

Generics



EECS3311 A: Software Design
Fall 2019

CHEN-WEI WANG

Motivating Example: A Book of Any Objects

```
class BOOK
  names: ARRAY[STRING]
  records: ARRAY[ANY]
  -- Create an empty book
  make do ... end
  -- Add a name-record pair to the book
  add (name: STRING; record: ANY) do ... end
  -- Return the record associated with a given name
  get (name: STRING): ANY do ... end
end
```

Question: Which line has a type error?

```
1 birthday: DATE; phone_number: STRING
2 b: BOOK; is_wednesday: BOOLEAN
3 create {BOOK} b.make
4 phone_number := "416-677-1010"
5 b.add ("SuYeon", phone_number)
6 create {DATE} birthday.make(1975, 4, 10)
7 b.add ("Yuna", birthday)
8 is_wednesday := b.get("Yuna").get_day_of_week = 4
```

Motivating Example: Observations (1)

- In the `BOOK` class:
 - In the attribute declaration

```
records: ARRAY[ANY]
```

- **ANY** is the most general type of records.
- Each book instance may store any object whose *static type* is a **descendant class** of **ANY**.
- Accordingly, from the return type of the `get` feature, we only know that the returned record has the static type **ANY**, but not certain about its *dynamic type* (e.g., `DATE`, `STRING`, *etc.*).
∴ a record retrieved from the book, e.g., `b.get("Yuna")`, may only be called upon features defined in its *static type* (i.e., **ANY**).
- In the tester code of the `BOOK` class:
 - In **Line 1**, the *static types* of variables `birthday` (i.e., `DATE`) and `phone_number` (i.e., `STRING`) are **descendant classes** of **ANY**.
∴ **Line 5** and **Line 7** compile.

Motivating Example: Observations (2)

Due to **polymorphism**, in a collection, the *dynamic types* of stored objects (e.g., `phone_number` and `birthday`) need not be the same.

- Features specific to the *dynamic types* (e.g., `get_day_of_week` of class `Date`) may be new features that are not inherited from ANY.
- This is why **Line 8** would fail to compile, and may be fixed using an explicit **cast**:

```
check attached {DATE} b.get("Yuna") as yuna_bday then
  is_wednesday := yuna_bday.get_day_of_week = 4
end
```

- But what if the *dynamic type* of the returned object is not a DATE?

```
check attached {DATE} b.get("SuYeon") as suyeon_bday then
  is_wednesday := suyeon_bday.get_day_of_week = 4
end
```

⇒ An **assertion violation** at *runtime*!

Motivating Example: Observations (2.1)

- It seems that a combination of `attached check` (similar to an `instanceof check` in Java) and type cast can work.
- Can you see any potential problem(s)?
- **Hints:**
 - Extensibility and Maintainability
 - What happens when you have a large number of records of distinct *dynamic types* stored in the book (e.g., `DATE`, `STRING`, `PERSON`, `ACCOUNT`, `ARRAY_CONTAINER`, `DICTIONARY`, *etc.*)? [all classes are descendants of **ANY**]

Motivating Example: Observations (2.2)

Say a client stores 100 distinct record objects into the book.

```
rec1: C1
... -- declarations of rec2 to rec99
rec100: C100
create {C1} rec1.make(...) ; b.add(..., rec1)
... -- additions of rec2 to rec99
create {C100} rec100.make(...) ; b.add(..., rec100)
```

where **static types** C1 to C100 are **descendant classes** of ANY.

- **Every time** you retrieve a record from the book, you need to check “exhaustively” on its **dynamic type** before calling some feature(s).

```
-- assumption: 'f1' specific to C1, 'f2' specific to C2, etc.
if attached {C1} b.get("Jim") as c1 then
  c1.f1
... -- cases for C2 to C99
elseif attached {C100} b.get("Jim") as c100 then
  c100.f100
end
```

- Writing out this list multiple times is tedious and error-prone!

