

EECS 490 – Lecture 18

Generics and Modules

1

11/15/17

Announcements


- ▶ HW4 due Tue 11/14 at 8pm
- ▶ Project 4 due Tue 11/21 at 8pm
- ▶ Midterm regrade requests due today at 8pm

Parametric Polymorphism

- Subtype polymorphism relies on subtype relationships and dynamic binding to enable polymorphic code
- *Parametric polymorphism*, on the other hand, allows code to operate on different types without requiring any subtype relationships
- The compiler *instantiates* a polymorphic piece of code to work with the actual types with which it is used
- Examples: C++ templates and Java generics

Implicit Parametric Polymorphism

- Some languages, particularly those in the ML family, allow types to be elided from a function, in which case it is implicitly polymorphic
- The compiler infers the type for each use
- Example in OCaml:



```
let max x y =
  if x > y then
    x
  else
    y;;
```

```
# max 3 4;;
- : int = 4
# max 4.1 3.1;;
- : float = 4.1
# max "Hello" "World";;
- : string = "World"
```

'a → 'a → 'a



Explicit Parametric Polymorphism

- In other languages, an entity must be explicitly marked as polymorphic
- Example in C++:

class

```
template <typename T>
T max(const T &x, const T &y) {
    return x > y ? x : y;
}
```

max<int>(3, 4);

Type inference
determines
type to use in
instantiation

```
max(3, 4); // returns 4
max(4.1, 3.1); // returns 4.1
max("Hello"s, "World"s) // returns "World"s
max(3, 4.1); // error
```

Conflicting
argument types

C++14 std::string
literal

max<double>(3, 4.1);

Multiple Type Parameters

- We can use multiple type parameters to handle arguments of different type
- We need to use type inference in order to determine the return type

```
template <typename T, typename U>  
→ auto max(const T &x, const U &y) ->  
→ decltype(x > y ? x : y) {  
    return x > y ? x : y;  
}
```

Trailing return
type can be
elided in
C++14

Will be computed as
double in this case

```
max(3, 4.1); // returns 4.1
```

Non-Type Parameters

- In a few languages, a parameter to a generic may be a value rather than a type
- In C++, the parameter can be of integral, enumeration, reference, pointer, or pointer-to-member type
- Example (similar to `std::array`):

```
template <typename T, int N>  
class array;
```

```
array<double, 5> arr;  
arr[3] = 4.1;
```

Handwritten red annotations illustrating the non-type parameter `N`:

- A red arrow points from the `N` in the template parameter list to the `N` in the array definition.
- A red arrow points from the `N` in the array definition to the `N` in the array access.
- The handwritten text shows the transformation of the array definition from `T[N] elements;` to `T elements[N];`, with the `N` in the second version underlined.

Implicit and Explicit Constraints

- A generic entity usually does not work on all possible types
 - Example: calling `max()` on streams, Ducks, etc.
- In some languages, the constraints on a generic are implicit
 - The compiler will attempt to instantiate the generic and then report a failure
- In other languages, constraints can be specified explicitly
 - The generic is then checked once for validity
 - Upon instantiation, only the type argument needs to be checked against the explicit constraints

Implicit Constraints in C++

- Example:

```
max(cin, cin);
```

```
foo.cpp:7:12: error: invalid operands to binary expression
      ('const std::__1::basic_istream<char>' and
       'const std::__1::basic_istream<char>')
      return x > y ? x : y;
             ~ ^ ~
foo.cpp:11:5: note: in instantiation of function template
      specialization 'max<std::__1::basic_istream<char> >'
      requested here
      max(cin, cin);
      ^  ~  ~  ~
[long list of overloads of > that were not viable]
```

- Inscrutable error messages are a side effect of waiting to check until instantiation

Implementation Strategies

- Two general strategies:
 - Produce a single copy of generic code for all instantiations
 - Produce a separate copy for each instantiation
- Languages with strong support for dynamic binding often only generate a single copy
 - Smaller code size
 - Not able to specialize code based on the type
- Other languages generate a specialized copy for each instantiation
 - Larger code size
 - Compiler needs full access to generic when instantiating it

Generics in Java

- Similar syntax as in C++
- Example of using a generic:

```
ArrayList<String> strings =  
    new ArrayList<String>();  
strings.add("Hello");  
strings.add("World");  
System.out.println(strings.get(1));
```



Prints World

Generic Types

- Defining a basic generic type has similar syntax to C++, but without the template header

```
class Foo<T> {  
    private T x;  
    public Foo(T x_in) {  
        x = x_in;  
    }  
    public T get() {  
        return x;  
    }  
}
```

Type parameters
go here

Can use type
parameters within
the generic class

new T(); new T[10];

Foo<String> f = new Foo<String>("Hello");
System.out.println(f.get());

Generic Functions

- In a generic function, the type parameter must be specified before the return type, since the return type may use it

↓ **Type parameters go here**

```
static <T> T max(T x, T y) {  
    return x.compareTo(y) > 0 ? x : y;  
}
```

**This will not compile;
not all objects have a
compareTo() method**

Constraints

- We can specify constraints on a type parameter to ensure that it supports the required set of operations

