Übung 03

Arbeitsaufwand insgesamt: 10h

Inhaltsverzeichnis

Inhaltsverzeichnis

Teil 1 – Operatoren überladen

Ziel dieser Übung ist die Implementierung einer Klasse „rational_t“, welche es ermöglicht mit Bruchzahlen (über den ganzen Zahlen) zu rechnen.

Lösungsidee:

Grundsätzlich bedarf es zu diesem Beispiel keine großartige „Lösungsidee“, da ein generisches Problem gelöst werden soll. Zudem sind die genauen Schritte bereits durch die Implementierungshinweise der Angabe angegeben worden.

Da es in der Übung um die Überladung von Operatoren für unsere Klasse „rational_t“ geht, ist es sinnvoll viel zu delegieren, da sich einige Operatoren in deren Funktion überschneiden.
Begonnen habe ich deswegen mit der Implementierung von Funktionen für die 4 Grundrechnungsarten. Die jeweiligen zugehörigen Operatoren können dann einfach an diese Funktionen delegieren.

Beim Vergleich wäre es sinnvoll eine „compare“ Funktion zu schreiben und an diese zu delegieren.

Da eine Funktion „is_consistent“ zu implementieren ist, gilt es zu klären, was „gültig“ bedeutet:
Gültig/Consistent sind für mich alle Brüche, welche keine Null im Nenner aufweisen.

Zudem sollen Brüche immer auf deren kanonischen Repräsentanten konvertiert werden, wo notwendig. Dabei soll, falls möglich, gekürzt und „falsche“ Vorzeichen korrigiert (-3/-3 = 3/3 usw.) werden.
Hilfreich dafür wäre eine Funktion, welche den größten gemeinsamen Teiler zum Kürzen retourniert.

Ausgegeben werden Brüche laut Angabe als „<Zähler/Nenner>“.
Eingelesen werden könnten die Brüche als Zahlenpaare („-13 2“), durch ein Space getrennt.

Bei der Bestimmung von „is_positive“ zähle ich 0 nicht mit, da es eine Eigene Funktion „is_zero“ gibt, welche genau diesen Zweck erfüllt.

Da das Rechnen von Bruchzahlen in Kombination mit Integer möglich sein soll, müssen die Rechenoperator Overrides zusätzlich als non-member Funktionen implementiert werden, da es sich bei „int“ nicht um eine Datenkomponente der Klasse handelt und wir die Definition von „int“ nicht ändern können.

Zu guter Letzt würde ich die Operatoren „<<“ und „>>“ den jeweiligen Stream retournieren lassen, um ein Chaining bei der Ausgabe zu ermöglichen. Zusätzlich delegieren diese Funktionen an „print“ und „scan“.

Da einige Operatoren (Addition, Subtraktion) das Umformen auf den gleichen Nenner fordern, wäre zudem eine Funktion sinnvoll, welche das kleinste gemeinsame Vielfache von den beiden Nennern liefert.

Lösung:

Als C++ Projekt „Teil_1“ im Archiv
Nachfolgender Source-Code

Source-Code:

Main.cpp

```
#include "rational_type.h"
#include "divide_by_zero_exception.h"
#include<iostream>
#include<fstream>

void test_reference()
{
	rational_t r(-1, 2);

	std::cout	<< r * -10 << std::endl
			<< r * rational_t(20, -2)  << std::endl;

	r = 7;

	std::cout	<< r + rational_t(2, 3)				<< std::endl
			<< 10 / r / 2 + rational_t(6, 5)  << std::endl;

	std::cout << std::endl;
}

void test_constructors()
{
	std::cout << "Default constructor: " << rational_t() << std::endl;
	std::cout << "Default constructor: " << rational_t(10) << std::endl;
	std::cout << "Default constructor : " << rational_t(10, 100) << std::endl;

	std::cout << std::endl;
}
```

```
void test_add()
{
	rational_t r1(1, 2);
	rational_t r2(1, 4);

	rational_t r3 = r1 + r2;

	rational_t r4 = r1;
	r4 += r2;
	std::cout << r1 << " + " << r2 << " (normal vs assign): " << r3 << " = " << r4 << 
std::endl;

	r4 += 1;
	r3 = r3 + 1;
	std::cout << "Both plus 1 (normal vs assign): " << r3 << " = " << r4 << std::endl;

	std::cout << std::endl;
}
void test_sub()
{
	rational_t r1(1, 2);
	rational_t r2(1, 4);

	rational_t r3 = r1 - r2;

	rational_t r4 = r1;
	r4 -= r2;
	std::cout << r1 << " - " << r2 << " (normal vs assign): " << r3 << " = " << r4 << 
std::endl;

	r4 -= 1;
	r3 = r3 - 1;
	std::cout << "Both minus 1 (normal vs assign): " << r3 << " = " << r4 << std::endl;

	std::cout << std::endl;
}
```

```
void test_mul()
{
	rational_t r1(1, 2);
	rational_t r2(1, 2);

	rational_t r3 = r1 * r2;

	rational_t r4 = r1;
	r4 *= r2;
	std::cout << r1 << " * " << r2 << " (normal vs assign): " << r3 << " = " << r4 <<
std::endl;

	r4 *= 2;
	r3 = 2 * r3;
	std::cout << "Both multiplied by 2 (normal vs assign): " << r3 << " = " << r4 <<
std::endl;

	std::cout << std::endl;
}
void test_div()
{
	rational_t r1(7, 18);
	rational_t r2(1, 2);

	rational_t r3 = r1 / r2;

	rational_t r4 = r1;
	r4 /= r2;
	std::cout << r1 << " * " << r2 << " (normal vs assign): " << r3 << " = " << r4 <<
std::endl;

	r4 /= 2;
	r3 = r3 / 2;
	std::cout << "Both divided by 2 (normal vs assign): " << r3 << " = " << r4 <<
std::endl;

	std::cout << std::endl;
}
```

```
void test_comparison()
{
	rational_t r1(1, 2);
	rational_t r2(12, 36);

	std::cout << "Comparison of unequal rationals: " << std::endl;
	std::cout << r1.as_string() + " == " + r2.as_string() << ": " << (r1 == r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " != " + r2.as_string() << ": " << (r1 != r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " <  " + r2.as_string() << ": " << (r1 <  r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " <= " + r2.as_string() << ": " << (r1 <= r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " >  " + r2.as_string() << ": " << (r1 >  r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " >= " + r2.as_string() << ": " << (r1 >= r2 ? 
"YES" : "NO") << std::endl;
	std::cout << std::endl;

	r2 = rational_t(1, 2);
	std::cout << "Comparison of equal rationals: " << std::endl;
	std::cout << r1.as_string() + " == " + r2.as_string() << ": " << (r1 == r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " != " + r2.as_string() << ": " << (r1 != r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " <  " + r2.as_string() << ": " << (r1 < r2 ? "YES" : 
"NO") << std::endl;
	std::cout << r1.as_string() + " <= " + r2.as_string() << ": " << (r1 <= r2 ? 
"YES" : "NO") << std::endl;
	std::cout << r1.as_string() + " >  " + r2.as_string() << ": " << (r1 > r2 ? "YES" : 
"NO") << std::endl;
	std::cout << r1.as_string() + " >= " + r2.as_string() << ": " << (r1 >= r2 ? 
"YES" : "NO") << std::endl;

	std::cout << std::endl;
}
```

```
void test_signfunctions()
{
	rational_t r1(-3, -7);
	rational_t r2(-21, 4);
	rational_t r3(0, 33);

	std::cout << "Is" << r1.as_string() << " positive? " << (r1.is_positive()  ? 
"YES" : "NO") << std::endl;
	std::cout << "Is" << r1.as_string() << " negative? " << (r1.is_negative()  ? 
"YES" : "NO") << std::endl;
	std::cout << "Is" << r1.as_string() << " zero?     " << (r1.is_zero()            ? 
"YES" : "NO") << std::endl;
	std::cout << std::endl;
	std::cout << "Is" << r2.as_string() << " positive? " << (r2.is_positive() ? "YES" : 
"NO") << std::endl;
	std::cout << "Is" << r2.as_string() << " negative? " << (r2.is_negative() ? "YES" : 
"NO") << std::endl;
	std::cout << "Is" << r2.as_string() << " zero?     " << (r2.is_zero() ? "YES" : 
"NO") << std::endl;
	std::cout << std::endl;
	std::cout << "Is" << r3.as_string() << " positive? " << (r3.is_positive() ? "YES" : 
"NO") << std::endl;
	std::cout << "Is" << r3.as_string() << " negative? " << (r3.is_negative() ? "YES" : 
"NO") << std::endl;
	std::cout << "Is" << r3.as_string() << " zero?     " << (r3.is_zero() ? "YES" : 
"NO") << std::endl;

	std::cout << std::endl;
}

void test_IO()
{
	rational_t r1;

	std::ifstream in("rational.txt");
	in >> r1;
	std::cout << "From populated file:       " << r1 << std::endl;
	in.close();

	std::cout << std::endl;

	in.open("empty.txt");
	in >> r1;
	std::cout << "From empty file:    " << r1 << std::endl;
	in.close();

	std::cout << std::endl;
}
```

```
void test_exception()
{
	int num(1);
	int den(0);
	rational_t r1;

	std::cout << "Trying to create rational <1/0>: \n";
	try {
		 r1 = rational_t(num, den);
	}
	catch (std::exception& e)
	{
		std::cout << e.what() << std::endl;
		r1 = rational_t(num);
	}

	std::cout << std::endl;

	std::cout << "Created rational: " << r1 << std::endl;
	std::cout << "Trying to divide by <0/1>: \n";
	try {
		r1 /= 0;
	}
	catch (std::exception& e)
	{
		std::cout << e.what() << std::endl;
	}

	std::cout << std::endl;
}

void main()
{
	test_reference();

	//test_add();
	//test_sub();
	//test_mul();
	//test_div();

	//test_comparison();
	//test_signfunctions();

	//test_IO();

	//test_exception();
}
```

