Programming Language Design

9/30 Object-Oriented Programming

A First Look at OOP

- Use objects to describe components in a system
  - An object has properties
    - *data (variables) and actions (methods)*
  - An object hides details from others
    - *clients use it without knowing how it works*
    - *encapsulation*

A First Look at OOP (cont.)

- Objects can be generated according to classes
  - to share the same properties
  - Classes are a kind of mold, stamp
- Children classes (subclasses) can inherit properties from their parent classes (super classes)
  - inheritance

The history of Object-oriented

- Record classes
  - Represent and manipulate complex data structure
- Simula 67 is the first OO language
  - Use OO concept for simulation
- Smalltalk: the first pure OO language
  - Everything is an object

Reference

- *C. A. R. Hoare. Record Handling, Journal ALGOL Bulletin Issue 21, pp. 39-69, Nov. 1965.*
- *Ole-Johan Dahl. The roots of object orientation: the Simula language, Software pioneers, pp. 78-90, Springer, 2002.*
- *Adele Goldberg and David Robson. Smalltalk-80: the language and its implementation, Addison-Wesley, 1983.*

Pure Object-Oriented?

- How pure?
  - "Everything" should be an object!
- Including all definitions you might have known
  - Encapsulation, polymorphism, etc.
- All predefined types are objects
- All operations performed on objects are the methods on the objects

We are going to…

- Check the OO features in them
  - Record classes, Simula 67, Smalltalk
  - To see the evolution of OO
- And learn the details in OO
  - Since modern OO languages follow those ideas

Record handling

- Proposed by C. A. R. Hoare
  - An extension of ALGOL 60
- Records
  - Represent some objects
  - During the execution of program, records can be created or destroyed
- Record classes
  - Each record must belong to one record class
  - Declared as many as required

Reference

- *C. A. R. Hoare. Record Handling, Journal ALGOL Bulletin Issue 21, pp. 39-69, Nov. 1965.*

An example of record classes

- An example from the paper
  - To represent family relations

```
record class person;
        begin integer date of birth;
                Boolean male;
                reference father, mother,
        youngest offspring, elder sibling (person)
        end;
```

An example of records

- Suppose John is the son of Jack and Jill
  - John was born today

```
begin reference John (person);
      John:=person;
      date of birth(John):=today;
      male(John):=true;
      father(John):=Jack;
      mother(John):=Jill;
      youngest offspring(John):=null;
      elder sibling(John):=youngest offspring(Jack);
      youngest offspring(Jack):=John
end;
```

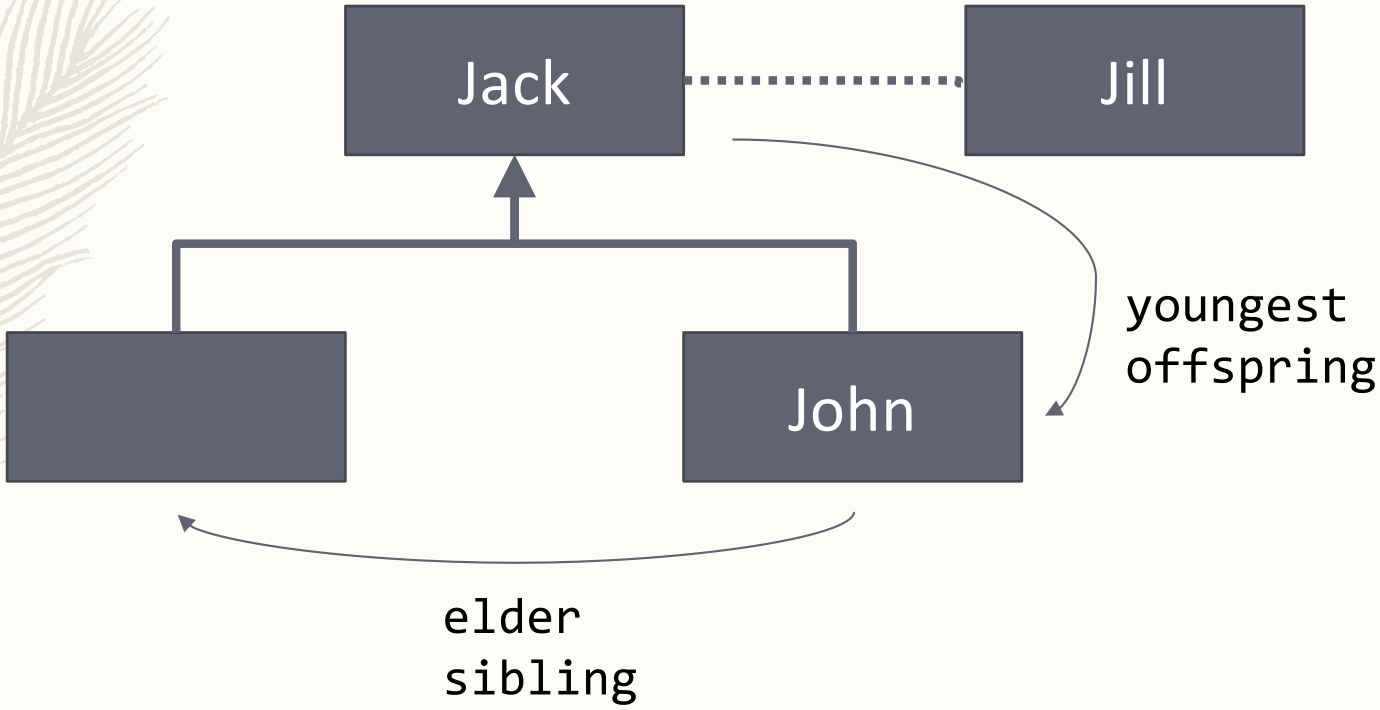
Create a record to represent John

No child yet

Set his elder sibling to the youngest offspring of Jack

Now the youngest offspring of Jack is John

An example of records (cont.)



