301AA - Advanced Programming

Lecturer: Andrea Corradini

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

Course pages:

http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-13: Polymorphisms
Outline

• Polymorphism: a classification
• Overloading
• Coercion
• Inclusion polymorphism
• Overriding
Polymorphism

- From Greek: πολυμορφος, composed of πολυ (many) and μορφή (form), thus “having several forms”
- “Forms” are types
- “Polymorphic” are function names (also operators, methods, …)
- “Polymorphic” can also be types (parametric data types, type constructors, generics, …)
  - Usually as encapsulation of several related function names
Flavors of polymorphism

- Ad hoc
- Bounded
- Contravariant
- Covariant
- Inclusion
- Invariant
- Parametric
- Universal
- ...

Related concepts:
- Coercion
- Generics
- Inheritance
- Macros
- Overloading
- Overriding
- Subtyping
- Templates
- ...

• Ad hoc
• Bounded
• Contravariant
• Covariant
• Inclusion
• Invariant
• Parametric
• Universal
• ...

4
Universal vs. ad hoc polymorphism

- With **ad hoc** polymorphism the same function name denotes different algorithms, determined by the actual types
- With **universal** polymorphism there is only one algorithm: a single (universal) solution applies to different objects
- Ad hoc and universal polymorphism can coexist
Binding time

• The binding of the function name with the actual code to execute can be
  – at compile time – early, static binding
  – at linking time
  – at execution time – late, dynamic binding
• If it spans over different phases, the binding time is the last one.
• The earlier the better, for debugging reasons.
Classification of Polymorphism

- Polymorphism
  - Universal
  - Ad hoc
  - Overloading
  - Coercion
    - Implicit
    - Explicit
      - Bounded
        - Covariant
        - Invariant
        - Contravariant
  - Inclusion
  - Overriding
Ad hoc polymorphism: overloading

• Present in all languages, at least for built-in arithmetic operators: +, *, -, ...
• Sometimes supported for user defined functions (Java, C++, ...) 
• C++, Haskell allow overloading of primitive operators 
• The code to execute is determined by the type of the arguments, thus
  – early binding in statically typed languages
  – late binding in dynamically typed languages
Overloading: an example

• Function for squaring a number:
  \[
  \text{ sqr}(x) \{ \text{ return } x \times x; \}
  \]

• Typed version (like in C):
  \[
  \text{ int } \text{sqr}(\text{int } x) \{ \text{return } x \times x; \}
  \]

• Multiple versions for different types:
  \[
  \text{ int } \text{sqrInt}(\text{int } x) \{ \text{return } x \times x; \}
  \]
  \[
  \text{ double } \text{sqrDouble}(\text{double } x) \{ \text{return } x \times x; \}
  \]

• Overloading (Java, C++):
  \[
  \text{ int } \text{sqr}(\text{int } x) \{ \text{return } x \times x; \}
  \]
  \[
  \text{ double } \text{sqr}(\text{double } x) \{ \text{return } x \times x; \}
  \]
Overloading in Haskell

- Haskell introduces **type classes** for handling overloading in presence of type inference
- Very nice and clean solution, unlike most programming languages
- We shall present this later in the course
Universal polymorphism: Coercion

• **Coercion**: automatic conversion of an object to a different type

• Opposed to **casting**, which is explicit

```java
double sqrt(double x){...}
double d = sqrt(5) // applied to int
```

• Thus the same code is applied to arguments of different types

• Degenerate (and uninteresting) case of polymorphism
Classification of Polymorphism

Polymorphism

- Universal
- Ad hoc

Coercion:
- Implicit
- Explicit
  - Bounded
    - Covariant
    - Invariant
    - Contravariant

Inclusion

Overriding

Overloading
Inclusion polymorphism

• Also known as subtyping polymorphism, or just inheritance

• Polymorphism ensured by (Barbara Liskov’) **Substitution principle**: an object of a subtype (subclass) can be used in any context where an object of the supertype (superclass) is expected

• [Java, C++,...] methods/functions with a formal parameter of type T accept an actual parameter of type S <: T (S subtype of T).

• Methods/virtual functions declared in a class can be invoked on objects of subclasses, if not redefined...
Overriding

• [Java] A method m(…) of a class A can be redefined in a subclass B of A.

• **Dynamic binding:**

  ```java
  A a = new B(); // legal
  a.m(…) // overridden method in B is invoked
  ```

• Overriding introduces ad hoc polymorphism in the universal polymorphism of inheritance

• Resolved at runtime by the lookup done by the `invokevirtual` operation of the JVM
Overloading + Overriding: C++ vs Java

class A {
public:
    virtual void onFoo() {}
    virtual void onFoo(int i) {}
};
class B : public A {
public:
    virtual void onFoo(int i) {}
};
class C : public B {
};

int main() {
    C* c = new C();
    c->onFoo();
    //Compile error –
    // doesn't exist
}

class A {
    public void onFoo() {}
    public void onFoo(int i) {}
}
class B extends A {
    public void onFoo(int i) {}
}
class C extends B {
}
class D {
public static void main(String[] s) {
    C c = new C();
    c.onFoo();
    //Compiles !!
}
}
Overriding + Overloading

- **[Java]** Overloading is type-checked by the compiler
- **Overriding** resolved at runtime by the lookup done by `invokevirtual`
- **[C++]** Dynamic method dispatch: C++ adds a v-table to each object from a class having virtual methods
- The compiler does not see any declaration of `onFoo` in C, so it continues upwards in the hierarchy. When it checks B, it finds a declaration of `void onFoo(int i)`, so it stops lookup and tries overload resolution, but it fails due to the inconsistency in the arguments.
- `void onFoo(int i)` **hides** the definitions of onFoo in the superclass.
- Solution: add `using A::onFoo;` to class B