CS1010X: Programming Methodology I

Chapter 11: Multiple Representations and Data-Directed Programming

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 11: Multiple Representations and Data-Directed Programming

Learning Objectives

By the end of this chapter, students should be able to:

- **Explain** why multiple representations co-exist in real software projects and **compare** strategies to manage them (tagging, dispatch, tables, message passing).
- **Implement** tagged data and **dispatch on type** using generic operators.
- **Implement data-directed programming** using a method table (`put / get + apply_generic`) and **install** packages safely.
- **Describe layering** and explain why systems sometimes carry "tags upon tags".
- **Explain coercion** and relate it to a simple **number hierarchy**.
- **Explain message passing** as an alternative dispatch strategy and relate it to OOP.

11.1 Motivation: When One Representation is Not Enough

In Chapter 10, we used **data abstraction** to hide the representation of complex numbers behind a stable interface (constructors + selectors). This works well when a program commits to exactly *one* representation at a time.

Real systems are rarely so tidy. Multiple representations may co-exist because:

- different operations benefit from different representations (e.g. add vs. multiply),
- teams build components independently and choose different internal formats,
- performance requirements change over time (e.g. caching magnitude, precision modes),
- backwards compatibility requires supporting old and new formats simultaneously.

Suppose we have two complex-number packages:

- **Rectangular package** expects raw data shaped like `(x, y)`.
- **Polar package** expects raw data shaped like `(r, a)`.

Then two issues appear:

(1) Representation mismatch. If we pass `(r, a)` into a function that expects `(x, y)`, the program still runs, but computes the wrong thing.

(2) Name conflicts. Both packages naturally want to define names like `real_part` and `angle`. If we import them into the same namespace, one overwrites the other.

We want a design where:

- clients call `real_part(z)` / `add(z1,z2)` without knowing representation,
- adding a new representation requires *local* changes (not a rewrite),
- names from different packages do not collide.

11.2 Solutions to Handling Multiple Representations

There are three main methods of handling multiple representations, which we will see below:

1. Dispatch on Type
2. Data-Directed Programming
3. Message Passing

In particular, we would like to add generic operators as a new abstraction barrier to allow us to handle multiple representations more easily.

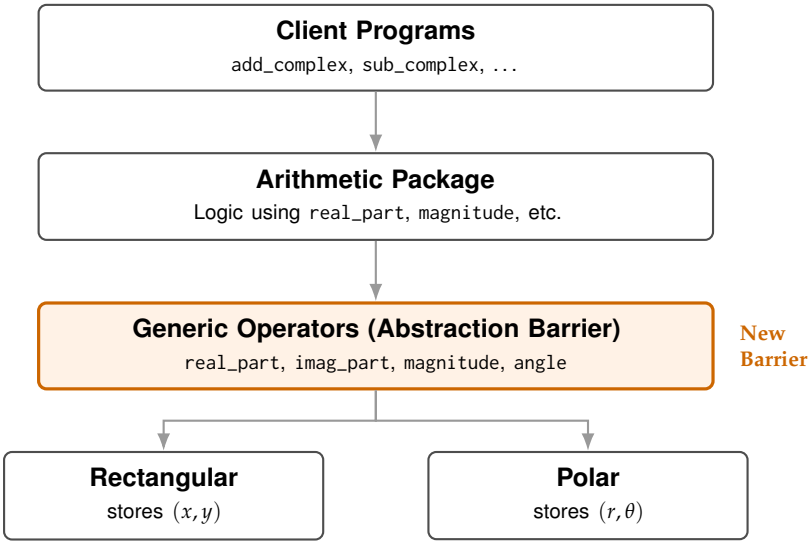


Figure 11.1: The generic operators act as a new abstraction barrier, shielding arithmetic logic from data representation.

11.3 Tagged Data and Dispatch on Type

The simplest approach is to attach a tag to each datum that records its representation. For complex numbers, we store:

`('rectangular', (x,y))` or `('polar', (r,a))`.

We can define some helper functions for tagging data.

```
def attach_tag(type_tag, contents):
    """Return a tagged datum."""
    return (type_tag, contents)

def type_tag(datum):
    """Extract the tag from a tagged datum."""
    if type(datum) == tuple and len(datum) == 2:
        return datum[0]
    raise Exception("Bad tagged datum (type_tag): " + str(datum))

def contents(datum):
    """Extract the payload from a tagged datum."""
    if type(datum) == tuple and len(datum) == 2:
        return datum[1]
    raise Exception("Bad tagged datum (contents): " + str(datum))
```

Invariant for tagged data

A tagged datum must always have the shape `(tag, payload)`. Dispatch relies on this invariant, so do not let client code treat tagged values as ordinary tuples.