Then we can handle records

- An example of using them
  - Determine whether Joe is a paternal uncle of Jack

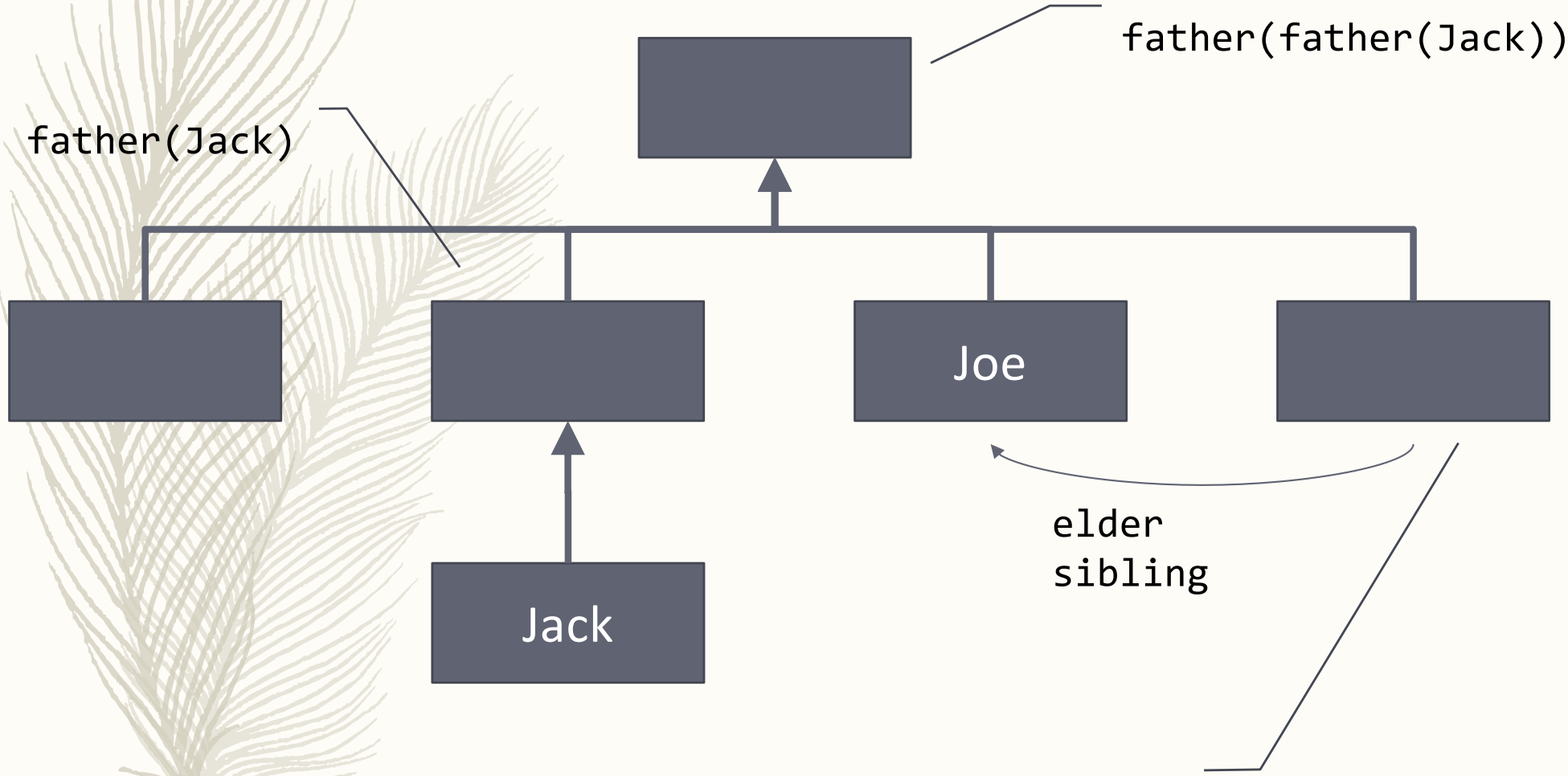
assign

i.e. `Jack.father` in Java style

```
        uncle:=false
        if Joe=father(Jack) then goto complete;
        begin reference search(person);
              search:=youngest offspring(father(father(Jack)));
try again: if search=null then goto complete;
        if search=Joe then begin uncle:=true;
                                 goto complete
                           end;
        search:=elder sibling(search);
        goto try again
end;
complete:
```

equal

Then we can handle records (cont.)



Not sufficient?

- Be able to model complex data structure
  - And handle them with procedures
- However, in record classes
  - Only attributes can be defined
  - Actions are not included in class definitions

→ How about adding actions?

Class prefixing in Simula 67

- Introduced by Ole Johan Dahl and Kristen Nygaard
  - Use the terms class, object, and subclass
  - Collect the common properties in a separate class, e.g. C
    - Use C as a prefix to another class
    - A subclass of C inherits all properties of C
  - Attributes may be redeclared in subclasses
    - As is the case of inner blocks
    - Resolved at compile-time; static

Reference

- *Ole-Johan Dahl. The roots of object orientation: the Simula language, Software pioneers, pp. 78-90, Springer, 2002.*

Class prefixing in Simula 67 (cont.)

- Concept of virtual procedures
  - A procedure P can be specified as virtual in a class C
  - Semi-dynamic

i.e. `C.P()` in Java style

- Any call for P occurring in C or in any subclass of C
  - Bind to the declaration of P occurring at the innermost
  - Binding scheme is dynamic but implemented through a table produced by the compiler

→ Modern OO languages like C++ basically are the same

Smalltalk-80

- Xerox Palo Alto Research Center (PARC)
  - Led by Alan Kay
- Smalltalk is a vision of the ways using computer power
  - A programming language and a user interface
- Based on a small number of concepts
  - Object, message, class, instance, and method
- An environment
  - Graphical and interactive
- Made up of many components
  - Integrated development environment, including debugger and inspector

To invent it

"The best way to predict the future is to invent it."

--- Alan Kay

- He is known for his ideas about
  - Object-oriented programming
  - Graphical user interface

