

# **CSC 447 - Concepts of Programming Languages**

## **Inheritance and Dynamic Dispatch**

Instructor: Stefan Mitsch



## Learning Objectives

- ❓ How do we support code sharing between classes?
- ❓ What should happen when methods have the same name?
  - Identify classes and inheritance
  - Describe the difference between static and dynamic dispatch

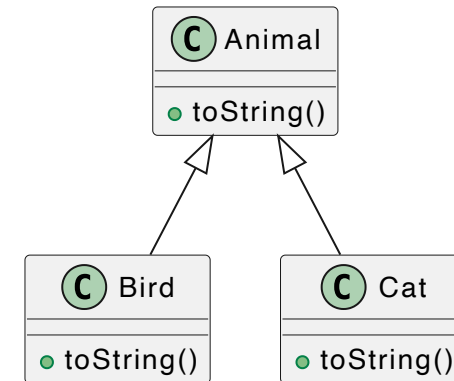


# Scala: Which `toString`?

? What is the type of `x`? Which `toString` is invoked?

```
1 class Animal { override def toString () = "Animal" }
2 class Bird extends Animal { override def toString () = "Bird" }
3 class Cat extends Animal { override def toString () = "Cat" }
4
5 val xs = Array[Animal] (new Bird(), new Cat ())
6 for (x <- xs) println (x.toString())
```

- `x` has
  - static type `Animal`
  - dynamic type `Bird / Cat`
- Dynamic dispatch, late binding: use dynamic type (of object)



- Output at runtime

```
1 Bird
2 Cat
```



# Java: Which `toString`?

? What is the type of `x`? Which `toString` is invoked?

```
1 class Animal { public String toString () { return "Animal"; } }
2 class Bird extends Animal { public String toString () { return "Bird"; } }
3 class Cat extends Animal { public String toString () { return "Cat"; } }
4
5 Animal[] xs = { new Bird(), new Cat () };
6 for (Animal x : xs) System.out.println (x.toString());
```

- `x` has
  - static type `Animal`
  - dynamic type `Bird / Cat`
- Dynamic dispatch, late binding: use dynamic type (of object)

- Output at runtime

```
1 Bird
2 Cat
```



# C++: Which toString?

? What is the type of `x`? Which `toString` is invoked?

```
1 class Animal { public: string to_string() { return "Animal"; } };
2 class Bird: public Animal { public: string to_string() { return "Bird"; } };
3 class Cat: public Animal { public: string to_string() { return "Cat"; } };
4
5 Animal *xs[] = { new Bird(), new Cat() };
6 for (Animal *x : xs) cout << x->to_string() << endl;
```

- `x` has
  - static type `Animal`
  - dynamic type `Bird / Cat`
- Static dispatch, early binding: use static type (of variable)

- Output at runtime

```
1 Animal
2 Animal
```



# C++: Which toString?

? What is the type of `x`? Which `toString` is invoked?

```
1 class Animal { public: virtual string to_string() { return "Animal"; }
2 class Bird: public Animal { public: virtual string to_string() { return "Bird"; }
3 class Cat: public Animal { public: virtual string to_string() { return "Cat"; }
4
5 Animal *xs[] = { new Bird(), new Cat() };
6 for (Animal *x : xs) cout << x->to_string() << endl;
```

- `x` has
  - static type `Animal`
  - dynamic type `Bird / Cat`
- C++ dynamically dispatches `virtual` methods **only when** object accessed with a pointer/reference

• Output at runtime

```
1 Bird
2 Cat
```



# C++: Which toString?

? What is the type of `x`? Which `toString` is invoked?

```
1 class Animal { public: virtual string to_string() { return "Animal"; }
2 class Bird: public Animal { public: virtual string to_string() { return "Bird"; }
3 class Cat: public Animal { public: virtual string to_string() { return "Cat"; }
4
5 Animal xs[] = { Bird(), Cat() };
6 for (Animal x : xs) cout << x.to_string() << endl;
```

- `x` has
  - static type `Animal`
  - dynamic type `Animal`
- Output at runtime

```
1 Animal
2 Animal
```
- Truncated object, stored in place: C++ **truncates** the object, coercing to parent class



# Which Object?

❓ Which object are we accessing when invoking `g()` and `f(x-1)` in line 4?

- Class definition

```
1 class A {  
2     int f (int x) {  
3         System.out.println("A.f(" + x + ")");  
4         return (x == 0) ? g() : f(x - 1);  
5     }  
6     int g () { System.out.println ("A.g()"); return 0; }  
7 }
```

```
1 A x = new A();  
2 x.f(2);
```

- Output at runtime

```
1 A.f(2) // x.f  
2 A.f(1) // f(x-1)  
3 A.f(0) // f(x-1)  
4 A.g()  // g()
```

- The object referenced by `x`; what is its explicit name inside class `A` ?





# Which Object: `this`

❓ What are the static and dynamic types of `this` ?