We can then rewrite our constructors to make use of these tags:

```
def make_rectangular(x, y):
    return attach_tag('rectangular', (x, y))

def make_polar(r, a):
    return attach_tag('polar', (r, a))
```

A **generic operator** can inspect the tag and route it to the correct implementation.

```
import math

# representation-specific selectors on raw data
def real_part_rect(z): return z[0]
def imag_part_rect(z): return z[1]

def real_part_polar(z):
    r, a = z
    return r * math.cos(a)

def imag_part_polar(z):
    r, a = z
    return r * math.sin(a)

def real_part(z):
    """Generic selector: works on tagged data."""
    if type_tag(z) == 'rectangular':
        return real_part_rect(contents(z))
    elif type_tag(z) == 'polar':
        return real_part_polar(contents(z))
    else:
        raise Exception("Unknown type -- real_part: " + str(type_tag(z
            )))
```

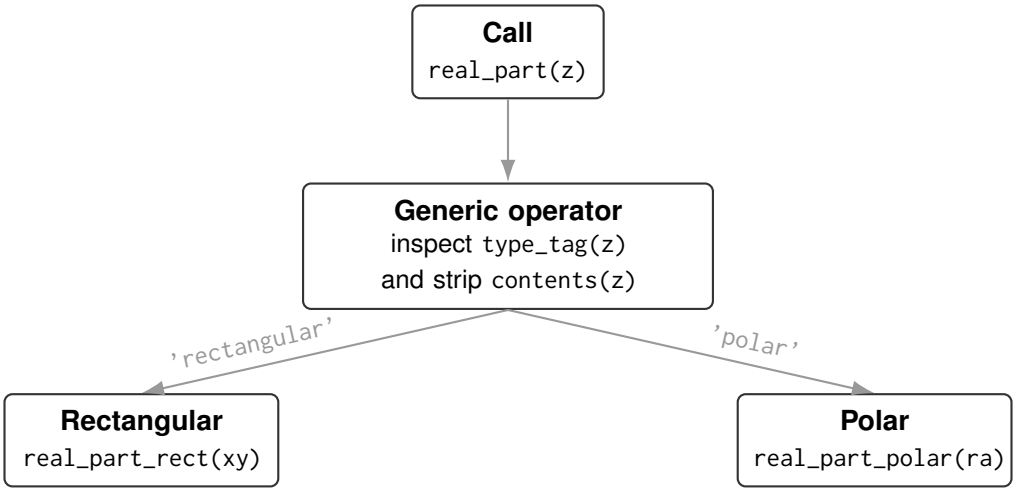


Figure 11.2: Generic operators dispatch using type tags.

This approach has two practical drawbacks:

- **Generic operators must know all representations.** If we add a new representation, we must edit *every* generic operator.
- **Name conflicts remain awkward.** Representation-specific functions must be renamed (e.g. `real_part_rect`) or isolated in separate modules.

Dispatch on type is fine for small systems, but becomes brittle as the number of types grows.

11.4 Dictionaries

Python provides a built-in `dict` type: a **mutable mapping** from **keys** to **values**. Dictionaries are implemented using a **hash table**, so most key-based operations run in $O(1)$ average time.

A dictionary d guarantees:

- **Unique keys:** Each key appears at most once; assigning `d[k] = v` overwrites the old value for `k`.
- **Fast lookup:** `d[k]`, `k in d`, insert, and delete are typically $O(1)$ on average.
- **Iteration order is insertion order (Python 3.7+):** Iterating over a dict visits keys in the order they were inserted.

Insertion order is consistent but not sorted

Dictionary order is **deterministic** and follows insertion order in modern Python, but it is **not sorted**. Deleting and re-inserting a key moves it to the end. If you need a sorted traversal, use `for k in sorted(d): ...`.

11.4.1 Creating dictionaries and basic usage

Construction.

```
d1 = {}                              # empty dict
d2 = {'a': 1, 'b': 2}                # dict literal
d3 = dict([('x', 10), ('y', 20)])# from (key,value) pairs
d4 = dict(temp=26.8, wind=0)      # keyword-style keys (must be valid
    identifiers)
```

Lookup, insert/overwrite, and deletion.

```
weather = {'temp': 26.8, 'wind': 0}

print(weather['temp'])       # lookup by key
print('wind' in weather)     # membership checks keys

weather['desc'] = 'cloudy'   # insert a new key
weather['temp'] = 31.1       # overwrite an existing key

del weather['wind']          # delete a key-value pair
```

Hashability. Just like sets, keys must be hashable (e.g. numbers, strings, tuples of hashables).

```
d = {(1, 2): 'ok'}      # OK: tuple is hashable
d = {[1, 2]: 'bad'}     # ERROR: list is unhashable
```

11.4.2 Construction and Basic Operations

The most common dictionary operations are summarised in Table 11.1. Here n denotes the number of entries in the dictionary, and m is the number of pairs being merged/added from another mapping or iterable.