The Dynabook proposal

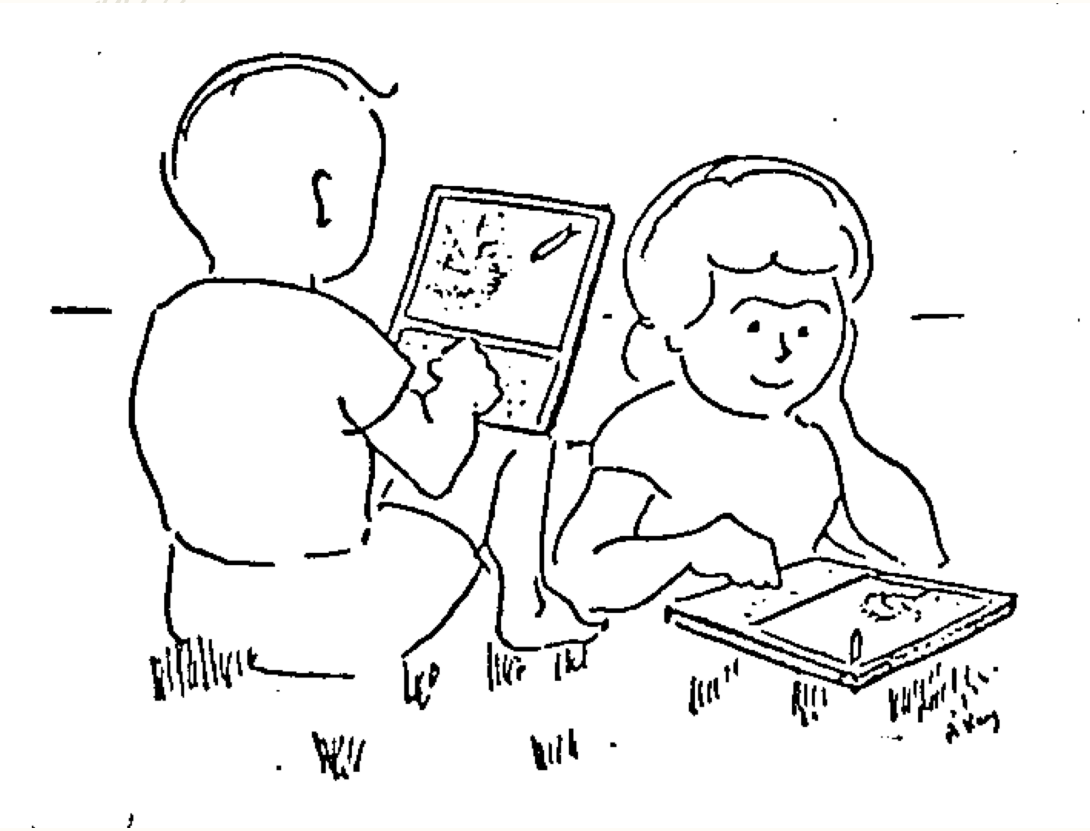
"This note speculates about the emergence of personal, portable information manipulators and their effects when used by both children and adults. Although it should be read as science fiction, current trends in miniaturization and price reduction almost guarantee that many of the notions discussed will actually happen in the near future."

Reference

– *Alan C. Kay, A Personal Computer for Children of All Ages, ACM '72 Proceedings of the ACM annual conference - Volume 1 Article No 1, 1972.*

The Dynabook proposal (cont.)

It's proposed in 1972!



The object in Smalltalk-80

- An object represents a component
  - E.g. numbers, strings, queues, rectangles, text editors, programs
- An object consists of some private memory and a set of operations
  - The nature of an object's operations depends on the type
- A message is a request for an object
  - Specify which operation is desired

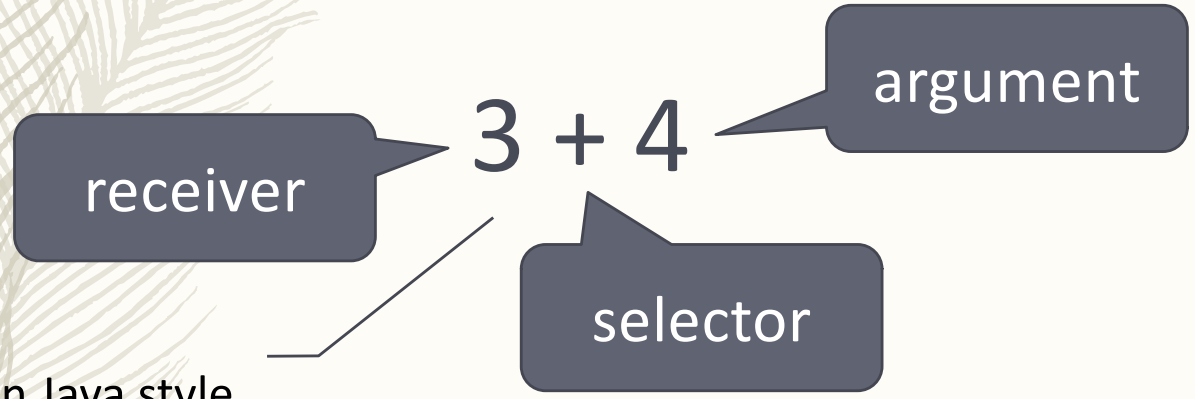
Reference

- *Adele Goldberg and David Robson. Smalltalk-80: the language and its implementation, Addison-Wesley, 1983.*

