Value semantics and reference semantics in C++ and other languages

Adam Graliński

C++ Friends, July 2022

Value semantics and reference semantics

Value semantics	**Reference semantics**
Refers to concrete objects (one specific location in memory)	Refers to any type-like object held anywhere in memory
Comparison based on values	Comparison based on memory addresses
Deep copying, deep assignment	Shallow copying, shallow assignment

C++ uses *value semantics* for most things* by default.

Semantics for primitive types: C++

01_primitives.cpp

```cpp
#include <iostream>

int main()
{
  int a{10};
  int b{a};
  a = 20;
  std::cout << "b == " << b << "\n";
  std::cout << (b == 10 ? "value" : "reference")
      << " semantics for primitives\n";
}
```

Semantics for primitive types: C++

01_primitives.cpp

```cpp
#include <iostream>

int main()
{
  int a{10};
  int b{a};
  a = 20;
  std::cout << "b == " << b << "\n";
  std::cout << (b == 10 ? "value" : "reference")
      << " semantics for primitives\n";
}
```

output

```
b == 10
value semantics
for primitives
```

https://godbolt.org/z/zrWMcjbYG

Semantics for primitive types: Python

01_primitives.py

```
a = 10
b = a
a = 20
print(f"b == {b}")
semantic_method = "value" if b == 10 else "reference"
print(f"{semantic_method} semantics for primitives")
```

Semantics for primitive types: Python

01_primitives.py

```
a = 10
b = a
a = 20
print(f"b == {b}")
semantic_method = "value" if b == 10 else "reference"
print(f"{semantic_method} semantics for primitives")
```

output

```
b == 10
value semantics
for primitives
```

Semantics for primitive types: summary

- both **C++** and **Python** use *value semantics* for primitives
- same for **Java**, **Lua**, and virtually all other languages
- that was the easiest part...

Semantics for classes: C++

02_classes.cpp

```cpp
#include <iostream>

struct Fraction {
  int numerator, denominator;
  auto operator<=>(Fraction const &) const = default;
};

int main() {
  Fraction a {3, 5};
  Fraction b {a};
  a.numerator = 0;
  std::cout << (a != b ? "value" : "reference")
    << " semantics for classes\n";
}
```

Semantics for classes: C++

02_classes.cpp

```cpp
#include <iostream>

struct Fraction {
  int numerator, denominator;
  auto operator<=>(Fraction const &) const = default;
};

int main() {
  Fraction a {3, 5};
  Fraction b {a};
  a.numerator = 0;
  std::cout << (a != b ? "value" : "reference")
    << " semantics for classes\n";
}
```

output

```
value semantics
for classes
```

https://godbolt.org/z/d78qYrxvY

Semantics for classes: Python

02_classes.py

```
class Fraction:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

a = Fraction(3, 5)
b = a
a.numerator = 0

method = "value" if a != b else "reference"
print(f"{method} semantics for classes")
```

Semantics for classes: Python

02_classes.py

```
class Fraction:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

a = Fraction(3, 5)
b = a
a.numerator = 0

method = "value" if a != b else "reference"
print(f"{method} semantics for classes")
```

output

```
reference semantics
for classes
```

Semantics for classes: summary

- `C++` offers *value semantics* with objects by default
- `Python` uses *reference semantics* for objects
- performing a *deep copy* is necessary to actually copy an object
- same with `Java` (`clone()`), `Lua` (roll your own), `JavaScript`, ...