Operation	Syntax / Example	Time	Notes
Constructors	`{}` `dict()`	$O(1)$	Create an empty dictionary.
	`{'a': 1, 'b': 2}`	$O(n)$	Literal construction inserts all pairs.
	`dict(iterable_of_pairs)`	$O(n)$ avg.	Builds a dict by inserting pairs `(k, v)`.
	`dict(k1=v1, k2=v2)`	$O(n)$ avg.	Keyword-style constructor; keys become strings.
Lookup	`d[k]`	$O(1)$ avg.	Returns value; raises `KeyError` if `k` absent.
Safe lookup	`d.get(k, default)`	$O(1)$ avg.	Returns `default` instead of raising if missing.
Insert / overwrite	`d[k] = v`	$O(1)$ avg.	Adds key if absent; overwrites value if present.
Delete	`del d[k]`	$O(1)$ avg.	Removes key; raises `KeyError` if missing.
Membership (keys)	`k in d`	$O(1)$ avg.	Tests whether `k` is a key in `d`.
Length	`len(d)`	$O(1)$	Size is stored in the dict header.
Iteration (keys)	`for k in d: ...`	$O(n)$	Iterates over keys in insertion order (Python 3.7+).
Iteration (pairs)	`for k, v in d.items(): ...`	$O(n)$	Iterates over `(key, value)` pairs.
Merge (new dict)	`d3 = d1 | d2`	$O(n+m)$	Python 3.9+: returns a new dict; right operand wins on conflicts.
Update (in place)	`d1.update(d2)`	$O(m)$ avg.	Inserts/overwrites pairs from `d2` into `d1`.

Table 11.1: Basic `dict` operations: syntax, time complexity, and behavior.

11.4.3 Dictionary methods

A dictionary maps *keys* to *values*. It is implemented as a hash table, so most key-based operations are $O(1)$ on average.

Method	Example	Time	Effect
`keys()`	`d.keys()`	$O(1)$ (view)	Return a dynamic view object over all keys. Iterating over the keys takes $O(n)$ time.
`values()`	`d.values()`	$O(1)$ (view)	Return a view object over all values.
`items()`	`d.items()`	$O(1)$ (view)	Return a view over `(key, value)` pairs.
`get(k, default)`	`d.get('id', 0)`	$O(1)$ avg.	Return value for `'id'` if present; otherwise return `0` (or a provided default) without raising an error.
`update(other)`	`x': 1d.update(')`	$O(m)$	Insert or overwrite m key–value pairs from `other` (another dict or iterable of pairs).
`pop(k)`	`d.pop('x')`	$O(1)$ avg.	Remove and return value associated with key `'x'`, or raise `KeyError` if missing (unless default given).
`popitem()`	`d.popitem()`	$O(1)$	Remove and return one key–value pair (in Python 3.7+, the last inserted).
`setdefault(k, default)`	`d.setdefault('x', 0)`	$O(1)$ avg.	If key `'x'` exists, return its value; otherwise insert `('x', 0)` and return `0`.
`clear()`	`d.clear()`	$O(n)$	Remove all key–value pairs from the dictionary.
`copy()`	`d2 = d.copy()`	$O(n)$	Shallow copy: new dictionary object, but values are the same underlying objects.

Table 11.2: Specialised `dict` operations: syntax, time complexity, and behavior.

11.4.4 Dictionaries as tables: put and get

Dictionaries are a natural tool for **tables**: store items in a table indexed by keys, and retrieve them later by the same keys. This is exactly the mechanism we will use in data-directed programming.

Connection to data-directed programming

A dispatch table is just a dictionary (often nested) that maps `(operation, type-tags)` to the correct implementation procedure. Adding a new operation or a new type becomes *adding entries to the table*.

11.5 Data-Directed Programming

Data-directed programming replaces big `if`/`elif` chains with a **method table**:

$$\texttt{(operation, type-tags)} \mapsto \texttt{procedure}.$$

This is called **data-directed programming** because the data's type tags direct the lookup.

11.5.1 A method table: put and get

We implement a method table using a dictionary-of-dictionaries.

```
method_table = {}

def put(op, type_key, proc):
    """
    Install proc under (op, type_key).

    type_key is either:
      - a tuple of tags for operations on tagged data, e.g. ('polar',)
          or ('complex','complex'),
      - a single tag string for constructors, e.g. 'rectangular' or '
          polar'.
    """
    if op not in method_table:
        method_table[op] = {}
    method_table[op][type_key] = proc

def get(op, type_key):
    """Retrieve proc installed under (op, type_key), or None."""
    return method_table.get(op, {}).get(type_key, None)
```

11.5.2 Variadic arguments (*args)

In Python, a function can accept a **variable number of positional arguments** using the `*args` syntax.