Sending a message

- A message expression describes a receiver, selector, and possibly arguments

- For example,



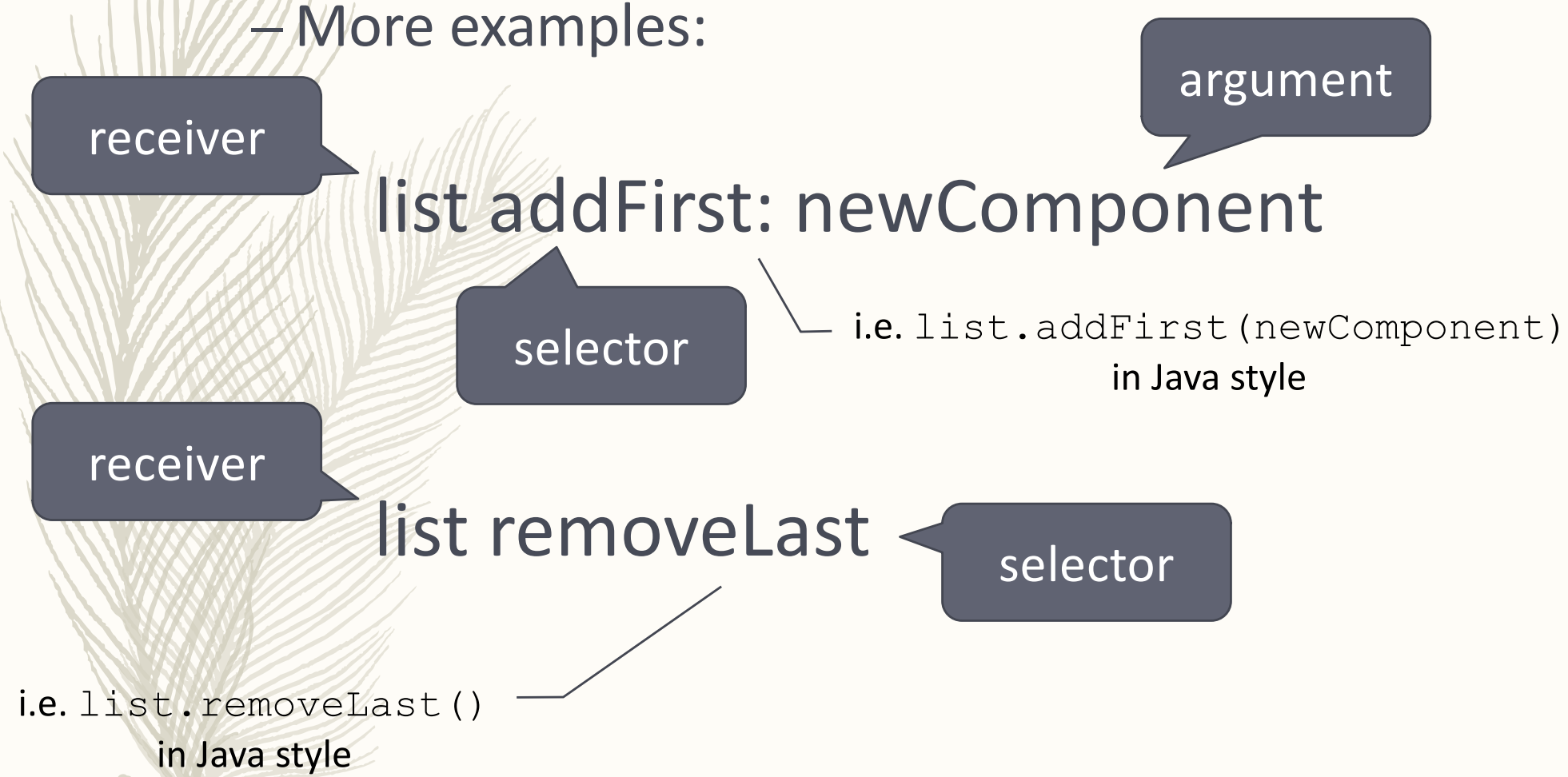
i.e. `3.+(4)` in Java style

Sending a message (cont.)

- The selector of a message determines
  - Which of the receiver's operations will be invoked
  - Arguments are other objects that are involved in the selected operation
- Unary messages: without arguments
  - E.g. theta sin
- Keyword messages: with one or more arguments
  - E.g. ages at: 'Brett Jorgensen' put: 3
- Binary messages: for arithmetic
  - E.g. 3 + 4

Sending a message (cont.)

– More examples:



Encapsulation

- An object encapsulates the details
  - Hide data (variables) and actions (methods) from others
- The clients of the object can use it without knowing how it is implemented

Encapsulation (cont.)

- Access modifiers are given to control the visibility
  - In languages like C++ and Java
  - private
    - *Only visible inside the class*
  - public
    - *Open to everyone*
  - protected
    - *Visible to the subclasses*

Classes and object instances

- A class describes the implementations of a set of objects
  - All represent the same kind of system component
  - Individual objects are called its instances
- The instances of a class are similar in both their public and private properties
  - Public properties: messages that make up its interface
  - Private properties: a set of instance variables and methods

Object creation

- How to create an object instance
  - Send the unary message “new” to the class
  - For example,

```
OrderedCollection new
```

returns a new collection that is an instance of the class OrderedCollection

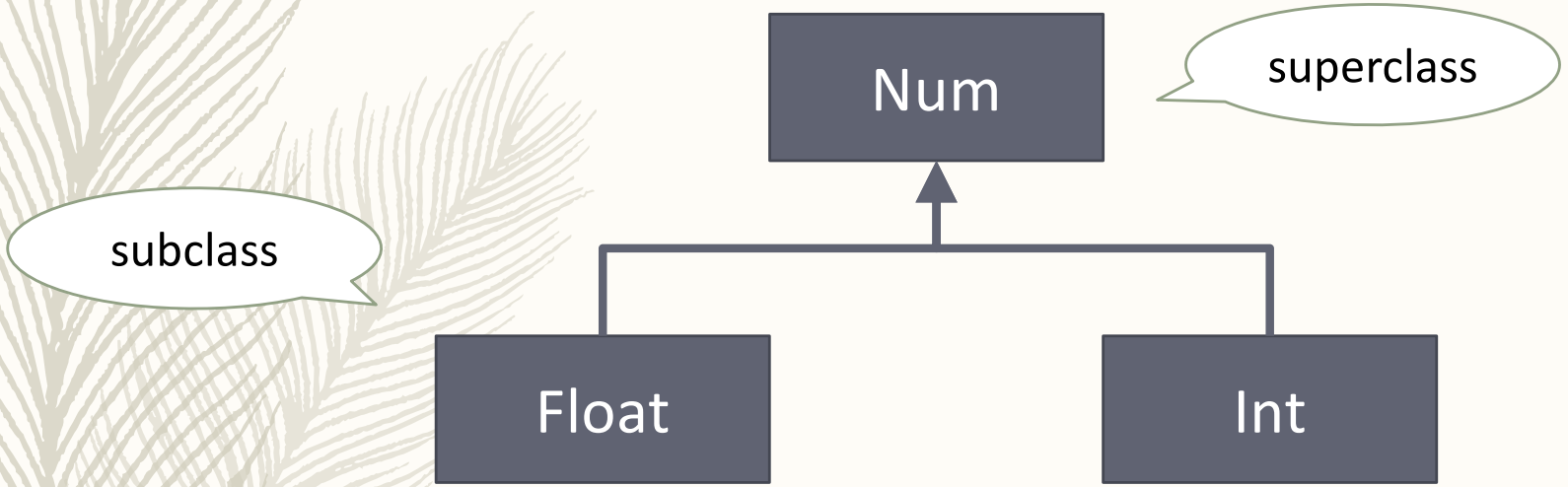
- Classes are also objects!

Subclasses

- Might want two objects to be substantially similar
  - But to differ in some particular way
- E.g. floating-point numbers and integers
  - Similar in ability to respond to arithmetic messages
  - Different in the way they represent numeric values
- E.g. ordered collections and bags
  - Similar in that they are containers
  - Different in how individual elements are accessed

A-kind-of; is-a relation

– Float is a kind of Num; Int is a kind of Num



– In Smalltalk-80, a system class named Object

– Every class will at least be a subclass of Object

Inheritance

- Subclasses can reuse the code in superclass
  - Subclassing is strictly hierarchical
  - Inheriting the behavior unless there is a redefinition
  - For example:

```
class Animal {
    int age;
    float weight;
    void hello() { print("..."); }
}
class Cat extends Animal {
    void hello() { print("meow~"); }
}
class Dog extends Animal {
    void hello() { print("bark!"); }
}
```

Inheritance (cont.)