Built-in interface for objects that support comparisons

```
interface Comparable<T> {  
    int compareTo(T other);  
}
```

Require that the type argument supports comparisons to the same type

```
static <T extends Comparable<T>> T max(T x, T y) {  
    return x.compareTo(y) > 0 ? x : y;  
}
```

Type inference determines type to use in instantiation

```
System.out.println(max("Hello", "World"));
```

Modifying Foo

- We can modify Foo to implement the Comparable interface:

```
class Foo<T> implements Comparable<Foo<T>> {  
    private T x;  
    public Foo(T x_in) {  
        x = x_in;  
    }  
    public T get() {  
        return x;  
    }  
    public int compareTo(Foo<T> other) {  
        return x.compareTo(other.x);  
    }  
}
```

The diagram illustrates a code modification and a resulting compilation error. A red arrow points from the text 'We can modify Foo to implement the Comparable interface:' to the `Comparable<Foo<T>>` interface in the code. A blue box labeled 'Implement compareTo() method' has a blue arrow pointing to the `compareTo` method in the code. A red box labeled 'This will not compile; not all objects have a compareTo() method' has red arrows pointing to the `x.compareTo(other.x)` call in the `compareTo` method, indicating that the `compareTo` method is not guaranteed to exist on the `T` type.

Adding a Type Constraint

- We can add a type constraint to Foo itself:

```
class Foo<T> extends Comparable<T> {  
    implements Comparable<Foo<T>> {  
        private T x;  
        public Foo(T x_in) {  
            x = x_in;  
        }  
        public T get() {  
            return x;  
        }  
        public int compareTo(Foo<T> other) {  
            return x.compareTo(other.x);  
        }  
    }  
}
```

Type argument
must be
comparable to
itself

Prints World

```
Foo<String> f1 = new Foo<String>("Hello");  
Foo<String> f2 = new Foo<String>("World");  
System.out.println(max(f1, f2).get());
```


Rectangles

- Consider the following classes:

```
class Rectangle
    implements Comparable<Rectangle> {
    private int side1, side2;
    public Rectangle(int s1_in, int s2_in) {
        side1 = s1_in;
        side2 = s2_in;
    }
    public int area() {
        return side1 * side2;
    }
    public int compareTo(Rectangle other) {
        return area() - other.area();
    }
}
```

This works

```
Foo<Rectangle> f1 = new Foo<Rectangle>(3, 4);
```

Squares

- Now consider the following derived class:

```
class Square extends Rectangle {  
    public Square(int side) {  
        super(side, side);  
    }  
}
```

This fails

```
Foo<Square> f1 = new Foo<Square>(3, 4);
```

- Clearly a Square is comparable to another Square, since it can be compared to any Rectangle
- This fails because Square does not satisfy `Comparable<Square>`
- It derives from `Comparable<Rectangle>`, which is more general

Loosening the Constraint

- We can loosen the constraint as follows:

```
class Foo<T extends Comparable<? super T>>
    implements Comparable<Foo<T>> {
    ...
}
```

Allow T to
implement
Comparable<U>,
where U is a
supertype of T

```
public static void main(String[] args) {
    Foo<Square> f1 =
        new Foo<Square>(new Square(3));
    Foo<Square> f2 =
        new Foo<Square>(new Square(4));
    System.out.println(f1.compareTo(f2));
}
```

Prints -7

20

- ▶ We'll start again in five minutes.

Modules

- An ADT defines an abstraction for a single type
- A *module* is an abstraction for a collection of types, variables, functions, etc.
- Often, a module defines a scope for the names contained within the module
- Examples:
 - `math` module in Python
 - `java.util` package in Java
 - `<string>` header in C++

Translation Units

- ▶ A *translation or compilation unit* is the unit of compilation in languages that support separate compilation
- ▶ Often consists of a single source file
- ▶ In C and C++, consists of a source file along with the files that it recursively `#includes`
- ▶ A translation unit only needs to know basic information about the entities in other translation units in order to be compiled
 - ▶ Example: names and types of variables, return type, name, and parameter types of functions, members of a class

Headers

- In some languages, the public interface of a module is located in a header file, which is then included in other translation units

```
class Triangle {  
    double a, b, c;  
public:  
    Triangle();  
    Triangle(double, double, double);  
    double area() const;  
    double perimeter() const;  
    void scale(double s);  
};
```

Triangle.h

```
#include "Triangle.h"  
  
Triangle::Triangle(double a_in,  
    double b_in, double c_in)  
    : a(a_in), b(b_in), c(c_in) { }  
  
double Triangle::area() const {  
    return a * b * c;  
}
```

Triangle.cpp

- In other languages, all the code for a module is located in a single file, and the compiler extracts the public interface needed by other translation units

Python Modules

- ▶ A Python source file is called a *module*
 - ▶ First unit of organization for interrelated entities
- ▶ A module is associated with a scope containing the names defined within it
- ▶ Names can be *imported* from another module

```
from math import sqrt
```

Import single name
from a module

```
def quadratic_formula(a, b, c):  
    return (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

