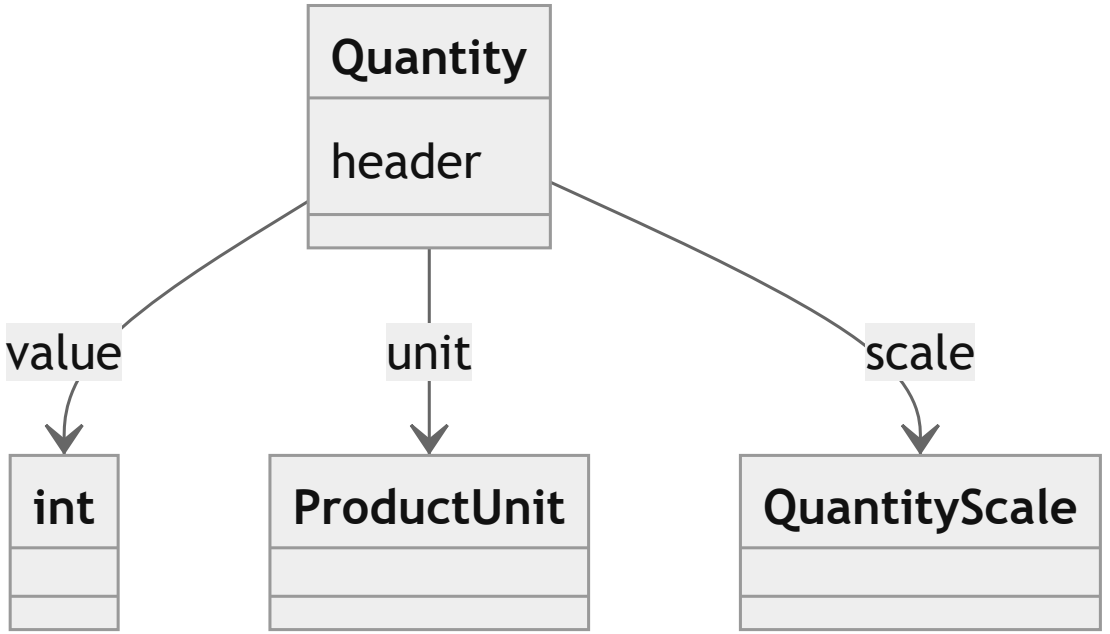


Pint

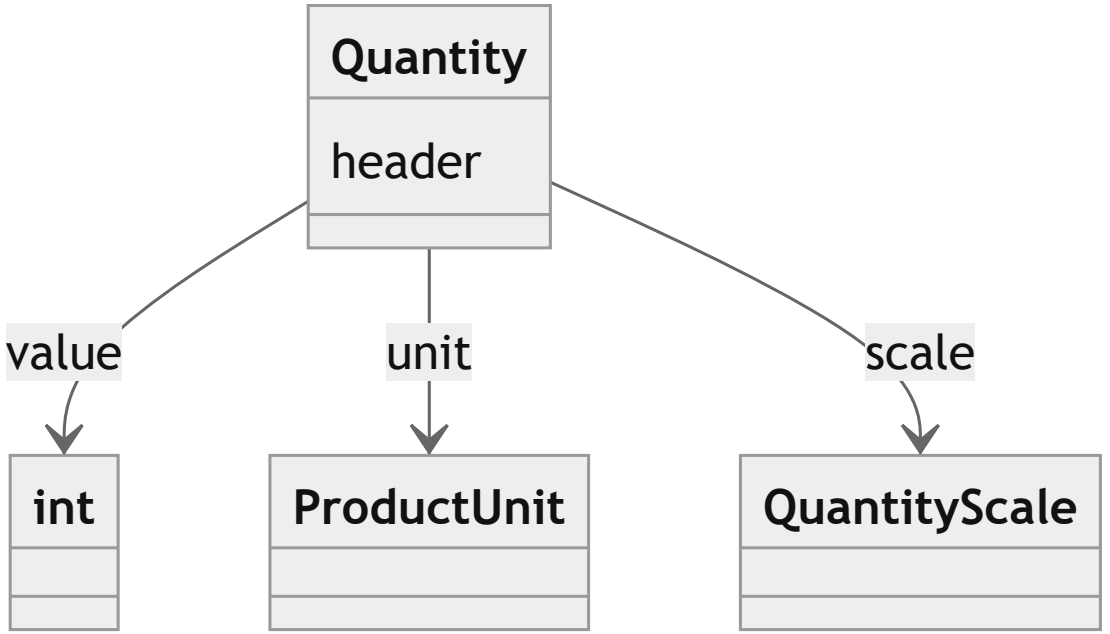


1 Quantity ~ 700 bytes

JSR 385


Quantity
header
value
unit
scale
int
ProductUnit
QuantityScale

JSR 385



1 Quantity ~ 500 bytes

mp-units

quantity<Reference auto R, Representation Rep>
Rep number_

mp-units

quantity<Reference auto R, Representation Rep>
Rep number_

```
static_assert(sizeof(quantity<si::metre>) == sizeof(double));
static_assert(sizeof(quantity<si::metre, int>) == sizeof(int));
static_assert(sizeof(quantity<si::metre, std::int8_t>) == 1);
```

Modern C++

Modern C++ provides safety and high-level abstractions without sacrificing on efficiency.

Modern C++

Modern C++ provides safety and high-level abstractions without sacrificing on efficiency.

If you care about performance or you need a low-level hardware access, then C++ is probably the best programming language for your needs.


CAUTION
Programming
is addictive
(and too much fun)