- Class definition

```
1 class A {  
2     int f (int x) {  
3         System.out.println("A.f(" + x + ")");  
4         return (x == 0) ? this.g () : this.f (x - 1);  
5     }  
6     int g () { System.out.println ("A.g()"); return 0; }  
7 }
```

```
1 A x = new A();  
2 x.f(2);
```

- Output at runtime

```
1 A.f(2) // x.f  
2 A.f(1) // this.f  
3 A.f(0) // this.f  
4 A.g()  // this.g
```

- Reference `this` has
  - static type of enclosing class (here: `A` )
  - dynamic type of object (here: `A` )



# Static and Dynamic Type of `this`

❓ What are the static and dynamic types of `x` and `this` ?

- Class `B` inherits `f`, overrides `g`

```
1 class A {  
2     int f (int x) {  
3         System.out.println("A.f(" + x + ")");  
4         return (x == 0) ? this.g() : this.f(x - 1);  
5     }  
6     int g () { System.out.println ("A.g()"); return 0; }  
7 }  
8 class B extends A {  
9     int g () { System.out.println ("B.g()"); return 0; }  
10 }
```

```
1 B x = new B();  
2 x.f(2);
```

- Output at runtime

```
1 A.f(2) // x.f  
2 A.f(1) // this.f  
3 A.f(0) // this.f  
4 B.g()  // this.g
```

- `x` has static type `B`, dynamic type `B`
- `this` (line 4) has static type `A`, dynamic type `B`



# Static and Dynamic Type of `this`

- What are the static and dynamic types of `x` and `this` ?

```
1 class A {  
2     int f (int x) {  
3         System.out.println("A.f(" + x + ")");  
4         return (x == 0) ? this.g() : this.f(x - 1);  
5     }  
6     int g () { System.out.println("A.g()"); return 0; }  
7 }  
8 class B extends A {  
9     int f (int x) { System.out.println("B.f(" + x + ")"); return this.f(x); }  
10    int g ()      { System.out.println("B.g()"); return 0; }  
11 }
```

```
1 A x = new B();  
2 x.f(1);
```

? What happens?

- Output at runtime

```
1 B.f(1) // x.f  
2 B.f(1) // this.f  
3 ...    // infinite loop
```

- ? Why?

- `x` : static type `A` , dynamic type `B`
- `this` (l. 4): static `A` , dynamic `B`
- `this` (l. 9): static `B` , dynamic `B`



# How to Access Method in Super-Class?

- Access `A.f` with `super`

```
1 class A {  
2     int f (int x) {  
3         System.out.println("A.f(" + x + ")");  
4         return (x == 0) ? this.g() : this.f(x - 1);  
5     }  
6     int g () { System.out.println("A.g()"); return 0; }  
7 }  
8 class B extends A {  
9     int f (int x) { System.out.println("B.f(" + x + ")"); return super.f(x); }  
10    int g ()      { System.out.println ("B.g()"); return 0; }  
11 }
```

```
1 A x = new B ();  
2 x.f(1);
```

? What happens?

- Output at runtime

```
1 B.f(1) // x.f  
2 A.f(1) // super.f  
3 B.f(0) // this.  
4 A.f(0) // super.f  
5 B.g()  // this.g
```

? Why?

- `x` : static `A` , dynamic `B` , so `x.f` selects `B.f`
- `super` (line 9) explicitly requests `f` of super-class
- `this` (line 4): static `A` , dynamic `B` , so `this.f` selects `B.f`



## Document and Check Inheritance

- ❓ Let programmers specify how classes can be instantiated?
- ❓ Let programmers decide which methods can/must/cannot be overridden?



# Abstract Classes

- Abstract class `A`

```
1 abstract class A {  
2     int f (int x) { /* ... */ }  
3     int g () { /* ... */ }  
4 }
```

```
1 A x = new A();
```

❓ What happens?

- Compile error: abstract classes cannot be instantiated

```
1 | Error:  
2 | A is abstract; cannot be instantiated  
3 | A x = new A ();  
4 |     ^-----^
```



# Abstract Methods

- Abstract method `g`

```
1 abstract class A {  
2     int f (int x) { /* ... */ }  
3     abstract int g ();  
4 }
```

```
1 class B extends A {  
2  
3  
4 }
```

❓ What happens?

- Compile error: Abstract methods must be overridden in concrete subclass

```
1 | Error:  
2 | B is not abstract and does not override abstract method g() in A  
3 | class B extends A {  
4 | ^-----...
```



# Final Methods

- Final method `f`

```
1 abstract class A {  
2     final int f(int x) { /* ... */ }  
3     abstract int g();  
4 }
```

```
1 class B extends A {  
2     int f(int x) { /* ... */ }  
3     int g() { /* ... */ }  
4 }
```

? What happens?

