Programming Languages: Recitation 10

Goktug Saatcioglu

11.14.2019

1 Object Oriented Programming

1.1 Features of OOP Languages

- Encapsulation of Data and Code – Data and procedures acting on that data (methods) grouped together within types (classes). Instances of these classes are objects.
- Inheritance – Can define new, 'child', classes based on existing, 'parent', classes. Child class inherits data fields and methods of parent
- Subtyping with Dynamic Dispatch – Objects of child class can appear wherever parent is expected. Dynamic dispatch means that the choice of which method to call – the parent's or child's – is made at runtime.

1.2 Accessebility Modifiers

By default, all members (i.e. fields and methods) of objects are accessible by all other objects. Information hiding and encapsulation can be realized by modifying the accessibility of class or object members using so-called accessibility modifiers. The most important accessebility modifiers in Java are as follows:

- `public` (not that is actually not the default modifier): every other object has access to the member
- `private`: only instances of the current class have access to the member
- `protected`: each instance of the current class as well as all its subclasses have access to the member.

1.3 Subset interpretation of Subtyping

- A type defines a set of values. A subtype defines a set of values that is a subset of the set defined by its parent type.
- A subclass inherits all methods and fields of superclass.
- If the values of `S` are a subset of `T`, then an expression expecting `T` values will not be unpleasantly surprised to receive only `S` values, i.e. If `S <: T`, then every value of type `S` is also a value of `T`.
- In Java: As we go up the inheritance chain a class has fewer and fewer methods and fields (width subtyping), until we reach Object, the supertype of all classes, which has the fewest. Thus for all class types `C` in Java, `C <: Object`. The interpretation of subtyping as subsets holds: every object that has a type lower in an inheritance hierarchy also has a type higher in the hierarchy, but not vice versa.

1.4 Static vs. Dynamic Types and Dynamic Dispatch

A superclass `A` is open for extension since it allows behavior to be extended without modifying `A`'s source code and by adding overriding methods in the subclasses of `A`. To understand the semantics of calls to overridden methods, we have to understand the difference between static and dynamic types.

The static type of an expression in a program is the type that the compiler infers for that expression at compile-time. It is simply the left side of the way we declare variables in Java, i.e. `A a = new...()` has static type `A`. The static type of an object determines which of its fields and methods we can access (i.e. how we can interact with that object). If an expression `e` has static type `A`, then the compiler will only allow us to access the members of type `A` on the result value of `e`.

On the other hand, the dynamic type of an expression is the actual type of the value obtained when the expression is evaluated at run-time. For instance, consider the following code snippet:

```
public class A {
  int x;

  public A(int x) {
    this.x = x;
  }
  public int m() {
    return this.x;
  }
}

public class B extends A {
  int y;

  public B(int x, int y) {
    super(x)
    this.y = y;
  }
  public int m() {
    return this.x + this.y;
  }
}

public class Main {
  public static void main(String[] args) {
    A a = new B(1, 2);
    a.m()
  }
}
```

The static type of `a` in the last line is `A`. The compiler infers this type from the type annotation in the declaration of `a` from the previous line. On the other hand, the dynamic type of a on the last line is `B` since when a is evaluated at run-time, it refers to the `B` instance created on the previous line. The behavior of a call to an overridden method such as `m` on the last line is determined by the dynamic type object whose method we are calling of the method call. The call goes to the most recent implementation of the method in the subtype hierarchy, starting from the dynamic type of the object itself. So in this example we end up calling `B.m()` and not `A.m()`. This semantics of method calls is referred to as dynamic dispatch. Methods that are dynamically dispatched are also called virtual methods.

The key point to remember is: in Java, all public and protected methods of classes are virtual by default whereas private methods are non-virtual.

1.5 Exercise

- Let's practice answering questions about dynamic method dispatch.
- Consider the following class definitions.

```
public class Base {
  int x;
  private int y;

  Base(int x, int y) {
    this.x = x;
    this.y = y;
  }
```

```
  public Base m1(String s) {
    System.out.println("Base.m1(String)")
    m2();
    return this;
  }

  public void m2() {
    System.out.println("Base.m2()");
  }
}

public class Derived extends Base {
  Base z;
  int s;

  Derived(Base z, int s) {
    super(0, 0);
    this.z = z;
    this.s = s;
  }

  public Base m1(String s) {
    System.out.println("Derived.m2(String)")
    m2();
    return this;
  }

  public void m2() {
    System.out.println("Derived.m2()");
  }
}
```

- Now consider the following piece of code.

```
public class Main {
  public static void main(String[] args) {
    Base d = new Derived(new Base(0,0), 0);
    d.m1("Hello");
  }
}
```

Answer the following questions. For each question point out which of the methods are virtual and which are non-virtual.

1. What is the static type of d? A: Base. What is the dynamic type of d? A: Derived.
2. What does this call print? A: "Derived.m1(String)" followed by "Derived.m2(String)".
3. What does this call print if I comment out Derived.m1? A: "Base.m1(String)" followed by "Derived.m2(String)".
4. What does this call print if I comment out Derived.m2? A: "Derived.m1(String)" followed by "Base.m2()".
5. What does this call print if I comment out Derived.m1 and make both Base.m2 and Derived.m2 private? A: "Base.m1(String)" followed by "Base.m2()".

6. What does this call print if I change the declaration to `val d: Base = new Base(0,0)`? A: "Base.m1(String)" followed by "Base.m2()".
7. In the preceeding case, what is the static type of d? A: Base. What is the dynamic type of d? A: Base.

- Now consider the following piece of code.

```
Base e = ((new Derived(new Base(0,0), 0)).m1("Hello")).m1("Hello")
```

  Answer the following questions.

  - Repeat questions 2-6 from above. What do we print now for each case?What is the type of `e` for each case?
  - Change the `this` at the end of Derived.m1 to `z`. What do I print out now?
  - Bonus question: Consider another piece of code as continuation to the code above.

```
((new Derived(e, 0)).m1("Hello")).m1("Hello")
```

    What happens in the case I return `this` for questions 2-6? What happens in the case I return `z` for questions 2-6?

1.6 Subtyping on functions

- Covariant subtyping (return types): If `B <: A`, then `C -> B` is a subtype of `C -> A`. The relative order of subtyping of functions is the same as relative order of subtyping of classes
- Contravariant subtyping (argument type): If `B <: A`, then `A -> C` is a subtype of `B -> C`.
- `S1 -> S2 <: T1 -> T2` if `T1 <: S1` and `S2 <: T2`.
- `S1 -> S2` means function is expecting an argument of type `S1` and returns a value `S2`.
- `->` type constructor is covariant in return type and contravariant in argument type.