- We only have to redefine the behaviors
  - E.g. hello()
- Or add additional things
  - E.g. dance() // just an example, don't take it seriously!
- For example:

```
class Animal {
    int age;
    float weight;
    void hello() { print("..."); }
}
class Cat extends Animal {
    void hello() { print("meow~"); }
    void sleep() { print("Zzz..."); }
    void dance() { print("dance~~"); }
}
class Dog extends Animal {
    void hello() { print("bark!"); }
}
```

Since it is a…

- Since Cat is an Animal
  - The pointer of Animal can point to a Cat
  - `Animal c = new Cat();`
- Since Integer is a Num
  - The pointer of Num can point to an Integer
  - `Num i = new Int();`

But it might not be…

- But a Num might not be an Integer
  - `Int i = new Num();  // !`
- But an Animal might not be a cat
  - `Cat c = new Animal();  // !`
- Need type conversion
  - Either implicitly according to the language design
  - Or explicitly by user (casting)
  - But might cause error at runtime
    - *Whether it can be converted or not is unknown until runtime*

Polymorphism in OOP

- The objects of subclasses can behavior differently
  - according to the definition of their subclasses

- For example,

```
Animal c = new Cat();
c.hello();     // meow~
Animal d = new Dog();
d.hello();     // bark!
```

Polymorphism in OOP (cont.)

- The term “polymorphism” used in OOP
  - actually refers to subtyping polymorphism
- Be polymorphic based on subtype
  - We will see other kinds of polymorphism later

Apparent type and actual type

- Apparent type
  - The type of the pointer
- Actual type
  - The type of the object that pointed by the pointer

- Suppose we have a pointer that point to Animal

```
Animal c = new Cat();
```

Apparent type: Animal
Actual type: Cat

How hello() is invoked?

- The method hello() is defined in the class Animal
- However, we need to invoke the proper implementation
  - According to its actual type
  - "meow~" for Cats, "bark!" for Dogs

Apparent type: Animal
Actual type: Cat

```
Animal c = new Cat();
c.hello();    // meow~
```

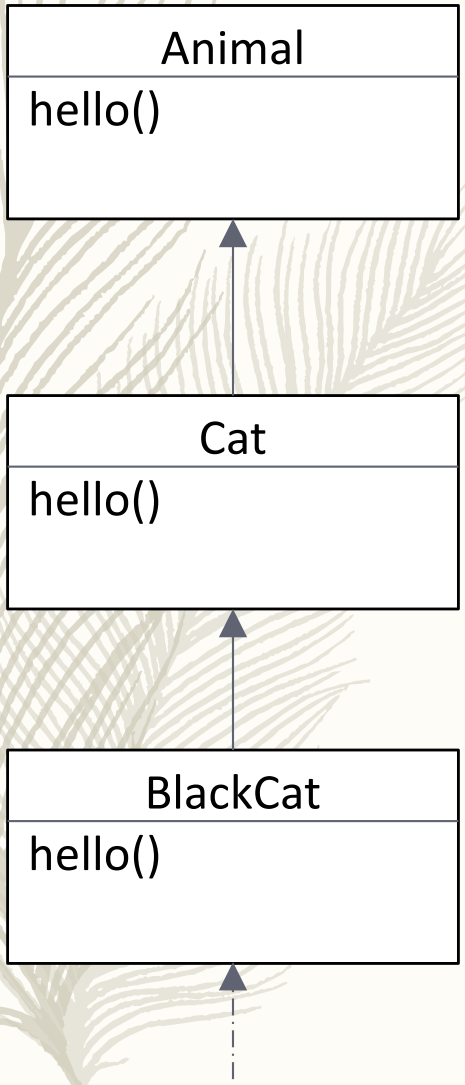
Apparent type: Animal
Actual type: Dog

```
Animal d = new Dog();
d.hello();    // bark!
```

Dynamic dispatch

- Dispatch the message to proper method implementation
  - According to the actual type at runtime
- The lookup of the implementation for a method call is performed at runtime
  - Since the actual type cannot be known until the message is sent
  - Flexible but expensive

Dynamic dispatch (cont.)



- `Animal c;`
  `:`
  `c.hello();`

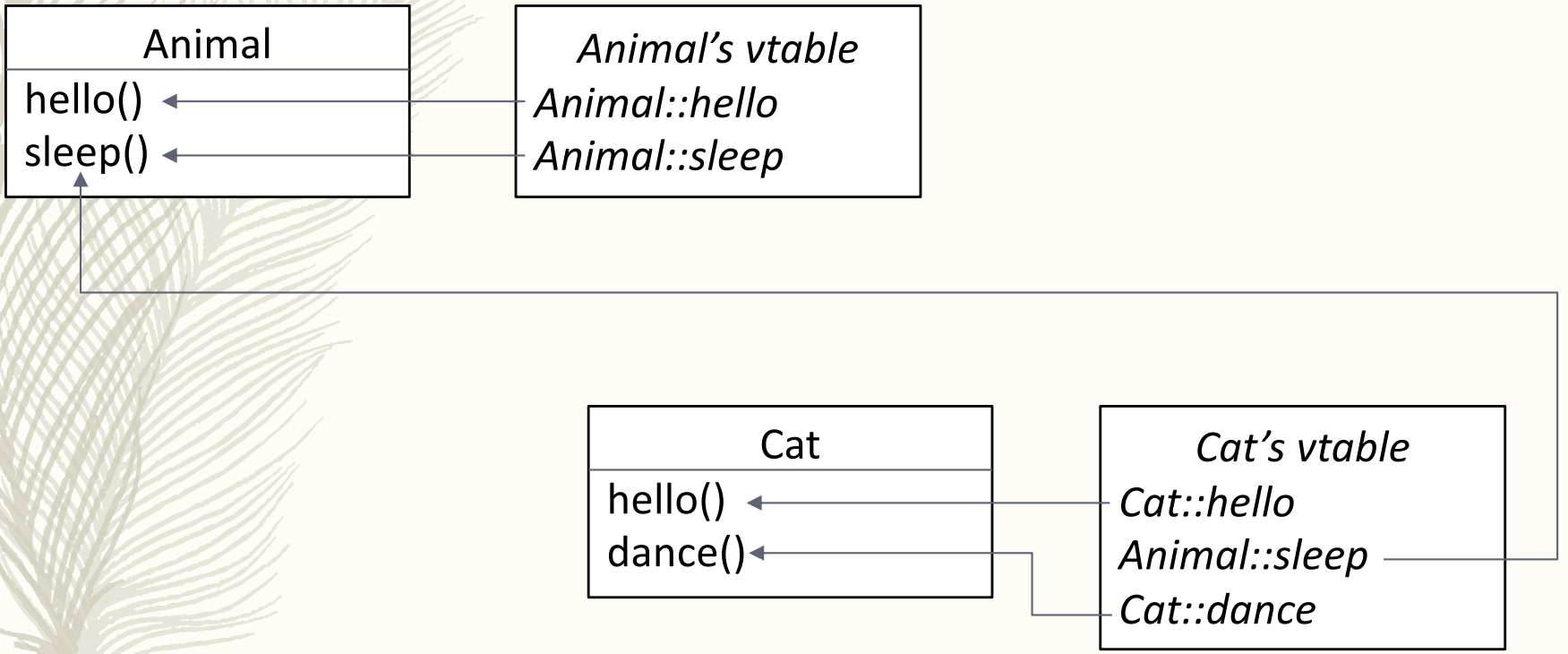
- How to find out the proper implementation for hello()?
  - Check the actual type
  - Check if the class have an implementation for it
    - *If YES, we found*
    - *If NO, check its superclass*
- The inheritance chain might be very long
  - The cost of lookup is expensive

When is it used?