Packing arguments. When defining a function, `*args` *collects* extra positional arguments into a tuple.

```
def f(x, y, *args):
    # x and y are required
    # args is a tuple of any remaining positional arguments
    return (x, y, args)

print(f(1, 2))           # (1, 2, ())
print(f(1, 2, 3, 4, 5))  # (1, 2, (3, 4, 5))
```

Unpacking arguments. When calling a function, `*tuple_or_list` *spreads* the sequence into positional arguments.

```
def add3(a, b, c):
    return a + b + c

xs = (10, 20, 30)
print(add3(*xs))   # same as add3(10, 20, 30)
```

Generic arithmetic operators (e.g. `add`, `mul`) may need to work on one or two inputs (or sometimes more). The `*args` mechanism lets us write a single dispatcher that handles any arity.

```
def apply_generic(op, *args):
    """
    Look up the correct implementation procedure based on:
      - operation name op
      - the type tags of all arguments
    then apply it to the arguments.
    """
    type_tags = tuple(map(type_tag, args))   # e.g. ('complex','complex')
    proc = get(op, type_tags)                # lookup in dispatch table
    return proc(*args)                       # apply proc to the same args

def type_tag(datum):
    """Extract the tag from a tagged datum."""
    if type(datum) == tuple and len(datum) == 3:
        return datum[0]
    raise Exception("Bad tagged datum (type_tag): " + str(datum))
```

`*args` lets `apply_generic` support **unary** operators (one argument) and **binary** operators (two arguments) without writing separate dispatcher functions.

Looking Forward: Variadic Arguments

Python's `*args` and `**kwargs` are powerful, and other languages have similar ideas.

- **C**: Uses `<stdarg.h>` with macros like `va_start` and `va_arg`. It is type-unsafe and relies on the programmer to know the argument count (e.g. `printf`).
- **Java**: Uses the ellipsis syntax (`Type... name`). Under the hood, this is purely syntactic sugar for passing an array.
- **JavaScript (ES6)**: Uses the *rest parameter* syntax (`...args`), which is functionally very similar to Python's `*args`, gathering excess arguments into an array.
- **C++**: Modern C++ uses *variadic templates* (`template<typename... Args>`), which allow for type-safe compile-time recursion over arguments.

In CS2030/S, you will learn more about variadics.

11.5.3 Generic dispatcher: `apply_generic`

The dispatcher:

1. reads the tags of the arguments,
2. retrieves the method,
3. strips tags and calls the method on raw contents.

Using `apply_generic` from above, generic selectors are now thin wrappers. `apply_generic` retrieves the appropriate function `proc` from the table based on the operator and the value:

```
def real_part(z):
   return apply_generic('real_part', z)
def imag_part(z):
   return apply_generic('imag_part', z)
def magnitude(z):
   return apply_generic('magnitude', z)
def angle(z):
    return apply_generic('angle', z)
```

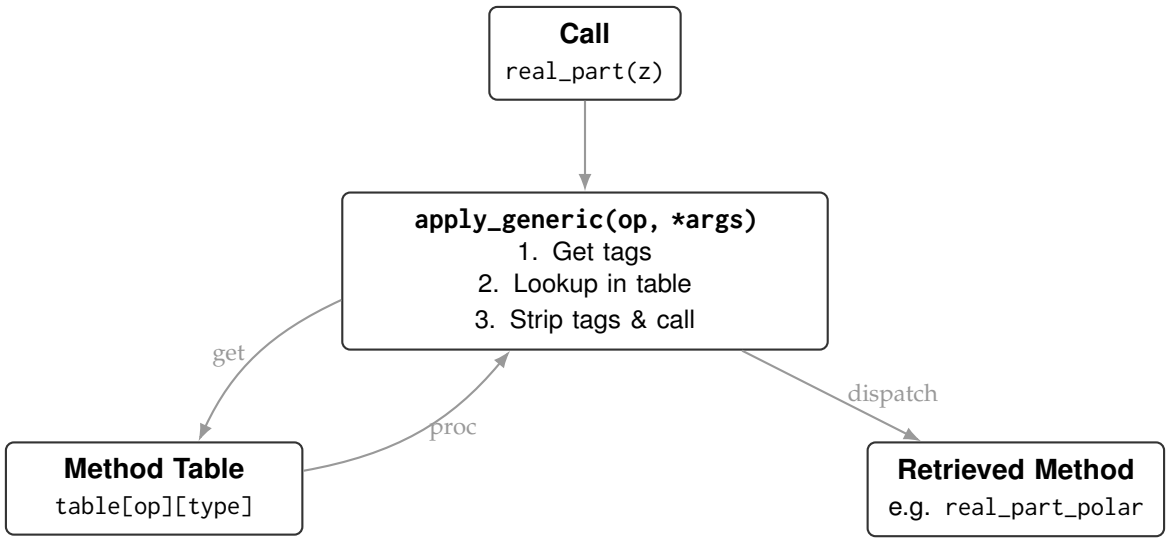


Figure 11.3: Data-directed programming using table lookup.