Semantics for containers: C++

03_containers.cpp

```
#include <iostream>
#include <vector>

int main() {
  std::vector a{0, 1, 2, 3};
  std::vector b{a};
  a[0] = 101;
  std::cout << (a != b ? "value" : "reference")
    << " semantics for containers\n";
}
```

Semantics for containers: C++

03_containers.cpp

```
#include <iostream>
#include <vector>

int main() {
  std::vector a{0, 1, 2, 3};
  std::vector b{a};
  a[0] = 101;
  std::cout << (a != b ? "value" : "reference")
    << " semantics for containers\n";
}
```

output

```
value semantics
for containers
```

https://godbolt.org/z/jh4a1WoYP

Semantics for lists: Python

03_lists.py

```
a = [0, 1, 2, 3]
b = a
a[0] = 101

method = "value" if a != b else "reference"
print(f"{method} semantics for lists")
```

Semantics for lists: Python

03_lists.py

```
a = [0, 1, 2, 3]
b = a
a[0] = 101

method = "value" if a != b else "reference"
print(f"{method} semantics for lists")
```

output

```
reference semantics
for lists
```

Semantics for passing objects to functions: C++

04_passing.cpp

```
void change_fraction(Fraction f) {
  f.numerator = 1;
  f.denominator = 2;
  cout << "Inside change_fraction, f == " << f << "\n";
}
void make_new_fraction(Fraction& f) {
  f = {8, 13}; // calls Fraction constructor
  cout << "Inside make_new_fraction, f == " << f << "\n";
}
int main() {
  Fraction a {3, 5};
  cout << "Before change_fraction, a == " << a << "\n";
  change_fraction(a);
  cout << "After change_fraction, a == " << a << "\n\n";

  cout << "Before make_new_fraction, a == " << a << "\n";
  make_new_fraction(a);
  cout << "After make_new_fraction, a == " << a << "\n";
}
```

output

```
a == (3/5)
f == (1/2)
a == (3/5)

a == (3/5)
f == (8/13)
a == (8/13)
```

no surprises

https://godbolt.org/z/44j55dboo

Semantics for passing objects to functions: Java

04_passing.java

```
class Fraction {
  public int numerator, denominator;
  public Fraction(int n, int d) {
    numerator = n;
    denominator = d;
  }
}

class Passing {
  static void change_fraction(Fraction f) {
    f.numerator = 1;
    f.denominator = 2;
    System.out.printf("Inside change_fraction, f == (%d/%d)\n", f.numerator, f.denominator);
  }

  static void make_new_fraction(Fraction f) {
    f = new Fraction(8, 13);
    System.out.printf("Inside make_new_fraction, f == (%d/%d)\n", f.numerator, f.denominator);
  }

  public static void main(String[] args) {
    Fraction a = new Fraction(3, 5);
    System.out.printf("Before change_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    change_fraction(a);
    System.out.printf("After change_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    System.out.println();
    System.out.printf("Before make_new_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    make_new_fraction(a);
    System.out.printf("After make_new_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
  }
}
```

Semantics for passing objects to functions: Java

04_passing.java

```
class Fraction {
  public int numerator, denominator;
  public Fraction(int n, int d) {
    numerator = n;
    denominator = d;
  }
}

class Passing {
  static void change_fraction(Fraction f) {
    f.numerator = 1;
    f.denominator = 2;
    System.out.printf("Inside change_fraction, f == (%d/%d)\n", f.numerator, f.denominator);
  }

  static void make_new_fraction(Fraction f) {
    f = new Fraction(8, 13);
    System.out.printf("Inside make_new_fraction, f == (%d/%d)\n", f.numerator, f.denominator);
  }

  public static void main(String[] args) {
    Fraction a = new Fraction(3, 5);
    System.out.printf("Before change_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    change_fraction(a);
    System.out.printf("After change_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    System.out.println();
    System.out.printf("Before make_new_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
    make_new_fraction(a);
    System.out.printf("After make_new_fraction, a == (%d/%d)\n", a.numerator, a.denominator);
  }
}
```

output

```
a == (3/5)
f == (1/2)
a == (1/2) // OK

a == (1/2)
f == (8/13)
a == (1/2) // <--
```

Key takeaways

- Know what kind of semantics are in effect in your language
- **C++**: value semantics by default,
  reference semantics on request (with *references&*)
- **Java**: reference semantics by default,
  but references are passed *by value* to functions
- one more reason to favor **array** and **string** over C-style arrays and C-strings

Thank you!