CS412/413
Introduction to Compilers
Radu Rugina

Lecture 14: Objects
22 Feb 06

Records
• Objects combine features of records and abstract data types
• Records = aggregate data structures
  – Combine several pieces of data into a higher-level structure
  – Their type is the cartesian product of element types
  – Need selection operator to access fields
  – Pascal records, C structures
• Example: struct {int x; float f; char a,b; } s;
  – Struct type: (s :.int, f:float, a:char, b:char)
  – Selection: s.x += i, s.x += s.y;

ADTs
• Abstract Data Types (ADT): separate implementation from specification
  – Specification: provide an abstract type for data
  – Implementation: must match abstract type
• Example: linked list

<table>
<thead>
<tr>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell = { int data; Cell next; }</td>
</tr>
<tr>
<td>List = { int len; Cell head, tail; }</td>
</tr>
<tr>
<td>int length() { return l.len; }</td>
</tr>
<tr>
<td>int first() { return head.data; }</td>
</tr>
<tr>
<td>int rest() { return head.next; }</td>
</tr>
<tr>
<td>List append(int d) { ... }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>int length();</td>
</tr>
<tr>
<td>List append(int d);</td>
</tr>
<tr>
<td>int first();</td>
</tr>
<tr>
<td>List rest();</td>
</tr>
</tbody>
</table>

Objects as Records
• Objects have fields
  – and methods = code that manipulates the data (fields) in the object
• Hence, objects combine data and computation

Objects as ADTs
• Specification: public methods and fields of the object
• Implementation: Source code for a class defines the concrete type (implementation)

<table>
<thead>
<tr>
<th>class List</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int len;</td>
</tr>
<tr>
<td>private Cell head, tail;</td>
</tr>
<tr>
<td>public static int length() {...};</td>
</tr>
<tr>
<td>public static List append(int d) {...};</td>
</tr>
<tr>
<td>public static int first() {...};</td>
</tr>
<tr>
<td>public static List rest() {...};</td>
</tr>
</tbody>
</table>

Objects
• What objects are:
  – Aggregate structures which combine data (fields) with computation (methods)
  – Fields have public/private qualifiers (can model ADTs)
• Need special support in many compilation stages:
  – Semantic analysis (type checking!)
  – Analysis and optimizations
  – Implementation, run-time support
• Features:
  – inheritance, subclassing, subtyping, dynamic dispatch
Inheritance

- Inheritance — mechanism which exposes common features of different objects
- Class B extends class A — “B has the features of A, plus some additional ones”, i.e., B inherits the features of A
  - B is subclass of A; and A is superclass of B

```java
class Point {
    float x, y;
    float getx();
    float gety();
}
class ColoredPoint extends Point {
    int color;
    int getcolor();
}
```

Single vs. Multiple Inheritance

- Single inheritance: inherit from at most one other object (Java)
- Multiple inheritance: may inherit from multiple objects (C++)

```java
class A {
    int a;
    int geta();
}
class B {
    int b;
    int getb();
}
class C : A, B {
    int c;
    int getc();
}
```

Inheritance and Scopes

- How do objects access fields and methods of:
  - Their own?
  - Their superclasses?
  - Other unrelated objects?
- Each class declaration introduces a scope
  - Contains declared fields and methods
  - Scopes of methods are sub-scopes
- Inheritance implies a hierarchy of class scopes
  - If B extends A, then scope of A is a parent scope for B

Example

```java
class A {
    int x;
    int f(int a) {
        int y; ...
    }
}
class B extends A {
    bool y;
    int z;
}
class C {
    & o;
    int a;
}
```

Class Scopes

- Resolve an identifier occurrence in a method:
  - Look for symbols starting with the symbol table of the current block in that method
- Resolve qualified accesses:
  - Accesses o.f, where o is an object of class A
  - Walk the symbol table hierarchy starting with the symbol table of class A and look for identifier f
  - Special keyword this refers to the current object, start with the symbol table of the enclosing class

Class Scopes

- Multiple inheritance:
  - A class scope has multiple parent scopes
  - Which should we search first?
  - Problem: may find symbol in both parent scopes!
- Overriding fields:
  - Fields defined in a class and in a subclass
  - Inner declaration shadows outer declaration
  - Symbol present in multiple scopes
Inheritance and Typing