rational_type.h

```
#ifndef RATIONAL_TYPE_H
#define RATIONAL_TYPE_H
#include<iostream>


class rational_t {

	// Friend declarations

	friend std::ostream& operator<<(std::ostream& lhs, rational_t const& rhs);
	friend std::istream& operator>>(std::istream& lhs, rational_t& rhs);
	friend rational_t operator+(int const lhs, rational_t& rhs);
	friend rational_t operator-(int const lhs, rational_t& rhs);
	friend rational_t operator*(int const lhs, rational_t& rhs);
	friend rational_t operator/(int const lhs, rational_t& rhs);
	friend rational_t operator+(rational_t& lhs, int const rhs);
	friend rational_t operator-(rational_t& lhs, int const rhs);
	friend rational_t operator*(rational_t& lhs, int const rhs);
	friend rational_t operator/(rational_t& lhs, int const rhs);

private:
	int numerator;
	int denominator;

	// Returns the "least common multiplier" of two numbers
	int lcm(int n1, int n2) const;
	// Returns the "greatest common divider" of two numbers
	int gcd(int n1, int n2) const;
	// Transforms the current rational into it's canonical representation
	void normalize();
	// Returns whether this rational is in a consistent state
	bool is_consistent() const;

	// Compares two rationals.
	// Returns 0 if equal, -1 if main rational is less and +1 if main rational is
greater.
	int cmp(rational_t const& other) const;

	// Adds the value of another rational
	void add(rational_t const& other);
	// Subtracts the value of another rational
	void sub(rational_t const& other);
	// Multiplies by the value of another rational
	void mul(rational_t const& other);
	// Divides by the value of another rational
	void div(rational_t const& other);

	// Prints the current rational to the designated outputstream
	std::ostream& print(std::ostream& out = std::cout) const;
	// Reads in a rational from the given inputstream
	std::istream& scan(std::istream& in);
public:
	// Creates a rational <1/1>
	rational_t();
	// Creates a rational <numerator/1>
	rational_t(int const _numerator);
	// Creates a rational <numerator, denominator>
	rational_t(int const _numerator, int const _denominator);
	// Creates a rational using another rational
	rational_t(rational_t const& src);

	// Assignment operator override.
	rational_t operator=(rational_t const& src);

	// Add operator override.
	rational_t operator+(rational_t const& other);
	// Subtract operator override.
	rational_t operator-(rational_t const& other);
	// Multiply operator override.
	rational_t operator*(rational_t const& other);
	// Divide operator override.
	rational_t operator/(rational_t const& other);

	// Add assignment operator override.
	rational_t operator+=(rational_t const& other);
	// Subtract assignment operator override.
	rational_t operator-=(rational_t const& other);
	// Multiply assignment operator override.
	rational_t operator*=(rational_t const& other);
	// Divide assignment operator override.
	rational_t operator/=(rational_t const& other);

	// Equal operator override.
	bool operator==(rational_t const& other) const;
	// Unequal operator override
	bool operator!=(rational_t const& other) const;
	// Greater than operator override
	bool operator>(rational_t const& other) const;
	// Less than operator override
	bool operator<(rational_t const& other) const;
	// Greater or equal than operator override
	bool operator>=(rational_t const& other) const;
	// Less or equal than operator override
	bool operator<=(rational_t const& other) const;

	// Returns the numerator of the current rational
	int get_numerator() const;
	// Returns the denominator of the current rational
	int get_denominator() const;

	// Returns the current rational as string as <numerator/denominator>
	std::string as_string() const;

	// Returns whether the rational is positive or not
	bool is_positive();
	// Returns whether the rational is negative or not
	bool is_negative();
	// Returns whether the rational is zero or not
	bool is_zero();
};
```

```
// Non member operators

// Output operator overide
std::ostream& operator<<(std::ostream& lhs, rational_t const& rhs);
// Input operator override
std::istream& operator>>(std::istream& lhs, rational_t& rhs);

// Add operator override. Combination with int.
rational_t operator+(int const lhs, rational_t& rhs);
// Subtract operator override. Combination with int.
rational_t operator-(int const lhs, rational_t& rhs);
// Multiply operator override. Combination with int.
rational_t operator*(int const lhs, rational_t& rhs);
// Divide operator override. Combination with int.
rational_t operator/(int const lhs, rational_t& rhs);
// Add operator override. Combination with int.
rational_t operator+(rational_t& lhs, int const rhs);
// Subtract operator override. Combination with int.
rational_t operator-(rational_t& lhs, int const rhs);
// Multiply operator override. Combination with int.
rational_t operator*(rational_t& lhs, int const rhs);
// Divide operator override. Combination with int.
rational_t operator/(rational_t& lhs, int const rhs);

#endif
```

