CSC 347 - 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?

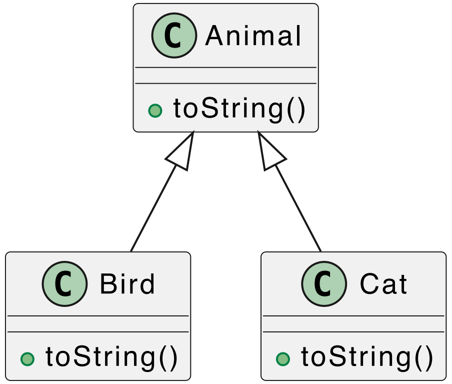
- Identify classes and inheritance
- Describe the difference between static and dynamic dispatch

Scala: Which `toString`?

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

```scala
class Animal                { override def toString () = "Animal" }
class Bird extends Animal { override def toString () = "Bird" }
class Cat  extends Animal { override def toString () = "Cat" }

val xs = Array[Animal] (new Bird(), new Cat ())
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

```
Bird
Cat
```

Java: Which `toString`?

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

```java
class Animal               { public  String toString () { return "Animal"; } }
class Bird extends Animal { public  String toString () { return "Bird"; } }
class Cat  extends Animal { public  String toString () { return "Cat"; } }

Animal[]  xs = { new Bird(), new Cat () };
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

```
Bird
Cat
```

C++: Which `toString`?

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

```cpp
class Animal                { public: string to_string() { return "Animal"; } };
class Bird: public Animal { public: string to_string() { return "Bird"; } };
class Cat:  public Animal { public: string to_string() { return "Cat"; } };

Animal *xs[] = { new Bird(), new Cat() };
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

```
Animal
Animal
```

C++: Which `toString`?

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

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

Animal *xs[] = { new Bird(), new Cat() };
for (Animal *x : xs) cout << x->to_string() << endl;
```

- `x` has
  - static type `Animal`
  - dynamic type `Bird` / `Cat`
- Output at runtime

```
Bird
Cat
```

- C++ dynamically dispatches `virtual` methods **only when** object accessed with a pointer/reference

C++: Which `toString`?

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

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

Animal  xs[] = {     Bird(),     Cat() };
for (Animal  x : xs) cout <<  x.to_string() << endl;
```

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

```
Animal
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

```java
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 }
```

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

- Output at runtime

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

```java
class A {
  int f (int x) {
    System.out.println("A.f(" + x + ")");
    return (x == 0) ? this.g () : this.f (x - 1);
  }
  int g () { System.out.println ("A.g()"); return 0; }
}
```

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

- Output at runtime

```
A.f(2) // x.f
A.f(1) // this.f
A.f(0) // this.f
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 }
```

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

- Output at runtime

```
A.f(2) // x.f
A.f(1) // this.f
A.f(0) // this.f
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`?

```java
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 }
```

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

❓ What happens?

- Output at runtime

```
B.f(1) // x.f
B.f(1) // this.f
...    // 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 }
```

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

What happens?

- Output at runtime

```
B.f(1) // x.f
A.f(1) // super.f
B.f(0) // this.
A.f(0) // super.f
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`

```
abstract class A {
  int f (int x) { /* ... */ }
  int g () { /* ... */ }
}
```

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

What happens?

- Compile error: abstract classes cannot be instantiated

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

Abstract Methods

- Abstract method `g`

```java
abstract class A {
  int f (int x) { /* ... */ }
  abstract int g ();
}
```

```java
class B extends A {


}
```

What happens?

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

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

Final Methods

- Final method `f`

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

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

What happens?

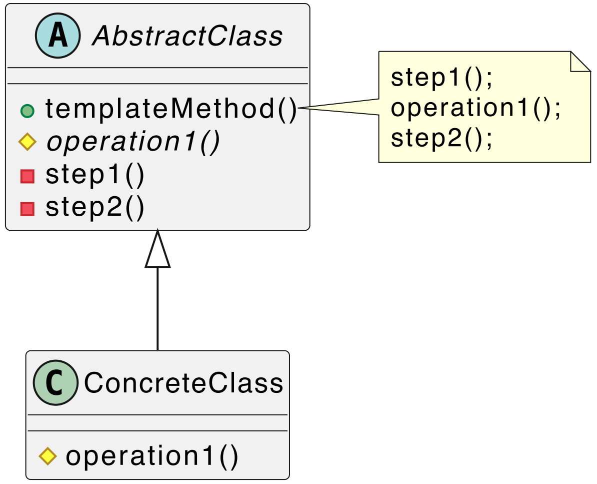
- Compile error: final methods cannot be overridden

```
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




```
abstract class AbstractClass {
  public final void templateMethod() {
    step1();
    operation1();
    step2();
  }
  protected abstract void operation1();
  private void step1() { /* ... */ }
  private void step2() { /* ... */ }
}
class ConcreteClass extends AbstractClass {
  @override
  protected void operation1() { /* ... */ }
}
```

In functional programming?

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

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

Lambda notation

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

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

Anonymous classes

```java
Fn[] fs = new Fn[2];
for (int i = 0;
       i < fs.length;
       i++) {
   int j = i + 1;
   fs[i] = new Fn() {
     int apply (int x) {
       return j + x;
     }
   }
}
```

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

Explicit classes

```java
Fn[] fs = new Fn[2];

class F implements Fn {
  public int apply(int x) {
    return x + 1;
  }
}
class G implements Fn {
  public int apply(int x) {
    return x + 2;
  }
}

fs[0] = new F();
fs[1] = new G();
```

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

Examples: C++

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

- Pointers + `virtual`:

```cpp
Animal* a = new Bird ();
cout << a->to_string() << endl;
a = new Cat ();
cout << a->to_string() << endl;
```

- Reference + `virtual`:

```cpp
void print(Animal& a) {
  cout << a.to_string() << endl;
}

Bird b = Bird ();
Cat c = Cat ();

print(b);
print(c);
```

- Static types:

```cpp
Bird* b = new Bird ();
cout << b->to_string() << endl;
b = new Cat ();
cout << b->to_string() << endl;
```

Examples: Scala

```scala
class A:
  def f () = { println("A.f"); this.g () }
  def g () = { println("A.g"); this.h () }
  def h () = println("A.h")
class B extends A:
  override def f () = { println("B.f"); super.f () }
  override def h () = println("B.h")
  def i ()          = println("B.i")
```

- Static `A`, dynamic `A`

```scala
val a:A = new A()
a.f()
a.g()
a.h()
a.i()
```

- Static `A`, dynamic `B`

```scala
val a:A = new B()
a.f()
a.g()
a.h()
a.i()
```

- Static `B`, dynamic `B`

```scala
val b:B = new B()
b.f()
b.g()
b.h()
b.i()
```