Motivating Example: Observations (3)

We need a solution that:

- Eliminates runtime assertion violations due to wrong casts
- Saves us from explicit `attached` checks and type casts

As a sketch, this is how the solution looks like:

- When the user declares a `BOOK` object `b`, they must commit to the kind of record that `b` stores at runtime.
e.g., `b` stores either `DATE` objects (and its **descendants**) only or `String` objects (and its **descendants**) only, but **not a mix**.
- When attempting to store a new record object `rec` into `b`, if `rec`'s *static type* is not a **descendant class** of the type of book that the user previously commits to, then:
 - It is considered as a **compilation error**
 - Rather than triggering a **runtime assertion violation**
- When attempting to retrieve a record object from `b`, there is **no longer a need** to check and cast.

∴ Static types of all records in `b` are guaranteed to be the same.

Parameters

- In mathematics:
 - The same *function* is applied with different *argument values*.
e.g., $2 + 3$, $1 + 1$, $10 + 101$, *etc.*
 - We **generalize** these instance applications into a definition.
e.g., $+: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$ is a function that takes two integer **parameters** and returns an integer.
- In object-oriented programming:
 - We want to call a *feature*, with different *argument values*, to achieve a similar goal.
e.g., `acc.deposit(100)`, `acc.deposit(23)`, *etc.*
 - We **generalize** these possible feature calls into a definition.
e.g., In class `ACCOUNT`, a feature `deposit(amount: REAL)` takes a real-valued **parameter**.
- When you design a mathematical function or a class feature, always consider the list of **parameters**, each of which representing a set of possible *argument values*.

Generics: Design of a Generic Book

```
class BOOK[ G ]  
  names: ARRAY[ STRING ]  
  records: ARRAY[ G ]  
  -- Create an empty book  
  make do ... end  
  /* Add a name-record pair to the book */  
  add (name: STRING; record: G) do ... end  
  /* Return the record associated with a given name */  
  get (name: STRING): G do ... end  
end
```

Question: Which line has a type error?

```
1 birthday: DATE; phone_number: STRING  
2 b: BOOK[DATE]; is_wednesday: BOOLEAN  
3 create BOOK[DATE] b.make  
4 phone_number = "416-67-1010"  
5 b.add ("SuYeon", phone_number)  
6 create {DATE} birthday.make (1975, 4, 10)  
7 b.add ("Yuna", birthday)  
8 is_wednesday := b.get("Yuna").get_day_of_week == 4
```

Generics: Observations

- In class `BOOK`:
 - At the class level, we *parameterize the type of records* :


```
class BOOK[G]
```
 - Every occurrence of `ANY` is replaced by `E`.
- As far as a client of `BOOK` is concerned, they must *instantiate* `G`.
 ⇒ This particular instance of `book` must consistently store items of that instantiating type.
- As soon as `E` instantiated to some known type (e.g., `DATE`, `STRING`), every occurrence of `E` will be replaced by that type.
- For example, in the tester code of `BOOK`:
 - In **Line 2**, we commit that the book `b` will store `DATE` objects only.
 - **Line 5** fails to compile. [`∴` `STRING` not **descendant** of `DATE`]
 - **Line 7** still compiles. [`∴` `DATE` is **descendant** of itself]
 - **Line 8** does *not need* any attached check and type cast, and does *not cause* any runtime assertion violation.
 ∴ All attempts to store non-`DATE` objects are caught at *compile time*.

Bad Example of using Generics

Has the following client made an appropriate choice?

```
book: BOOK[ANY]
```

NO!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- It allows **all** kinds of objects to be stored.
 - ∴ All classes are descendants of **ANY**.
 - We can expect **very little** from an object retrieved from this book.
 - ∴ The **static type** of `book`'s items are **ANY**, root of the class hierarchy, has the **minimum** amount of features available for use.
 - ∴ Exhaustive list of casts are unavoidable.
- [**bad** for extensibility and maintainability]