```
def main():
```

```
    import sys
```

```
    print(quadratic_formula(int(sys.argv[1]),  
                             int(sys.argv[2]),  
                             int(sys.argv[3])))
```

```
if __name__ == '__main__':  
    main()
```

Import the name
of a ~~package~~ *module*
into local scope

Use package name

Python Packages

- Python packages are a second level of organization, consisting of multiple modules in the same directory
- Packages can be nested

sound/

__init__.py

formats/

**Denotes a
package**

__init__.py
wavread.py
wavwrite.py
aiffread.py

...

effects/

__init__.py
echo.py
surround.py
reverse.py
...

Top-level package

Initialize the sound package

Subpackage for file format conversions

Subpackage for sound effects

Java Packages

- Java follows a similar organizational scheme, with the first unit a class and the second unit a package
- Multiple classes can be placed in the same file, but if a class is used outside of its file, it should be in its own file
- Packages can be nested, as in Python
- Example:

```
package formulas  
  
public class Quadratic {  
    ...  
}
```

**Specifies that this
file is part of the
formulas package**

Java Imports

- ▶ A package can be used without an import

```
java.util.Vector vec = new java.util.Vector();
```

- ▶ A single member from a package, or all its members, can be imported

```
import java.util.Vector; // import just one member
import java.util.*;      // import all members
```

- ▶ Static methods and constants can be imported from a class

```
import static java.lang.System.out;
...
out.println("Hello world!");
```

Namespaces in C++

- A *namespace* defines a scope for names

```
→ namespace foo {  
    struct A {};  
    int x; ↑  
→ } -
```

Can have multiple namespace blocks in the same or different files

```
⚡ namespace foo {  
    struct B : A {};  
}
```

Can use a name from the same namespace without qualification

Use scope-resolution operator to access a name

```
foo::A *a = new foo::A();
```

```
using foo::A;
```

Import a single name

```
using namespace foo;
```

Import all names

Global Namespace

- An entity defined outside of a namespace is actually part of the global namespace

```
int bar();
```

```
void baz() {  
    std::cout << ::bar() << std::endl;  
}
```

**Qualified access to
global namespace**

- Java similarly places code without a package declaration into the anonymous package

Internal Linkage

- C does not have namespaces, so it uses *linkage* specifiers to avoid name conflicts between translation units
 - Also in C++, since it's mostly compatible with C
- A variable or function at global scope can be declared `static`, which specifies *internal linkage*
 - Name will not be visible outside of translation unit, avoiding name conflicts at link stage
- Variables and functions defined, not just declared, in a header should generally be `static`

```
static const double PI = 3.1415926535;  
static double area(double r) {  
    return PI * r * r;  
}
```

External Linkage

- ▶ A global variable or function has *external linkage* if it does not have the `static` specifier
 - ▶ The name will be accessible from other translation units
- ▶ An entity with external linkage must have exactly one definition among the translation units of a program
- ▶ A function can be declared but not defined by leaving out the function body
- ▶ A variable declaration is also a definition, unless it has the `extern` specifier

```
extern int count;  
int count;
```

Just a declaration

A definition that default initializes count

Information Hiding

- ▶ Languages often support information hiding at the granularity of a module or package
- ▶ In Java, a non-`public` class is available only to the same package
- ▶ Java and C# have module or package-level access modifiers for class members
- ▶ In C and C++, entities declared with internal linkage in a `.c` or `.cpp` file are not available to other translation units

Opaque Types in C

- In C, struct members can be hidden by providing only a declaration and not the definition of a struct in the header file

```
typedef struct list *stack;  
stack stack_make();  
void stack_push(stack s, int i);  
int stack_top(stack s);  
void stack_pop(stack s);  
void stack_free(stack s);
```

- Other translation units can make use of the interface, but cannot access members or even directly create an object of an opaque type

```
stack s = stack_make();  
stack_push(s, 3);  
stack_pop(s);
```

Initialization

- ▶ Languages specify semantics for initialization of the contents of a class, module, or package
- ▶ In Java, a class is initialized the first time it is used
 - ▶ Generally when an instance is created or a static member is accessed for the first time
- ▶ In Python, a module's code is executed when it is imported
 - ▶ If a module is imported again from the same module, its code does not execute again

Circular Dependencies

- Circular dependencies between modules should be avoided
- Can require restructuring code
- Example:

```
> python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 1, in <module>
    import bar
  File "bar.py", line 1, in <module>
    import foo
  File "foo.py", line 9, in <module>
    print(func1())
  File "foo.py", line 4, in func1
    return bar.func3()
AttributeError: module 'bar' has no
attribute 'func3'
```

```
import bar
def func1():
    return bar.func3()

def func2():
    return 2

print(func1())
```

```
import foo
def func3():
    return foo.func2()
```

Initialization in C++

- C++ has a multi-step initialization process
 1. **Static initialization:** initialize compile-time constants to their values, and all other variables with static storage duration to zero
 2. **Dynamic initialization:** initialize static-storage variables using their specified initializers
 - Can be delayed until first use of the translation unit
- Within a translation unit, initialization is in program order, with some exceptions
- Order is undefined between translation units
 - Cannot rely on another translation unit being initialized first