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?

- Understand inheritance
- Understand how methods are chosen at runtime

Which `toString`?

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

```
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());
```

```
1 $ javac AnimalTest.java && java AnimalTest
2 Bird
3 Cat
```

Which `toString`?

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

```
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;
```

- Output at runtime

```
1 $ g++ -std=c++11 -o animal1 animal1.cpp && ./animal1
2 Animal
3 Animal
```

Which `toString`?

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

```
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;
```

- Output at runtime

```
1 $ g++ -std=c++11 -o animal2 animal2.cpp && ./animal2
2 Bird
3 Cat
```

Which `toString`?

- `x` has static type `Animal`, dynamic type `Animal`
- Truncated object, stored in place: C++ *truncates* the object, coercing to parent class

```
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;
```

- Output at runtime

```
1 $ g++ -std=c++11 -o animal3 animal3.cpp && ./animal3
2 Animal
3 Animal
```

Dynamic Dispatch (Java)

```
1 interface Fn {                    int apply (int x);                    }
2 class F implements Fn { public int apply (int x) { return x + 1; } }
3 class G implements Fn { public int apply (int x) { return x + 2; } }
4 class H implements Fn { public int apply (int x) { return x + 3; } }
```

```
1 Fn[] fs = { new F (), new G (), new H () };
2 for (int i = 0; i < fs.length; i++) {
3   System.out.format ("fs[%d].apply(20)=%d\n", i, fs[i].apply(20));
4 }
```

- Output at runtime

```
1 fs[0].apply(20)=21
2 fs[1].apply(20)=22
3 fs[2].apply(20)=23
```

With Anonymous Classes

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

```
1 Fn[] fs = new Fn[3];
2 for (int i = 0; i < fs.length; i++) {
3   int j = i + 1;       // effectively final
4   fs[i] = new Fn() { int apply (int x) { return j + x; } }
5 }
6 for (int i = 0; i < fs.length; i++) {
7   System.out.format ("fs[%d].apply(20)=%d\n", i, fs[i].apply(20));
8 }
```

- Output at runtime

```
1 fs[0].apply(20)=21
2 fs[1].apply(20)=22
3 fs[2].apply(20)=23
```

With Lambda Notation

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

```
1 Fn[] fs = new Fn[3];
2 for (int i = 0; i < fs.length; i++) {
3   int j = i + 1;       // effectively final
4   fs[i] = x -> x + j;
5 }
6 for (int i = 0; i < fs.length; i++) {
7   System.out.format ("fs[%d].apply(20)=%d\n", i, fs[i].apply(20));
8 }
```

- Output at runtime

```
1 fs[0].apply(20)=21
2 fs[1].apply(20)=22
3 fs[2].apply(20)=23
```

Delegation and Inheritance

- Delegation (aka Composition)
  - Object references another object
  - Dynamic relationship between objects
- Inheritance
  - Define class in layers
  - Static relationship between classes

Example

- `this` has static type `A`, dynamic type `A`

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

- Instantiate object, call method `f`

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

- Output at runtime

```
1 A.f (2)
2 A.f (1)
3 A.f (0)
4 A.g ()
5 $1 ==> 0
```

Inheritance

- `B` Inherits `f`, overrides `g`
- `this` has static type `A`, dynamic type `A` or `B`

```
1 class A {
2   int f (int x) {
3     System.out.format ("A.f (%d)%n", 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 }
```

- Instantiate object

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

- `x` has static type `A`, dynamic type `B`
- Output at runtime

```
1 A.f (2)
2 A.f (1)
3 A.f (0)
4 B.g ()
5 $1 ==> 0
```

Abstract Classes

- Abstract classes cannot be instantiated

```
1 abstract class A {
2   int f (int x) {
3     System.out.format ("A.f (%d)%n", 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 }
```

- Instantiate object

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

- Compile error

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

Abstract Methods

- Abstract methods must be overridden in concrete subclass

```
1 abstract class A {
2   int f (int x) {
3     // ...
4   }
5   abstract int g ();
6 }
7 class B extends A { }
```

- Compile error

```
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 methods cannot be overridden

```
1 abstract class A {
2   final int f (int x) {
3     // ...
4   }
5   abstract int g ();
6 }
7 class B extends A {
8   int f (int x) { System.out.format ("B.f (%d)%n", x); return 0; }
9   int g () { System.out.println ("B.g ()"); return 0; }
10 }
```

- Compile error

```
1 |  Error:
2 |  f(int) in B cannot override f(int) in A
3 |    overridden method is final
4 |    int f (int x) { System.out.format ("B.f (%d)%n", x); return 0; }
5 |    ^-------------------------------------------------------------^
```

Hook Methods

- Nonfinal and abstract methods must be overridden

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

- Instantiate object

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

- Output at runtime