- Compile error: final methods cannot be overridden

```
1 | Error:  
2 | f(int) in B cannot override f(int) in A  
3 |     overridden method is final  
4 |     int f (int x) { ... }  
5 |     ^-----^
```

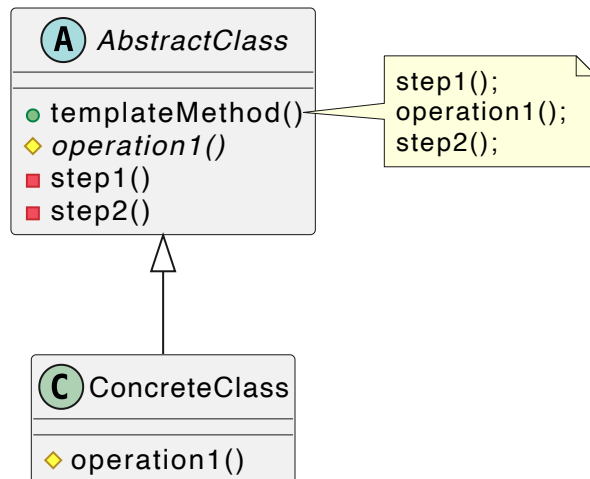




# Use Case: Design Pattern Template Method

Defines the skeleton of an algorithm, deferring varying steps to subclasses

- Factor common behavior into a base class to avoid code duplication
- Implement algorithm structure once, varying behavior in subclasses



? In functional programming?

```
1 abstract class AbstractClass {
2     public final void templateMethod() {
3         step1();
4         operation1();
5         step2();
6     }
7     protected abstract void operation1();
8     private void step1() { /* ... */ }
9     private void step2() { /* ... */ }
10 }
11 class ConcreteClass extends AbstractClass {
12     @override
13     protected void operation1() { /* ... */ }
14 }
```



# Summary

- **Inheritance:** links classes statically at compile time

## Static Dispatch

- Early binding at compile time
- Select method **based on type of reference**

## Dynamic Dispatch

- Late binding at runtime
- Select method **based on type of object**

- Design patterns often provide object-oriented solutions of functional paradigms
- GoF: Favor *object composition* over *class inheritance*
- See [Design Patterns](#), [Design Pattern Catalog](#), and [Design Principles](#)



# Static and Dynamic Types

```
1 interface Fn { int apply (int x); }
```

## Lambda notation

```
1 Fn[] fs = new Fn[2];
2 for (int i = 0;
3     i < fs.length;
4     i++) {
5     int j = i + 1;
6     fs[i] = x -> x + j;
7 }
```

- `fs[i]` : static type `Fn`,  
anonymous dynamic  
type of `x->x+j`

## Anonymous classes

```
1 Fn[] fs = new Fn[2];
2 for (int i = 0;
3     i < fs.length;
4     i++) {
5     int j = i + 1;
6     fs[i] = new Fn() {
7         int apply (int x) {
8             return j + x;
9         }
10    }
11 }
```

- `fs[i]` : static type `Fn`,  
anonymous dynamic  
type of `new Fn() ...`

## Explicit classes

```
1 Fn[] fs = new Fn[2];
2
3 class F implements Fn {
4     public int apply(int x) {
5         return x + 1;
6     }
7 }
8 class G implements Fn {
9     public int apply(int x) {
10        return x + 2;
11    }
12 }
13
14 fs[0] = new F();
15 fs[1] = new G();
```

- `fs[i]` : static type `Fn`,  
dynamic type `F` or `G`



# Examples: C++

```
1 class Animal { public: virtual string to_string() { return "Animal"; } };
2 class Bird: public Animal { public: virtual string to_string() { return "Bird"; } };
3 class Cat: public Animal { public: virtual string to_string() { return "Cat"; } };
```

- Pointers + virtual :

```
1 Animal* a = new Bird ();
2 cout << a->to_string() << endl;
3 a = new Cat ();
4 cout << a->to_string() << endl;
```

- Reference + virtual :

```
1 void print(Animal& a) {
2     cout << a.to_string() << endl;
3 }
4
5 Bird b = Bird ();
6 Cat c = Cat ();
7
8 print(b);
9 print(c);
```

- Static types:

```
1 Bird* b = new Bird ();
2 cout << b->to_string() << endl;
3 b = new Cat ();
4 cout << b->to_string() << endl;
```



# Examples: Scala

```
1 class A:
2   def f () = { println("A.f"); this.g () }
3   def g () = { println("A.g"); this.h () }
4   def h () = println("A.h")
5 class B extends A:
6   override def f () = { println("B.f"); super.f () }
7   override def h () = println("B.h")
8   def i () = println("B.i")
```

- Static A , dynamic A

```
1 val a:A = new A()
2 a.f()
3 a.g()
4 a.h()
5 a.i()
```

- Static A , dynamic B

```
1 val a:A = new B()
2 a.f()
3 a.g()
4 a.h()
5 a.i()
```

- Static B , dynamic B

```
1 val b:B = new B()
2 b.f()
3 b.g()
4 b.h()
5 b.i()
```