

Value semantics and reference semantics

Value semantics	Reference semantics
Refers to concrete objects	Refers to any type-like object
(one specific location in memory)	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.

See a lot more here: https://hackingcpp.com/cs/value_vs_reference_semantics.html

Semantics for primitive types: C++

O1_primitives.cpp

Semantics for primitive types: C++

01_primitives.cpp

output

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

https://godbolt.org/z/zrWMcjbYG

Semantics for primitive types: Python

O1_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

O1_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

Semantics for classes: C++

02_classes.cpp

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

Semantics for containers: C++

03_containers.cpp

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";</pre>
int main() {
 Fraction a {3, 5}:
  cout << "Before change_fraction, a == " << a << "\n";</pre>
  change fraction(a):
  cout << "After change fraction, a == " << a << "\n\n";</pre>
  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 == (\frac{1}{\sqrt{d}})n", a, numerator, a denominator):
    change fraction(a):
   System.out.printf("After change fraction, a == (\frac{1}{2}d/\frac{1}{2}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 == (\frac{1}{\sqrt{d}})n", a, numerator, a denominator):
    change fraction(a):
   System.out.printf("After change fraction, a == (\frac{1}{2}d/\frac{1}{2}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!