```
1 B.f (2)
2 $1 ==> 0
```

Hook Methods

- Nonfinal and nonabstract methods (aka *hooks*) can provide default implementation

```
 1 class A {
 2   int f (int x) {
 3     System.out.format ("A.f (%d)%n", 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.format ("B.f (%d)%n", x); return 0; }
10   int g () { System.out.println ("B.g ()"); return 0; }
11 }
```

- Instantiate object

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

- Output at runtime

```
1 B.f (2)
2 $1 ==> 0
```

Overriding `f`

- Access `A.f` with `this`

```
1  class A {
2    int f (int x) {
3      System.out.format ("A.f (%d)%n", 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.format ("B.f (%d)%n", x); return  this.f(x); }
10   int g () { System.out.println ("B.g ()"); return 0; }
11 }
```

- Instantiate object

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

- Output at runtime

```
1 B.f (2)
2 B.f (2)
3 B.f (2)
4 B.f (2)
5 ... // infinite loop
```

How to Access Method in Super-Class?

- Access `A.f` with `super`

```
1 class A {
2   int f (int x) {
3     System.out.format ("A.f (%d)%n", 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.format ("B.f (%d)%n", x); return super.f(x); }
10  int g () { System.out.println ("B.g ()"); return 0; }
11 }
```

- Instantiate object

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

- Output at runtime

```
1 B.f (2)                A.f (0)
2 A.f (2)                B.g ()
3 B.f (1)                $1 ==> 0
4 A.f (1)
5 B.f (0)
```

Difference between Inheritance and Delegation

- Inheritance links sub-class to super-class, dynamic dispatch
- Delegation requires explicit linking, static dispatch

```
1 class A {
2   B that;
3   int f (int x) {
4     System.out.format ("A.f (%d)%n", x);
5     return (x == 0) ? this.g () : this.f (x - 1);
6     // return (x == 0) ? that.g () : that.f (x - 1);
7   }
8   int g () { System.out.println ("A.g ()"); return 0; }
9 }
10 class B {
11   A that;
12   int f (int x) { System.out.format ("B.f (%d)%n", x); return that.f (x); }
13   int g () { System.out.println ("B.g ()"); return 0; }
14 }
```

- Instantiate objects

```
1 A x = new A ();
2 B y = new B ();
3 x.that = y;
4 y.that = x;
5 y.f (2);
```

- Output at runtime

```
1 B.f (2)
2 A.f (2)
3 A.f (1)
4 A.f (0)
5 A.g ()
6 $1 ==> 0
```

Difference between Inheritance and Delegation

- Inheritance links sub-class to super-class, dynamic dispatch
- Delegation requires explicit linking, static dispatch

```
1 class A {
2   B that;
3   int f (int x) {
4     System.out.format ("A.f (%d)%n", x);
5     // return (x == 0) ? this.g () : this.f (x - 1);
6     return (x == 0) ? that.g () : that.f (x - 1);
7   }
8   int g () { System.out.println ("A.g ()"); return 0; }
9 }
10 class B {
11   A that;
12   int f (int x) { System.out.format ("B.f (%d)%n", x); return that.f (x); }
13   int g () { System.out.println ("B.g ()"); return 0; }
14 }
```

- Instantiate objects

```
1 A x = new A ();
2 B y = new B ();
3 x.that = y;
4 y.that = x;
5 y.f (2);
```

- Output at runtime

```
1 B.f (2)
2 A.f (2)
3 B.f (1)
4 A.f (1)
5 B.f (0)
6 A.f (0)
7 B.g ()
8 $1 ==> 0
```

Delegation-based Languages

- Javascript provides syntax to express delegation

```
1  var A = {
2      f (x) {
3          console.log ("A.f (" + x + ")")
4          return (x == 0) ? this.g () : this.f (x - 1);
5      },
6      g ()  { console.log ("A.g ()"); return 0; }
7  }
8  var B = {
9      f (x) { console.log ("B.f (" + x + ")"); return super.f (x); },
10     g ()  { console.log ("B.g ()"); return 0; }
11 }
```

- Link objects

```
1 Object.setPrototypeOf(B, A);
2 B.f (2);
```

- Output at runtime

```
1 B.f (2)                A.f (0)
2 A.f (2)                B.g ()
3 B.f (1)                $1 ==> 0
4 A.f (1)
5 B.f (0)
```

Summary

- Static vs. dynamic dispatch
  - **Static dispatch**: select method **based on type of reference**
  - **Dynamic dispatch**: select method **based on type of object**
- Inheritance vs. delegation
  - **Inheritance: links classes statically** at compile time
  - **Delegation: links objects dynamically** at runtime
    - GoF: Favor *object composition* over *class inheritance*
    - See Design Patterns or Design Principles
    - Delegate to methods of components