rational_type.cpp

```
#include "rational_type.h"
#include "divide_by_zero_exception.h"
#include<string>

// private
int rational_t::lcm(int n1, int n2) const
{
	return (n1 * n2) / gcd(n1, n2);
}
int rational_t::gcd(int n1, int n2) const
{
	n1 = std::abs(n1);
	n2 = std::abs(n2);

	if (n1 == 0 || n2 == 0)
	{
		return 0;
	}

	// Subtract the lesser number from the greater number until equal (= either 0 or 
common divider)
	while (n1 != n2)
	{
		if (n1 > n2)
		{
			n1 -= n2;
		}
		else
		{
			n2 -= n1;
		}
	}

	return n1;
}
void rational_t::normalize()
{
	// Calculate common divider
	int gcd_value(gcd(numerator, denominator));

	// If there is a common divider between numerator and denominator, divide by it.
	if (gcd_value != 0)
	{
		numerator /= gcd_value;
		denominator /= gcd_value;
	}

	// Also, if the denominator is negative, flip the signs.
	if (denominator < 0)
	{
		numerator *= -1;
		denominator *= -1;
	}
}
```

```
bool rational_t::is_consistent() const
{
	return this->denominator!=0;
}

int rational_t::cmp(rational_t const& other) const
{
	// Calculate lowest common multiple and calculate "normalized" numerators (if they
had the same denominators)
	int lcm_value(lcm(this->denominator, other.denominator));
	int num1(lcm_value / this->denominator * this->numerator);
	int num2(lcm_value / other.denominator * other.numerator);

	// Then compare and return the result
	if (num1 > num2)
	{
		return 1;
	}
	else if (num1 < num2)
	{
		return -1;
	}
	else
	{
		return 0;
	}
}

void rational_t::add(rational_t const& other)
{
	// Calc lowest common multiple between the two values
	int new_denominator(lcm(this->denominator, other.denominator));

	// Then convert the value to have the final denominator
	this->numerator *= new_denominator / this->denominator;
	this->denominator = new_denominator;

	// Then add the multiplied numerator of the other half.
	numerator += other.numerator * (new_denominator / other.denominator);

	// At the end, normalize it to get canonical form.
	normalize();
}
void rational_t::sub(rational_t const& other)
{
	// Just like in add(), first convert both numbers to have the same denominator,
then substract them.
	int new_denominator(lcm(this->denominator, other.denominator));
	this->numerator *= new_denominator / this->denominator;
	this->denominator = new_denominator;

	numerator -= other.numerator * (new_denominator / other.denominator);
	normalize();
}
```

```
void rational_t::mul(rational_t const& other)
{
	// When multiplying to rationals, no previous conversion is needed.
	numerator *= other.numerator;
	denominator *= other.denominator;

	normalize();
}
void rational_t::div(rational_t const& other)
{
	// If denominator is 0, abort and throw an exception.
	if (!other.is_consistent())
	{
		throw divide_by_zero_exception();
	}

	rational_t tmp(other.denominator, other.numerator);
	mul(tmp);

	normalize();
}
```

```
// public
rational_t::rational_t()
{
	numerator = 1;
	denominator = 1;
}
rational_t::rational_t(int const _numerator)
{
	numerator = _numerator;
	denominator = 1;
}
rational_t::rational_t(int const _numerator, int const _denominator)
{
	numerator = _numerator;
	denominator = _denominator;

	// If not consistent (0 in denominator), display and throw an error/Exception.
	if (!is_consistent())
	{
		throw divide_by_zero_exception();
	}

	normalize();
}
rational_t::rational_t(rational_t const& src)
{
	numerator = src.numerator;
	denominator = src.denominator;
}

bool rational_t::is_positive()
{
	return numerator > 0;
}
bool rational_t::is_negative()
{
	return numerator < 0;
}
bool rational_t::is_zero()
