Programming Fundamentals 2

Pierre Talbot

23 March 2021

University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

Chapter VI. Casting Polymorphism

Casting of primitive types

Casting polymorphism

```
double price = 9.99;
int rounded_price = (int) price;
// rounded_price = ?
```

Casting

Casting is an operation allowing to convert a value from a type to a value of another type. For instance, to view `price` as an `int` instead of a `double`.

Recall from Chapter 2...

A type is a size $s \in \mathbb{N}$ in bits and a pair of imaginary functions $f : \{0,1\}^s \rightarrow T$ and $g : T \rightarrow \{0,1\}^s$, such that T is the values you manipulate in the program.

Examples

- For `int`: size $= 32$ bits, $f_{int}(0^{24}01000001) = 64$,
- For `float`: size $= 32$ bits, $f_{float}(0^{24}01000001) = 9.108\ldots^{-44}$,
- For `char`: size $= 16$ bits, $f_{char}(0^{8}01000001) = A$,
- For `boolean`: size $= 1$ bit, $f_{boolean}(1) = true$.

Bit-level Casting

We could just reinterpret the memory with the new type by changing the function f:

- Let `int x = 64;` and `float y = (float) x;`.
- We could view this operation as: $\texttt{(float)x} = f_{float}(g_{int}(x)) = f_{float}(0^{24}01000001) = 9.108\ldots^{-44} = \texttt{y}$.

However, we would normally expect the casting operation to give $\texttt{y} = 64.0$ as a result.

Type-level Casting

- To reach the expected result, we introduce a casting function $\mathit{cast} : \mathit{int} \rightarrow \mathit{float}$.
- This function does not reinterpret the bits, but work at the level of the type T.
- Therefore, we have $\mathit{cast}(64) = 64.0$.
- There are cast functions for each conversion ($\mathit{float} \rightarrow \mathit{int}$, $\mathit{char} \rightarrow \mathit{int}, \ldots$).

Cast operations are partial functions

Some casting functions are partial functions (in theory):

- $cast : float \rightarrow int$: 4.5 can't be converted to integer.
- $cast : int \rightarrow short$: 100000 can't be converted to a short (too large).
- ...

In practice, they are some rules that make these functions total:

- $cast : float \rightarrow int$: round towards 0, *e.g.*:
  - $cast(4.5) = 4$
  - $cast(-4.5) = -4$
  - $cast(NaN) = 0$
- $cast : int \rightarrow short$: truncate the extra bits, and simply use f_{short} on the remaining bits:
  1. $g_{int}(100000) = 00000000\ 00000001\ 10000110\ 10100000$,
  2. $f_{short}(10000110\ 10100000) = -31072$

Implicit casting

To improve readability, many languages provide some automatic and implicit type conversions.

- Generally implicit when no precision is lost, *e.g.*, `short x = 10; int y = x`.
- Sometimes implicit although precision might be lost, *e.g.*, `int` to `float`.

Some languages such as Rust, forbids implicit casts, and favor explicit casts instead.

Casting of object types

Casting of object types

Following inheritance relationships, we can cast an object to a superclass or subclass.

- *Upcast* (implicit): Cast an object of type T to an object of type U such that $T \leq U$.

```
Weapon w = new Axe(); // The type Axe is upcasted to the type Weapon.
```

- *Downcast*: Cast an object of type T to an object of type U such that $T > U$.

```
Axe a = (Axe) w; // The type Weapon is downcasted from the type Weapon to the type Axe.
```

Downcast

Imagine the following code:

```
Weapon w = new Axe();
// ...
Hammer h = (Hammer) w; // oops!
```

- By downcasting, we cannot be sure that the runtime type of *w* is actually a type `Hammer`, in contrast to upcasting where the relationship can be verified at compile-time.
- In the previous example a `ClassCastException` is thrown.

Instanceof and getclass

When downcasting, you must always verify that the object you downcast is of the expected type. Suppose T is the runtime type of x:

- `x instanceof U` evaluates to *true* if $T \leq U$.
- `x.getClass() == U.class` evaluates to *true* if $T = U$.

Example (Instanceof vs getclass)

```
class MithrilAxe extends Axe { ... }
//...
Weapon w = new MithrilAxe();
if(w instanceof Axe) { System.out.println("w is an axe or a subtype of Axe.\n"); }
else if(w instanceof Hammer) { System.out.println("w is a hammer or a subtype of Hammer.\
// ...
if(w.getClass() == Axe.class) { System.out.println("w is an Axe."); }
else if(w.getClass() == MithrilAxe.class) { System.out.println("w is a MithrilAxe."); }
```

Is downcast a bad practice?

- Downcast is not necessarily a bad practice, however it leads to a more imperative programming style, and might indicate some issues with your object-oriented design.
- Nevertheless, downcast is always required for very specific cases such as overriding the method `equals`, see Chapter 7.

The expression problem

This simple discussion on downcast actually leads to a fundamental problem called *the expression problem*[1].

Extending data or operation?

- Casting polymorphism makes it easy to add new algorithms on existing data, without modifying existing code.
- Subtype polymorphism makes it easy to add new data classes without modifying existing algorithms.

It is best explained through an example: see *Live Coding Session: Coding a calculator!*
We will see in Chapter 10 the *visitor design pattern*, an object-oriented pattern that partially solves this problem.

[1] https://en.wikipedia.org/wiki/Expression_problem

What to remember about casting polymorphism?

- We can transform a value to view it under various forms.
- This form of polymorphism is probably the most widespread across languages (C, C++, Python, Javascript, ...).
- You must be careful to the specificities of each language. For instance in C++, there are 4 different casting operators (`static_cast` (type-level casting), `reinterpret_cast` (bit-level casting), ...).
- *Expression problem*: Tensions between data extension and algorithmic extension, and casting polymorphism vs subtype polymorphism.