- In C++
  - Default is static
  - But dynamic dispatch is used when using pointer to call the function f that is virtual in that class c
  - C* p;
    :
    p->f(); // f is declared as virtual in C
- In Java
  - Default is using dynamic dispatch
  - But is static if the method m is declared as final

Virtual table in C++

- One virtual table (vtable) per class
  - All object instances of the class share the same vtable
  - Each entry is a function pointer to the most derived function



The overheads of virtual table

- vtable is generated by compilers
  - For every class that uses virtual function
  - No need to perform the lookup at runtime
- Not as fast as non-virtual function
  - Use pointer to vtable
  - Use index to access the function pointer in the table
  - However, might be negligible in most cases

Inline Caching in Smalltalk

- The lookup of the implementations for a method is performed at runtime
  - Due to dynamic typing
  - Lack of type information at compile-time
- Since the type of the receiver at a given call site rarely varies
  - Next time the message might be sent to the receivers with the same type

Inline Caching in Smalltalk (cont.)

- Cache the result of lookup at the call site
  - Call site: the place calling the function (sending the message)
  - For a call site
  - Check the type of the receiver
  - Check whether we have the previous result of lookup
    - *Yes → simply use the result*
    - *No → perform the lookup and then cache it*

Language support for OOP

- Let us back to talk about the language support…

- In OO languages like C++ and Java we can simply use
  - Inheritance and dynamic dispatch
  - Since they are “built-in” features

→ Without language support?

Without language support for OOP

- Define classes and create objects by yourself
- Manage the class hierarchy by yourself
- Dynamically dispatch the calls by yourself

- How to?
  - For example, in C we have
    - *Struct*
    - *Function pointer*

Reference

- *Axel Schreiner. Object-Oriented Programming With ANSI-C.* https://www.cs.rit.edu/~ats/books/ooc.pdf

What does OO C look like?

- Object creation and destruction
  - Define types and their constructor and destructor

```
void* new(const void* type, …);
void delete(void* item);
```

- Inheritance
  - Let subclasses begin with their superclass

```
struct Point { … };
struct Circle { const struct Point _; int rad; };
```

An example: GTK+

- A open source multi-platform widget toolkit
  - Initially developed for and used by the GIMP
    - *The GIMP ToolKit*
    - *Now used by a large number of applications including GNOME*
  - Written in C with object-oriented programming

- GIMP
  - GNU Image manipulation program
  - A free, open source image editor

Reference

- http://www.gtk.org

GUI widgets toolkit

- Graphical user interface (GUI)
  - A kind of interface that allows users to interact with devices using graphical elements
  - Rather than command line (console user interface)
- GUI Widgets
  - The graphical elements
- GUI Widgets toolkit
  - The library providing graphical elements

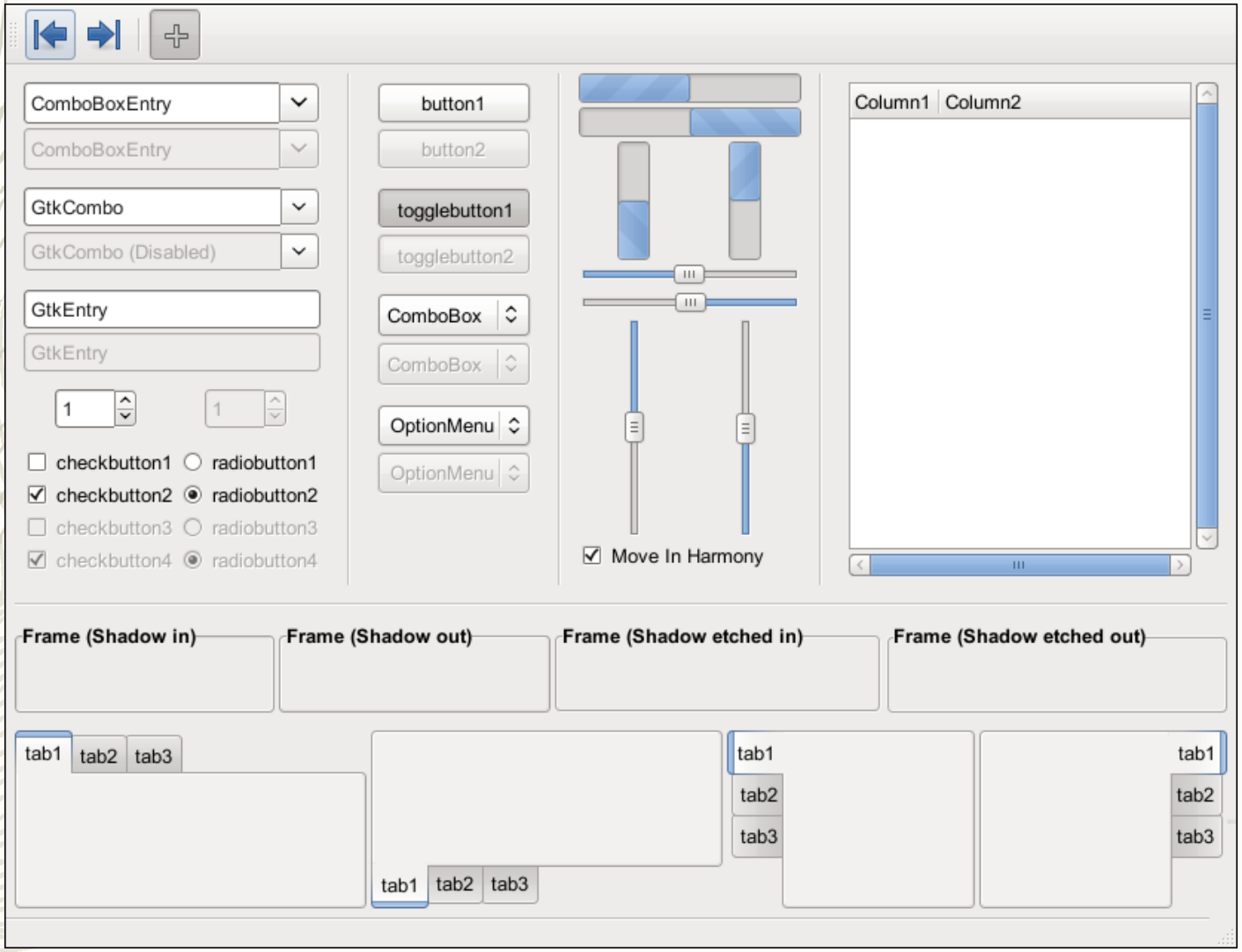
GUI widgets toolkit (cont.)