11.5.4 Installing packages eliminates name conflicts

A major advantage of data-directed programming is that we install each representation through an **installer function**. Internal helper names are local to the installer, so different packages can reuse names like `real_part` without collisions.

Rectangular installer

```
import math

def install_rectangular_package():
    # raw selectors for rectangular data (x,y)
    def real_part(z): return z[0]
    def imag_part(z): return z[1]

    def magnitude(z):
        return math.hypot(real_part(z), imag_part(z))

    def angle(z):
        return math.atan2(imag_part(z), real_part(z))

    def make_from_real_imag(x, y):
        return (x, y)

    def make_from_mag_ang(r, a):
        return (r * math.cos(a), r * math.sin(a))

    def tag(raw_xy):
        return attach_tag('rectangular', raw_xy)

    # register selector methods (receive raw contents)
    put('real_part', ('rectangular',), real_part)
    put('imag_part', ('rectangular',), imag_part)
    put('magnitude', ('rectangular',), magnitude)
    put('angle',     ('rectangular',), angle)

    # register constructors (policy: choose target representation
       explicitly)
    put('make_from_real_imag', 'rectangular',
        lambda x, y: tag(make_from_real_imag(x, y)))
    put('make_from_mag_ang', 'rectangular',
        lambda r, a: tag(make_from_mag_ang(r, a)))

    return 'done'
```

Polar installer

```
import math

def install_polar_package():
    def normalize_angle(a):
        return math.atan2(math.sin(a), math.cos(a))

    def make_from_mag_ang(r, a):
        if r < 0:
            r = -r
            a = a + math.pi
        return (r, normalize_angle(a))

    def magnitude(z): return z[0]
    def angle(z):     return z[1]

    def real_part(z):
        return magnitude(z) * math.cos(angle(z))

    def imag_part(z):
        return magnitude(z) * math.sin(angle(z))

    def make_from_real_imag(x, y):
        return make_from_mag_ang(math.hypot(x, y), math.atan2(y, x))

    def tag(raw_ra):
        return attach_tag('polar', raw_ra)

    put('real_part', ('polar',), real_part)
    put('imag_part', ('polar',), imag_part)
    put('magnitude', ('polar',), magnitude)
    put('angle',     ('polar',), angle)

    put('make_from_real_imag', 'polar',
        lambda x, y: tag(make_from_real_imag(x, y)))
    put('make_from_mag_ang', 'polar',
        lambda r, a: tag(make_from_mag_ang(r, a)))

    return 'done'
```

By using installers to register internal functions into a central table, we effectively create distinct **namespaces** for each data representation. Therefore, we have:

- **Scope Isolation:** Installers act as private boundary. Internal helper function names (e.g. a local `magnitude` calculation) are scoped to that namespace and cannot accidentally collide with global names or other representations.
- **Modularity:** Adding a new data representation means introducing a new, self-contained namespace. You do not have to worry about breaking existing namespaces when "installing" the new one.

11.6 Generic Arithmetic and Layering

Once we can dispatch on tags, we can go beyond complex numbers. We can build a **generic arithmetic system** with operations like: `add`, `sub`, `mul`, `div` that work across many types (integers, rationals, complex, polynomials, . . .).

11.6.1 The layering principle

As data crosses abstraction layers:

- Going **down** a layer, we often **strip** a tag to access raw contents.
- Going **up** a layer, we often **attach** a new tag that describes the returned value.

This idea appears throughout computing (e.g. network protocol stacks).

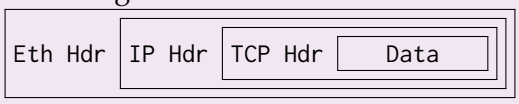
Looking Forward: Encapsulation in networking

The Internet relies on a strict hierarchy of abstraction layers, where **headers act as tags**. When your browser sends a request, each layer adds a specific tag to ensure delivery. The layer below treats the entire package from the layer above as opaque *contents*.

- **Application:** Starts with data (e.g. HTTP GET).
- **Transport:** Wraps data with a **TCP Tag** (Source/Dest Ports).
- **Network:** Wraps the TCP segment with an **IP Tag** (Source/Dest IP Address).
- **Link:** Wraps the IP packet with an **Ethernet Tag** (MAC Address).

When a server receives the signal, it reverses the process. It strips the Ethernet tag, checks the protocol type, strips the IP tag, checks the ports, and finally delivers the raw data to the web server software.

Thus, a packet has the following structure:



If we build a **complex type** as one case inside a larger numeric hierarchy, we often use:

- an **outer type tag** (e.g. `complex`) to participate in generic arithmetic, and
- an **inner representation tag** (e.g. `rectangular/polar`) to choose storage.