Instantiating Generic Parameters

- Say the **supplier** provides a generic `DICTIONARY` class:

```
class DICTIONARY[V, K] -- V type of values; K type of keys
  add_entry (v: V; k: K) do ... end
  remove_entry (k: K) do ... end
end
```

- **Clients** use `DICTIONARY` with different degrees of instantiations:

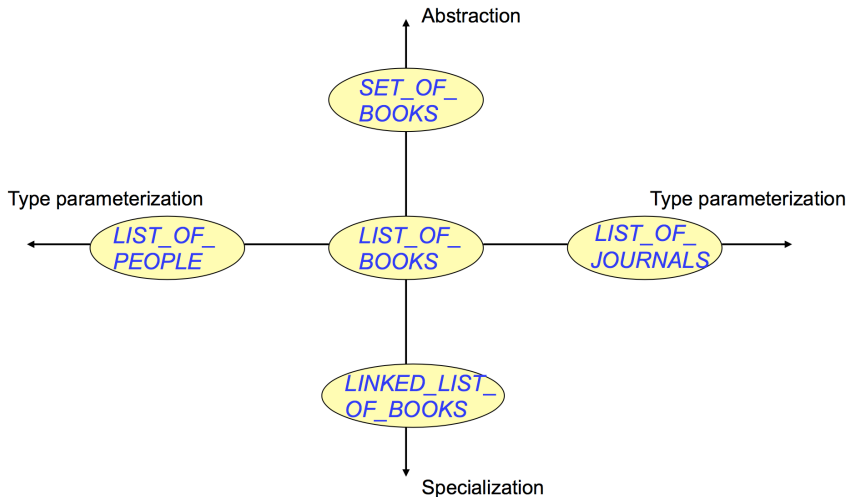
```
class DATABASE_TABLE[K, V]
  imp: DICTIONARY[V, K]
end
```

e.g., Declaring `DATABASE_TABLE[INTEGER, STRING]` instantiates
`DICTIONARY[STRING, INTEGER]`.

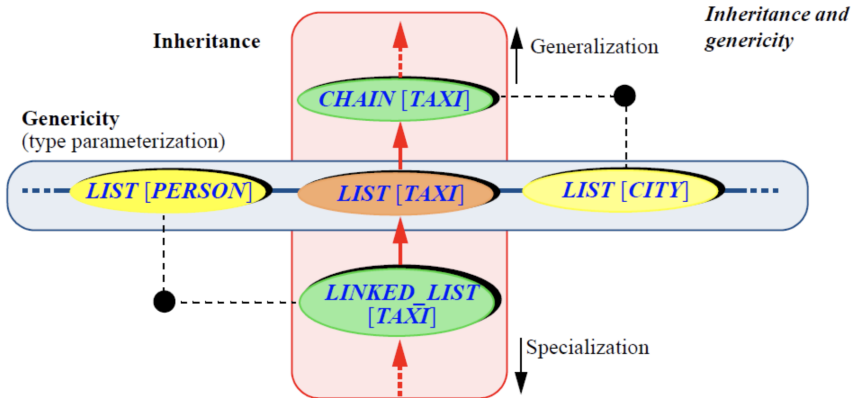
```
class STUDENT_BOOK[V]
  imp: DICTIONARY[V, STRING]
end
```

e.g., Declaring `STUDENT_BOOK[ARRAY[COURSE]]` instantiates
`DICTIONARY[ARRAY[COURSE], STRING]`.

Generics vs. Inheritance (1)



Generics vs. Inheritance (2)



Beyond this lecture ...

- Study the “Generic Parameters and the Iterator Pattern” Tutorial Videos.

Index (1)

Motivating Example: A Book of Any Objects

Motivating Example: Observations (1)

Motivating Example: Observations (2)

Motivating Example: Observations (2.1)

Motivating Example: Observations (2.2)

Motivating Example: Observations (3)

Parameters

Generics: Design of a Generic Book

Generics: Observations

Bad Example of using Generics

Instantiating Generic Parameters

Generics vs. Inheritance (1)

Generics vs. Inheritance (2)

Beyond this lecture ...