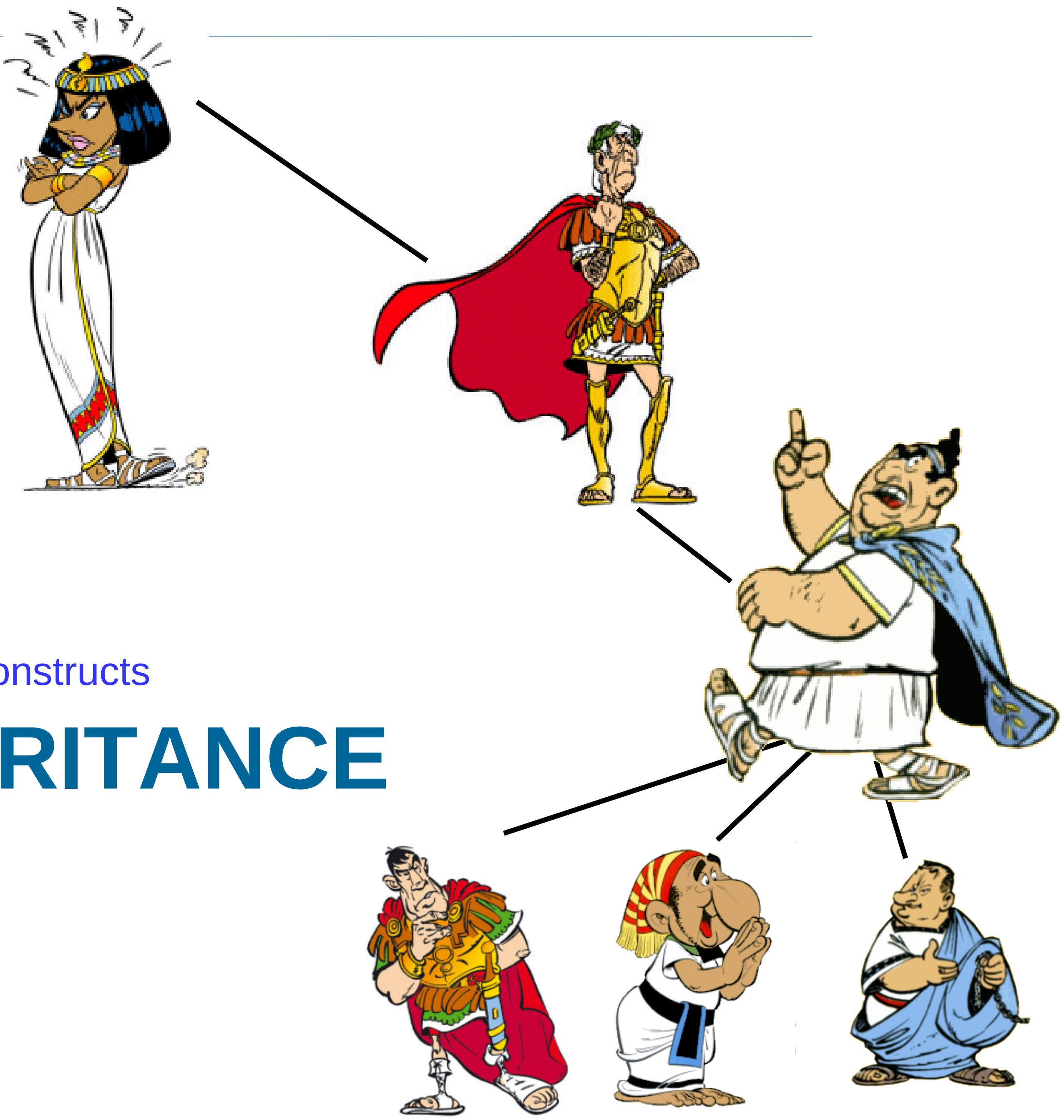


Part 1: Language constructs

1.2 INHERITANCE

Deferred Class (abstract class)

```
deferred class
    ACCOUNT

feature
    deposit (a_num: INT)
            deferred
            end
end
```

```
abstract class Account {
        abstract void deposit(int a);
}
```

A class must be **deferred** if it has at least one deferred routine. A class can be deferred without any deferred routines.

Simple Inheritance

```
class
    ACCOUNT
inherit
    ANY
end
```

```
public class Account
        extends Object {

}
```

Feature redefinition

```
class
    ACCOUNT
inherit
    ANY
            redefine out end

feature

    out: STRING
        do
            Result := "abc"
        end
end
```

```
public class Account
        extends Object {

        String toString() {
                return "abc";
        }

}
```

All routines that are redefined must be listed in the inherit clause.