- Classes have types
  - Type is cartesian product of field and method types
  - Type name is the class name
- What is the relation between types of parent and inherited objects?
- Subtyping: if class B extends A then
  - Type B is a subtype of A
  - Type A is a supertype B
- Notation: $B \leq A$

Subtype $\Rightarrow$ Subset

“A value of type S may be used wherever a value of type T is expected”

$S \leq T \Rightarrow \text{values}(S) \subseteq \text{values}(T)$

Subtype Properties

- If type S is a subtype of type T ($S \leq T$), then:
  A value of type S may be used wherever a value of type T is expected (e.g., assignment to a variable, passed as argument, returned from method)

  ```
  Point x;
  ColoredPoint y;
  x = y;
  ```

- Polymorphism: a value is usable at several types
- Subtype polymorphism: code using T’s can also use S’s; S objects can be used as S’s or T’s.

Implications of Subtyping

- We don’t actually know statically the types of objects
  - Can be the declared class or any subclasses
  - Precise types of objects known only at run-time
- Problem: overridden fields / methods
  - Declared in multiple classes in the hierarchy
  - We don’t know statically which declaration to use at compile time

Virtual Functions

- Virtual functions $\Rightarrow$ methods overridden by subclasses
  - Subclasses define specialized versions of the methods

  ```
  class List {
    List next;
    int length() { ... }
  }

  class LenList extends List {
    int n;
    int length() { return n; }
  }
  ```

Virtual Functions

- We don’t know what code to run at compile time

  ```
  List a;
  if (cond) { a = new List(); }
  else { a = new LenList(); }
  a.length()
  ```

  $\Rightarrow$ List.length() or LenList.length()?

- Solution: method invocations resolved dynamically
- Dynamic dispatch: run-time mechanism to select the appropriate method, depending on the object type
**Objects and Typing**

- Objects have types
  - ... but also have implementation code for methods

- ADT perspective:
  - Specification = typing
  - Implementation = method code, private fields
  - Objects mix specification with implementation

- Can we separate types from implementation?

**Interfaces**

- Interfaces are pure types; they don’t give any implementation

```java
interface List {
    int length();
    List append(int);  
    int first();
    List rest();
}
```

```java
class MyList implements List {
    private int len;
    private Cell head, tail;
    public int length() {
        return len;
    }
    public List append(int d) {
        ...;
    }
    public List rest() {
        ...;
    }
}
```

**Multiple Implementations**

- Interfaces allow multiple implementations

```java
interface List {
    int length();
    List append(int);  
    int first();
    List rest();
}
```

```java
class SimpleList impl. List {
    private int data;
    public int length() {
        return 1+next.length()
    }
}
```

```java
class MyList implements List {
    private int len;
    private Cell head, tail;
    public int length() {
        return len;
    }
    public List append(int d) {
        ...;
    }
    public List rest() {
        ...;
    }
}
```

**Subtyping vs. Subclassing**

- Can use inheritance for interfaces
  - Build a hierarchy of interfaces
    ```java
    interface A (...)  
    interface B extends A (...)  
    class C implements A (...)  
    ```
  - Objects can implement interfaces
    ```java
    class C implements A (...)  
    ```

- Subtyping: interface inheritance
  - Subclassing: object (class) inheritance
  - Subclassing implies subtyping

**Abstract Classes**

- Classes define types and some values (methods)
- Interfaces are pure object types

- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class

**Subtypes in Java**

```java
interface I, J extends I {}
interface K extends I {}
class C extends K {
    I i;
    J j;
}
```

```java
class C1 extends C2 {
    I i1;
    J j1;
    K k1;
}
```
Subtyping Properties

- Subtype relation is reflexive: $T \leq T$
- Transitive: $R \leq S$ and $S \leq T$ implies $R \leq T$
- Anti-symmetric: $T_1 \leq T_2$ and $T_2 \leq T_1$ implies $T_1 = T_2$
- Defines a partial ordering on types!
- Use diagrams to describe typing relations

Subtype Hierarchy

- Introduction of subtype relation creates a hierarchy of types: subtype hierarchy

\[
\begin{array}{c}
\text{type or subtype hierarchy} \\
\{ \text{C1, C2, C3, C4, C5} \} \\
\text{class/inheritance hierarchy}
\end{array}
\]