{
	return numerator == 0;
}


int rational_t::get_numerator() const
{
	return numerator;
}
int rational_t::get_denominator() const
{
	return denominator;
}
```

```
std::string rational_t::as_string() const
{
	return "<" + (	numerator%denominator==0 ?
					std::to_string(numerator/denominator) :
					std::to_string(numerator) + "/" + 
std::to_string(denominator) ) + ">";
}
std::ostream& rational_t::print(std::ostream& out) const
{
	out << as_string();
	return out;
}
std::istream& rational_t::scan(std::istream& in)
{
	// read in two values from the designated stream
	// If reading in fails, set it to 1 or 0 respectively
	if (!(in >> this->numerator))
		this->numerator = 0;
	if (!(in >> this->denominator))
		this->denominator = 1;

	return in;
}

rational_t rational_t::operator=(rational_t const& src)
{
	// On assignment, check if the object is the same.
	if (*this == src)
	{
		return *this;
	}
	else
	{
		// If it is a different object assign the values directly instead of 
claiming new memory.
		this->numerator = src.numerator;
		this->denominator = src.denominator;
		return *this;
	}
}
```

```
// Delegate to add/sub/mul/div
rational_t rational_t::operator+(rational_t const& other)
{
	rational_t tmp(*this);
	tmp.add(other);
	return tmp;
}
rational_t rational_t::operator-(rational_t const& other)
{
	rational_t tmp(*this);
	tmp.sub(other);
	return tmp;
}
rational_t rational_t::operator*(rational_t const& other)
{
	rational_t tmp(*this);
	tmp.mul(other);
	return tmp;
}
rational_t rational_t::operator/(rational_t const& other)
{
	rational_t tmp(*this);
	tmp.div(other);
	return tmp;
}

rational_t rational_t::operator+=(rational_t const& other)
{
	add(other);
	return *this;
}
rational_t rational_t::operator-=(rational_t const& other)
{
	sub(other);
	return *this;
}
rational_t rational_t::operator*=(rational_t const& other)
{
	mul(other);
	return *this;
}
rational_t rational_t::operator/=(rational_t const& other)
{
	div(other);
	return *this;
}
```

```cpp
// Delegate to cmp() function
bool rational_t::operator==(rational_t const& other) const
{
	return cmp(other) == 0;
}
bool rational_t::operator!=(rational_t const& other) const
{
	return cmp(other) != 0;
}
bool rational_t::operator>(rational_t const& other) const
{
	return cmp(other) == 1;
}
bool rational_t::operator<(rational_t const& other) const
{
	return cmp(other) == -1;
}
bool rational_t::operator>=(rational_t const& other) const
{
	return cmp(other) != -1;
}
bool rational_t::operator<=(rational_t const& other) const
{
	return cmp(other) != 1;
}
```

```
// Non member functions
std::ostream& operator<<(std::ostream& lhs, rational_t const& rhs)
{
	// Delegate to print
	rhs.print(lhs);
	return lhs;
}
std::istream& operator>>(std::istream& lhs, rational_t& rhs)
{
	// Delegate to scan
	rhs.scan(lhs);
	return lhs;
}

rational_t operator+(int const lhs, rational_t& rhs)
{
	rational_t tmp(lhs);
	return tmp + rhs;
}
rational_t operator-(int const lhs, rational_t& rhs)
{
	rational_t tmp(lhs);
	return tmp - rhs;
}
rational_t operator*(int const lhs, rational_t& rhs)
{
	rational_t tmp(lhs);
	return tmp * rhs;
}
rational_t operator/(int const lhs, rational_t& rhs)
{
	rational_t tmp(lhs);
	return tmp / rhs;
}

rational_t operator+(rational_t& lhs, int const rhs)
{
	return operator+(rhs, lhs);
}
rational_t operator-(rational_t& lhs, int const rhs)
{
	rational_t tmp(rhs);
	return lhs - tmp;
}
rational_t operator*(rational_t& lhs, int const rhs)
{
	return operator*(rhs, lhs);
}
rational_t operator/(rational_t& lhs, int const rhs)
{
	rational_t tmp(rhs);
	return lhs / tmp;
}
```