- Some famous toolkits you might hear
  - Motif, MFC, Cocoa
- AWT and Swing
  - Developed by Sun, for Java applications
- Standard Widget Toolkit (SWT)
  - Used in Eclipse, native looks for Java applications
- GTK+
  - Used in GNOME desktop environment
- Qt
  - Used in KDE desktop environment

GUI widgets toolkit (cont.)

- Every toolkit has its own design and implementation
  - Cross-platform or for a specific platform
  - Native looks or its own looks
  - Simple, fast or complex, sophisticated
  - Layout management, event propagation
- Some toolkit libraries have a lot of language bindings
  - Can be used in different languages
  - E.g. GTK+, Qt, Tk, wxWidgets, …
- Some toolkits libraries also provide data structure, threading, networking, etc.

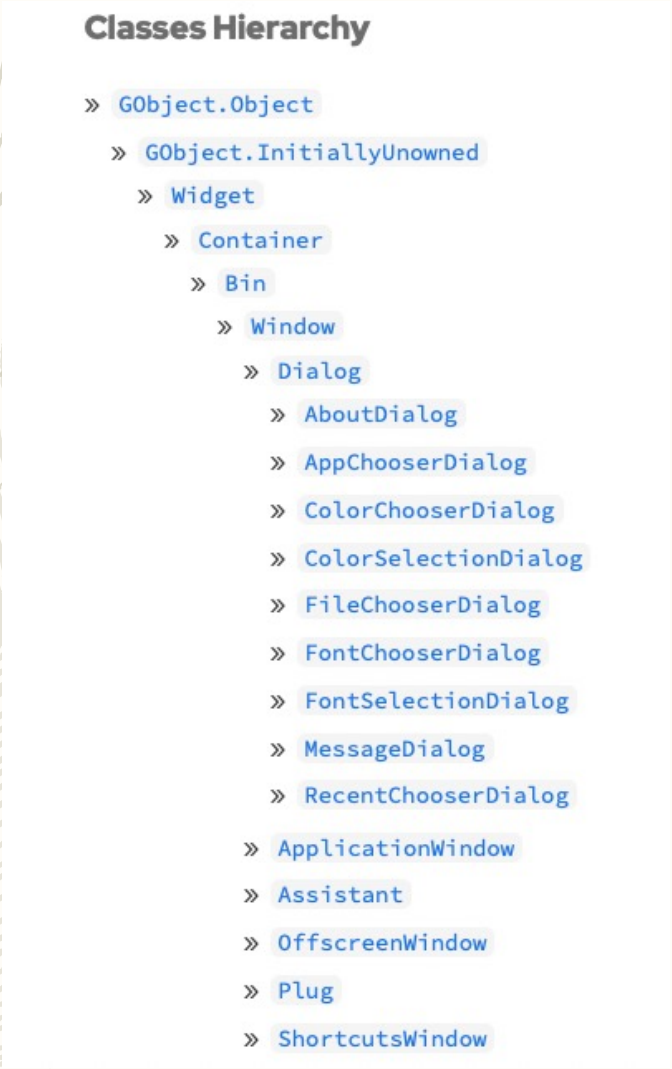
An example: GTK+ (cont.)

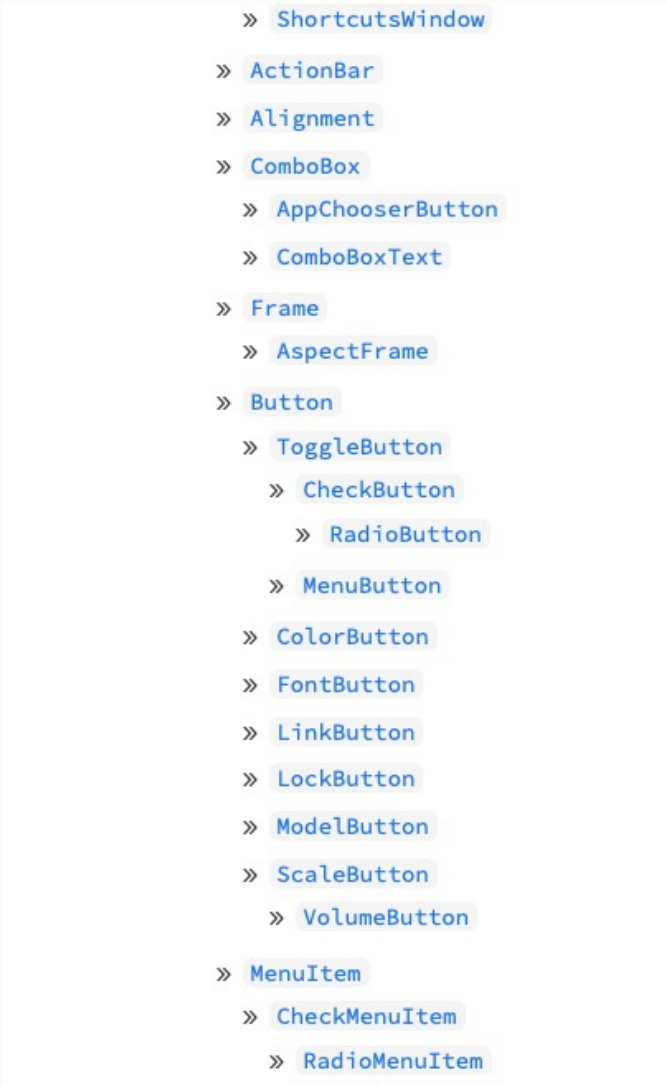


Reference

- https://docs.gtk.org/gtk4/visual_index.html

An example: GTK+ (cont.)