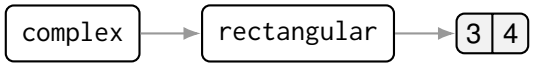


Figure 11.4: Memory view: The representation tag points to the data structure (tuple).

11.6.2 Top-level generic arithmetic

We define the generic operations:

```
def add(x, y):
    return apply_generic('add', x, y)
def sub(x, y):
    return apply_generic('sub', x, y)
def mul(x, y):
    return apply_generic('mul', x, y)
def div(x, y):
    return apply_generic('div', x, y)
```

In general, we can think of it as having multiple layers of abstraction, as seen below. We use data-directed programming with multiple layers of tags to handle different kinds of arithmetic. Our package becomes extensible, and we only need to write the installer for the new package.

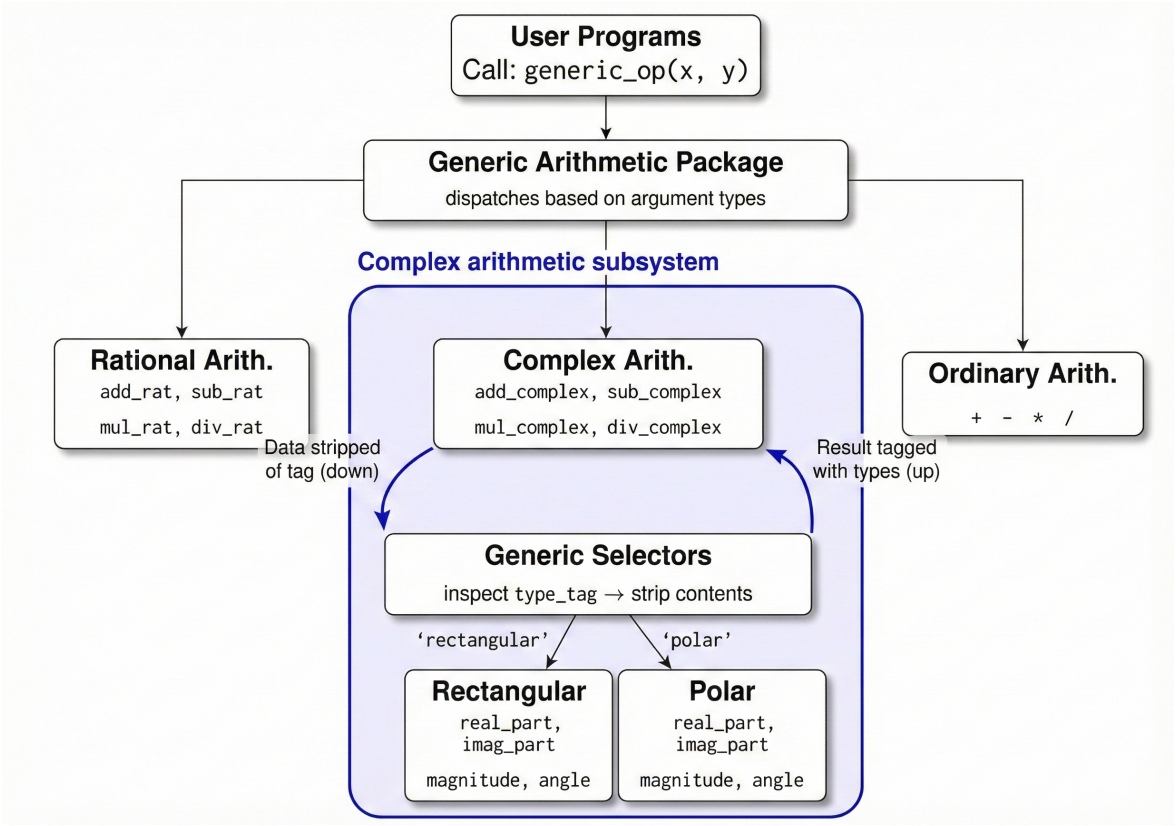


Figure 11.5: Generic Arithmetic Architecture.

Then we add a complex package that teaches the table how to handle `('complex','complex')`.

```
def install_complex_package():
    # internal constructors that delegate to the lower-level complex
       constructors
    def make_from_real_imag(x, y):
        # choose rectangular storage for (x,y)
        return get('make_from_real_imag', 'rectangular')(x, y)

    def make_from_mag_ang(r, a):
        # choose polar storage for (r,a)
        return get('make_from_mag_ang', 'polar')(r, a)

    # internal arithmetic on complex values (these use generic
       selectors)
    def add_complex(z1, z2):
        return make_from_real_imag(real_part(z1) + real_part(z2),
                                   imag_part(z1) + imag_part(z2))

    def mul_complex(z1, z2):
        return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                                 angle(z1) + angle(z2))

    def tag(inner_rep_value):
        # outer tag marks this as a complex number in the numeric
           hierarchy
        return attach_tag('complex', inner_rep_value)

    # methods installed under the outer type tag; note: args arrive as
        raw contents
    put('add', ('complex','complex'),
        lambda z1, z2: tag(add_complex(z1, z2)))
    put('mul', ('complex','complex'),
        lambda z1, z2: tag(mul_complex(z1, z2)))

    return 'done'
```

Key idea: layered dispatch

When you compute `add(z1,z2)`:

- first dispatch happens at the **outer** level: `('complex','complex')`,
- then `add_complex` calls selectors like `real_part`, which dispatch again at the **inner** level: `'rectangular'` vs. `'polar'`.