divide_by_zero_exception.h

```
#ifndef DIVIDE_BY_ZERO_EXCEPTION_H
#define DIVIDE_BY_ZERO_EXCEPTION_H
#include<exception>

class divide_by_zero_exception : public std::exception
{
public:
	virtual const char* what() const throw()
	{
		return "Divide by 0 exception";
	}
};

#endif
```

Testfälle

- Referenzbeispiel
- Grundrechenarten
- Vergleich
- Vorzeichenbasierte Funktionen
- Input/Output
- Exception-Handling

Alle Testfälle sind zusätzlich als Funktionen in main.cpp enthalten.

Referenzbeispiel:

```
<5>
<5>
<23/3>
<67/35>
```

Grundrechenarten:

```
<1/2> + <1/4> (normal vs assign): <3/4> = <3/4>
Both plus 1 (normal vs assign): <7/4> = <7/4>

<1/2> - <1/4> (normal vs assign): <1/4> = <1/4>
Both minus 1 (normal vs assign): <-3/4> = <-3/4>

<1/2> * <1/2> (normal vs assign): <1/4> = <1/4>
Both multiplied by 2 (normal vs assign): <1/2> = <1/2>

<7/18> * <1/2> (normal vs assign): <7/9> = <7/9>
Both divided by 2 (normal vs assign): <7/18> = <7/18>
```

Vergleich:

```
Comparison of unequal rationals:
<1/2> == <1/3>: NO
<1/2> != <1/3>: YES
<1/2> <  <1/3>: NO
<1/2> <= <1/3>: NO
<1/2> >  <1/3>: YES
<1/2> >= <1/3>: YES

Comparison of equal rationals:
<1/2> == <1/2>: YES
<1/2> != <1/2>: NO
<1/2> <  <1/2>: NO
<1/2> <= <1/2>: YES
<1/2> >  <1/2>: NO
<1/2> >= <1/2>: YES
```

Vorzeichenbasierte Funktionen:

```
Is<3/7> positive? YES
Is<3/7> negative? NO
Is<3/7> zero?     NO

Is<-21/4> positive? NO
Is<-21/4> negative? YES
Is<-21/4> zero?     NO

Is<0> positive? NO
Is<0> negative? NO
Is<0> zero?     YES
```

Input/Output:

```
From populated file:    <-23/3>

From empty file:        <0>
```

Exception-Handling:

```
Trying to create rational <1/0>:
Divide by 0 exception
Created rational: <1>

Trying to divide by <0/1>:
Divide by 0 exception
```