Reference

- https://docs.gtk.org/gtk3/classes_hierarchy.html

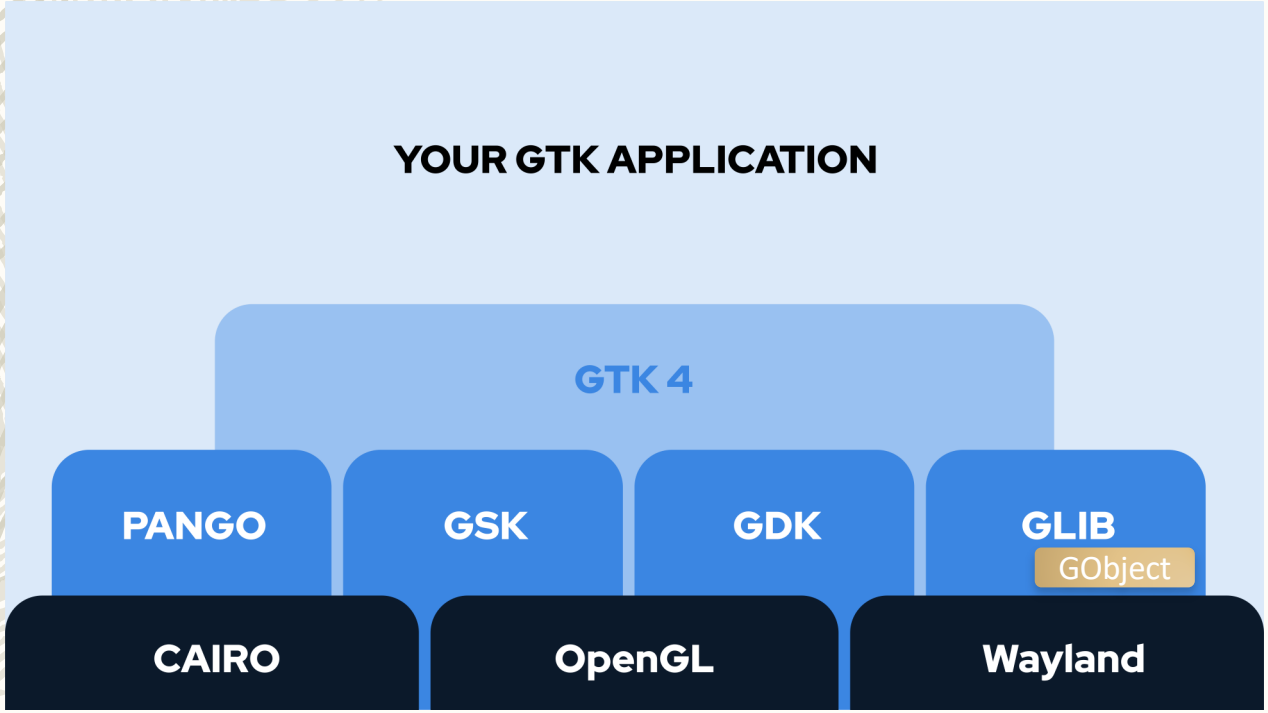
:

An example: GTK+ (cont.)

- Glib is a low-level core library including
  - An object system → GObject
  - Data structure handling
  - Event loop, Threads, etc.

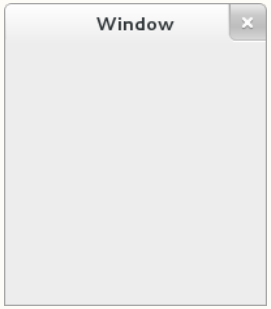
Reference

- https://www.gtk.org/docs/architecture/
- https://docs.gtk.org/gobject/



An example GTK program

– Create an empty window




```
#include <gtk/gtk.h>

static void activate (GtkApplication* app, gpointer user_data) {
  GtkWidget *window;
  window = gtk_application_window_new (app);
  gtk_window_set_title (GTK_WINDOW (window), "Window");
  gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
  gtk_widget_show_all (window);
}

int main (int argc, char **argv) {
  GtkApplication *app;
  int status;

  app = gtk_application_new ("org.gtk.example", G_APPLICATION_FLAGS_NONE);
  g_signal_connect (app, "activate", G_CALLBACK (activate), NULL);
  status = g_application_run (G_APPLICATION (app), argc, argv);
  g_object_unref (app);

  return status;
}
```

Reference

- https://developer.gnome.org/gtk3/stable/gtk-getting-started.html

Pure Object-Oriented? (cont.)

- Is the language designed for OO?
  - E.g. the ”methods” in OO C implementations are separate from objects (structs) → not OO
- Is everything an object?
  - E.g. the functions in C++ can be defined outside classes → not pure OO
- More strictly, include numbers and strings?
  - E.g. Java → not pure OO

Pure Object-Oriented? (again)

- How pure?
  - "Everything" should be an object!
- Including all definitions you might have known
  - Encapsulation, polymorphism, etc.

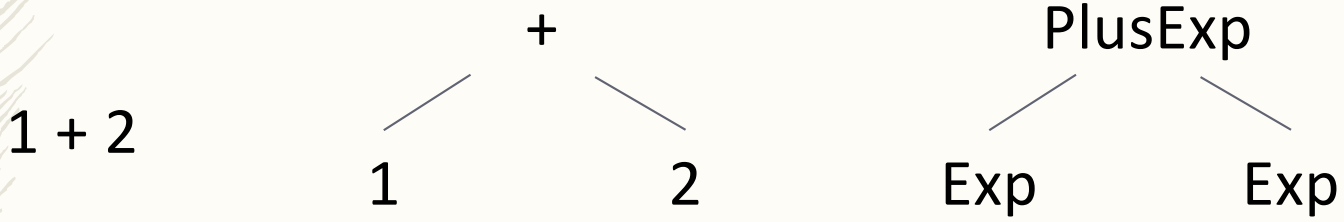
- All predefined types are objects
  - No primitive types
- All operations performed on objects are the methods on the objects
  - No System.out.println(), No math.pow()

Recall: AST in compilers

- Let parser produce concrete syntax tree
- But it still depends too much on the grammar

  → Abstract Syntax Tree

- How to represent and manipulate AST as data structures?



Reference

- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*

Use classes to implement AST

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- Here the inheritance in OOP is used
  - Both PlusExp and MinusExp inherit eval() from Exp
- Use the class Exp to represent expressions
  - PlusExp and MinusExp are kinds of Exp

Use classes to implement AST (cont.)

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- We can write an interpreter by building AST and then interpret it
  - Every class of AST nodes contains “eval” method
  - Return the value of the represented expression

Use classes to implement AST (cont.)

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- Looks good, but if we want to add another interpretation?
  - E.g. type check, optimization, etc.

→ Oops!
We need to add methods in every class every time…

Next lecture

10/7 **Practice 2**

10/14 Language Support