Precursor call

```
class
    ACCOUNT
inherit
    ANY
            redefine out end

feature

    out: STRING
        do
        Result :=
            Precursor {ANY}
        end
end
```

```
public class Account
    extends Object {

    String toString() {
        return super();
    }

}
```

Multiple Inheritance

```
class
    A
feature
    foo do end
end
```

```
class
    B
feature
    foo do end
end
```

Option 1:

```
class
    C
inherit
      A
    B rename foo  as foo_b end
end
```

Option 2:

```
class
    C
inherit
      A
    B undefine foo end
end
```

Frozen class / frozen routine

```
frozen class
   ACCOUNT
inherit
   ANY
end
```

```
final class Account
      extends Object {
}
```

```
class
   ACCOUNT
feature
   frozen deposit (a_num: INT)
     do
        …
     end
end
```

```
class Account {
      final void deposit(final int a) {
            …
      }
}
```

A frozen class cannot be inherited; a frozen routine cannot be redefined. All arguments are frozen.

Expanded class

```
expanded class
    MY_INT
end
```

int, float, double, char

Part 1: Language constructs

1.3 EXCEPTION HANDLING

Java: Exception Handling

```
public class Printer {
    public print(int i) {
        try {
            throw new Exception()
        }
        catch(Exception e) {   }
        }
}
```

Eiffel: Exception Handling

```
class
    PRINTER
feature
    print_int (a_int: INTEGER)
        local
            l_retried: BOOLEAN
        do
            if not l_retried  then
                (create {DEVELOPER_EXCEPTION}).raise
            else
                -- Do something alternate.
            end
        rescue
            l_retried := True
            retry
    end
end
```

Part 1: Language constructs

1.4 ONCE ROUTINES

What are once routines?

```
foo: INTEGER
    once
        Result := factorial (10)
    end
test_foo
    do
        io.put_integer (foo)  -- 3628800, calculated
        io.put_integer (foo)  -- 3628800, directly returned
    end
```

- Executed the first time
- Result is stored
- In further calls, stored result is returned
- In other languages
    - Static variables
    - Singleton pattern

Use of once routines

- Constants, other than basic types

```
i: COMPLEX
    once create Result.make (0, 1)  end
```

- Lazy initialization

```
settings: SETTINGS
    once create Result.load_from_filesystem  end
```

- Initialization procedures

```
init_graphics_system
    once ... end
```

- Sharing of objects (see next)

Sharing objects

- You can share objects
- Can be used to achieve effect of global/static variables

- How?
  - Once routine returning a reference
  - Will always return the same reference
  - Create a SHARED_X class and inherit from it

Sharing objects: example

```
class SHARED_X
    the_one_and_only_x:  X
        once
            create Result.make
        end
end

class X
create {SHARED_X}
    make
feature {NONE}
    make
        do
        end
end
```

```
Class EXAMPLE1
    inherit
        SHARED_X
feature
    f
        do
          … the_one_and_only_x …
        end
end

Class EXAMPLE2
    inherit
        SHARED_X
feature
    g
        do
          … the_one_and_only_x …
        end
end
```

Part 1: Language constructs

1.5 STYLE RULES

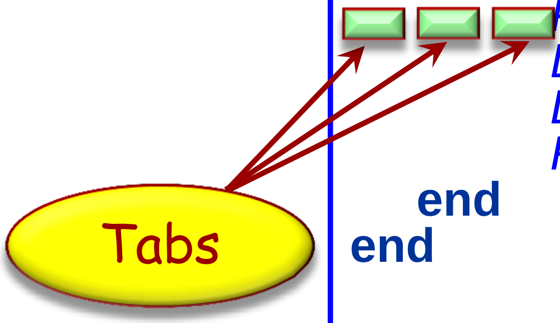
Style rule

For indentation, use tabs, not spaces

```
class
	PREVIEW

inherit
	TOURISM

feature
	explore
			-- Show city info
			-- and route.
		do
			Paris.display
			Louvre.spotlight
			Line8.highlight
			Route1.animate
		end
end
```



More style rules

- Class name: all upper-case

Full words, no abbreviations (with some exceptions)

- Classes have global namespace: two classes cannot have the same name (even in different clusters)

- Usually, classes are prefixed with a library prefix

    EiffelVision2: EV_

    Base is not prefixed

```
class
    PREVIEW

inherit
    TOURISM

feature
    explore
            -- Show city info
            -- and route.
        do

            Paris.display

            Louvre.spotlight
            Line8.highlight
            Route1.animate
        end
end
```

Even more style rules

- For feature names, use full words, not abbreviations
- Always choose identifiers that clearly identify the intended role
- Use words from natural language (preferably English) for the names you define
- For multi-word identifiers, use underscores

```
class
    PREVIEW

inherit
    TOURISM

feature
    explore
            -- Show city info
            -- and route.
        do
            Paris.display
            Louvre.spotlight
            Line8.highlight
            Line8.remove_all_sections
            Route1.animate
        end
end
```

Eiffel Naming: Locals / Arguments

- Locals and arguments share namespace with features
  - Name clashes arise when a feature is introduced, which has the same name as a local (even in parent)

- To prevent name clashes:
  - Locals are prefixed with **l_**
  - Some exceptions like “i“ exist
  - Arguments are prefixed with **a_**