Layering is powerful, but it also means your system performs multiple dispatches per operation.

11.7 Coercion and Type Hierarchies

When generic operations are applied to arguments of mixed types (e.g. $z + q$ where $z \in \mathbb{C}, q \in \mathbb{Q}$), the system must reconcile the representation differences. If the method table contains no entry for the specific type signature (T_1, T_2), we resort to **coercion**.

Formally, coercion is a mapping function $C : T_{src} \to T_{dst}$ that transforms a value of a source type into an equivalent (or approximate) value of a destination type, thereby enabling the dispatch of a defined operator.

11.7.1 Type Hierarchies

In arithmetic design, types often form a linear ordering or a hierarchy based on the mathematical subset relationships of their domains:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

This structure implies that every integer is technically a rational number, every rational is a real, and so forth. We can visualize this hierarchy and the directional operations within it:

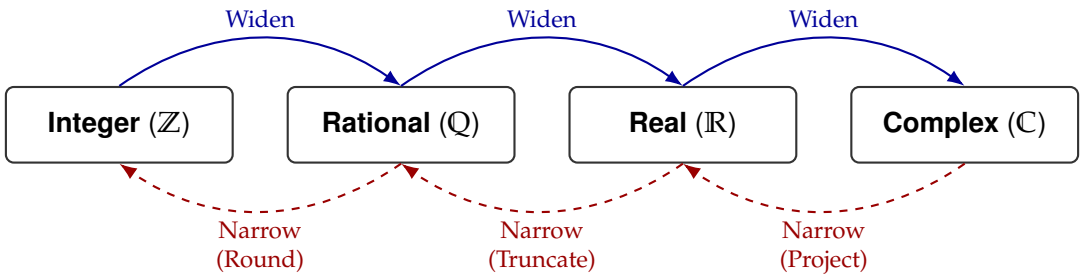


Figure 11.6: Numeric hierarchy. Moving right widens safely; moving left narrows and may lose information.

11.7.2 Widening vs. Narrowing

Type conversion is categorized based on information preservation:

Widening (Upcasting)

Widening converts a value from a specific type to a more general type.

- **Safety:** Usually safe and information-preserving (injective).
- **Example:** Converting 3 (Integer) to 3.0 (Real). The value represents the exact same point on the number line.
- **Usage:** Most generic arithmetic systems perform widening *implicitly* to find a common supertype.

Narrowing (Downcasting)

Narrowing converts a value from a general type to a more specific type.

- **Safety:** Often unsafe or lossy (non-injective). It involves projection, truncation, or approximation.
- **Example:** Converting 3.14 (Real) to 3 (Integer) loses precision. Converting $2 + 5i$ (Complex) to 2 (Real) loses a dimension.
- **Usage:** Systems rarely perform narrowing implicitly. It is usually reserved for explicit casting operations (e.g. `int(3.5)`).

Looking Forward: Formal Subtyping & Java Typecasting

An introduction to type theory will be formally discussed in CS2030/S.

1. Formal Definition ($<:$)

Type T is a *subtype* of S (written $T <: S$) if any program written for S can safely accept T. This relation forms a **partial order** with three key properties:

- **Reflexive:** $S <: S$ (A type is a subtype of itself).
- **Transitive:** $S <: T \land T <: U \implies S <: U$ (Inheritance chains).
- **Anti-symmetric:** $S <: T \land T <: S \implies S = T$ (No cycles allowed).

Note: T is a **proper subtype** ($T < S$) if $T <: S \land T \neq S$.

2. Application in Java

Java uses this partial order to determine legality of conversions:

- **Widening (Upcasting):** Converting $S \to T$.
  - **Rule:** Allowed implicitly if and only if $S <: T$.
  - **Safety:** Always type-safe (guaranteed by the Liskov principle).
- **Narrowing (Downcasting):** Converting $T \to S$.
  - **Rule:** If $S <: T$ (proper subtype), the compiler cannot guarantee safety. You must use the explicit cast operator: `S s = (S) t;`.
  - **Risk:** May throw `ClassCastException` at runtime if the object is not actually an instance of S.

11.7.3 Implementing Widening Typecasts

When implementing mixed-mode arithmetic (e.g. `add(x, y)`), we avoid defining N^2 separate methods for every combination of types. Instead, we define operations only for identical types (e.g. `add(Complex, Complex)`) and use the hierarchy to bridge gaps.

1. **Direct Lookup:** Attempt to find a method for the exact types `(type(x), type(y))`. If found, apply it.
2. **Level Check:** Determine the position of both types in the numeric hierarchy.
3. **Recursive Coercion:**
   - If level(x) < level(y), coerce x up one step: `add(raise(x), y)`.
   - If level(y) < level(x), coerce y up one step: `add(x, raise(y))`.
4. **Repeat:** Recursive calls continue raising the lower type until both arguments reach a common supertype, or the top of the hierarchy is reached without a match (error).

11.8 Message Passing

In the previous strategies, we treated **functions** as the "intelligent" component: functions inspect tags and decide what to do.

In **message passing**, the **data** is "intelligent": each datum is represented as a *dispatch function* that accepts a message (operation name) and returns the corresponding result.

For example, consider the following implementation of rectangular complex numbers as a dispatch function.

```
import math

def make_from_real_imag(x, y):
    """
    Return a dispatch function representing x + iy.
    The returned function accepts a message op and returns the
       corresponding value.
    """
    def dispatch(op):
        if op == 'real_part':
            return x
        elif op == 'imag_part':
            return y
        elif op == 'magnitude':
            return math.hypot(x, y)
        elif op == 'angle':
            return math.atan2(y, x)
        else:
            raise Exception("Unknown op -- make_from_real_imag: " + op
               )
    return dispatch
```

Selectors now send messages to the datum:

```
def real_part(z):
    return z('real_part')
def imag_part(z):
    return z('imag_part')
def magnitude(z):
    return z('magnitude')
def angle(z):
    return z('angle')
```

Then, we can write a similar function for the polar representation of a complex number:

```
import math

def make_from_mag_ang(r, a):
    def dispatch(op):
        if op == 'real_part':
            return r * math.cos(a)
        elif op == 'imag_part':
            return r * math.sin(a)
        elif op == 'magnitude':
            return r
        elif op == 'angle':
            return a
        else:
            raise Exception("Unknown op -- make_from_mag_ang: " + op)
    return dispatch
```

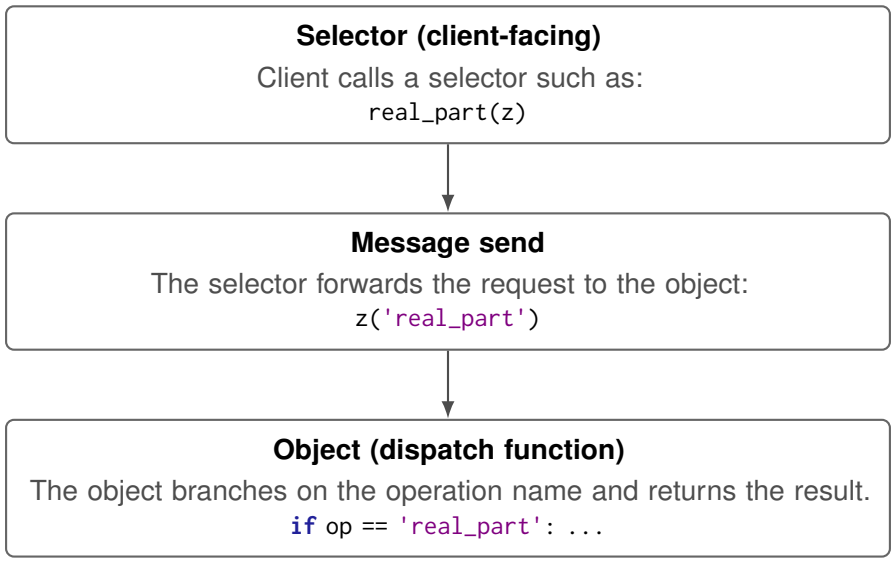


Figure 11.7: Message passing: The object itself dispatches on the requested operation.

Connection to object-oriented programming

Message passing is a foundational idea behind OOP:

- data are objects,
- operations are messages/method calls,
- objects know how to respond to messages.

However,

- Adding a *new operation* may require editing every constructor (each object must learn the new message).
- Messages are strings: spelling mistakes become runtime errors.

11.9 Summary: Choosing a Strategy

All three strategies are ways to manage change:

- adding a new **representation/type**,
- adding a new **operation**,
- avoiding **name conflicts**,
- minimizing ripple effects across modules.

Strategy	Core idea	Main engineering consequence
Dispatch on type	Generic operators use `if`/`elif` on tags	Simple, but generic operators must be edited when new representations are added.
Data-directed programming	Look up `(op, type-tags)` in a method table	Extensible: adding new representations installs new entries; avoids name conflicts via installers.
Message passing	Data are dispatch functions; operations are messages	Encapsulation shifts to data; resembles OOP; adding new operations can be intrusive.