CSE1720

Week 11, Class Meeting 28 (Lecture 19)

Winter 2013 ◆ Tuesday, March 19, 2013

YORK UNIVERSITÉ UNIVERSITY

9.2.1 The Substitutability Principle

"When a parent is expected, a child is accepted"

the "who"

Q: Is this principle employed by:
(i) the compiler,
(ii) the virtual machine,
(iii) both the compiler and the virtual machine?

A:

(i)

NOT (ii), (iii)

YORK UNIVERSITÉ UNIVERSITY

9.2.1 The Substitutability Principle

"When a parent is expected, a child is accepted"

Q: In which contexts is this principle applied?

A: in any context in which type matching is required

Compiler uses it:

- Assignment statements (LHR / RHS)
- Parameter passing
- Exception clause matching (Ch 11)

YORK UNIVERSITÉ UNIVERSITY

Eg: Substitutability and Assignment Statement

// from the top of the class body of `SpriteDataModel`, where the variables are defined, here is the **declaration**

```
ShooterSprite theShooter;
```

…

// from farther down in the class body of `SpriteDataModel`, in the body of the constructor, here is the **assignment statement**

```
theShooter = new FroggyShooter(dimensionAvailable);
```

// In this assignment statement, the substitutability principle has been applied

//What if we were to declare `theShooter` to be of type `Sprite` instead? Is substitutability principle still applied? what goes wrong?

Try it!!

YORK UNIVERSITÉ UNIVERSITY

Eg: Substitutability and Parameter Passing

```
// from the body of the constructor SpriteDataModel
theSprites.add(theShooter);
theSprites.add(theTarget);
theSprites.add(theScore);
```

// what is the method signature for “add”?

// what is the required type of the method’s parameter?

// signature is add(Sprite)

// passed values are ShooterSprite, TargetSprite,ScoreTally Sprite.

Binding

How would the following be **bound**:

```
theShooter.setInitialLocation();
```

More fundamental question: *what is binding?*

Binding

- Binding refers to the process of *resolving* an the identifiers in a java statement
- *resolve* ≋ locate the *referent* of an *identifier*
  - **identifier** are used for *variables*, *methods* or *class names*
  - the **referent** means “the thing that the identifier stands for”
    - the referent of a *variable* is its value (a reference to an object)
    - the referent of a *class name* is a class definition (the class body)
    - the referent of a *method signature* is a method body (the method name alone is insufficient, need the parameters too)

Early Binding

consider the case of the statement

```
r.m();
```

sec 3.1.3,
sec 9.2.2

- the compiler needs to *resolve* this expression:
  1. compiler determines: what is the *declared type* of r? this is the **referent class definition**
  2. compiler determines: what is the *signature of the method* that is being invoked?
  3. compiler needs to: find the *signature* in the **referent class definition.** It looks for the match that has:
     - the same method name
     - the same number of parameters?
     - parameters of the same type or higher in the hierarchy
  - if multiple target methods found, it chooses the method that requires *the least amount of promotion*

generate bytecode; stipulate the signature in the bytecode

Late Binding

sec 9.2.2

consider *the bytecode* corresponding to the statement:
`r.m(…)`

at runtime, the VM needs to *resolve* this expression:

1. VM determines: what is the class type of the object to which r refers?
   - **this is the *referent class*; it might not be the same as runtime!!!**
2. VM determines: what does the bytecode say is the signature of the invoked method?
   - this was already determined at compilation time, according to the declared type of `r`
3. VM needs to: find the signature from step #2 in the referent class definition
   - the VM *will always find a matching method*
   - the compiler's early binding ensures that at least one matching method can always be found (that of the parent class)
   - VM needs to resolve parameters and pass them to the matching method

9

How would the following be **bound**?

```
theShooter.setInitialLocation();
```

Early Binding

1. What is the declared type of `theShooter`?
   - It is: _____
2. **The compiler** searches that class definition for `setInitialLocation()`
3. Outcome?
   - If one matching class definition is found, great. compiler produces bytecode
   - If more than one found, compiler picks the most specific
   - If not found, compiler issues compile-time error

10

How would the following be **bound**?

`theShooter.setInitialLocation();`

Late Binding

1. What is the actual runtime type of `theShooter`?
   - It is: _____
   - NOTICE!!!! The runtime type of the object that is referenced by `theShooter` **is different than the declared type**
   - If `theShooter` is null, then VM throws `RuntimeException: NullPointerException`
2. The VM searches that class definition for `setInitialLocation()`
3. Possible outcomes?

11

Discussion

`theShooter.setInitialLocation();`

Early Binding

The compiler searched this class definition:_____

Late Binding

The VM searched this class definition:_____

Since the class searched by the VM is different than the class searched by the compiler, is it possible that the method `setInitialLocation()` will not be found?

12

```
theShooter.setInitialLocation();
```

Early Binding (Compiler)

- what is the declared type of `r`? (class `ShooterSprite`)
- what is the signature of the invoked method?
- Compiler locates the target signature in `ShooterSprite`
  - if multiple targets methods found, choose the method that requires the least amount of promotion

generate bytecode, stipulate the signature in the bytecode

Late Binding (VM)

- what is the runtime type of the the object to which r refers? (class `FroggyShooter`)
- what is the signature of the invoked method? (look in bytecode)
- VM locates the target signature in `FroggyShooter`

this step could even be done first!

13

YORK UNIVERSITÉ UNIVERSITY

Polymorphism

Which of the following statements are consistent with the meaning of polymorphism?

1. The *object* is polymorphic.
2. The *class definition* is polymophic.
3. The *method* is polymorphic.

```
Ans: Only#3 is consistent; #1 and #2 are
inconsistent
```


YORK UNIVERSITÉ UNIVERSITY

Polymorphism

TAKE AWAY POINT:

Polymorphism is a property that can be found in methods – NOT objects, NOT classes

A method can be polymorphic if it is capable of having multiple forms.

– E.g., one form defined in a parent class, another form defined in a child class

The form is able to change during late binding, when the **actual object type** (during runtime) is different from the **declared type**

YORK UNIVERSITÉ UNIVERSITY

Exercise

Identify polymorphic methods in the game code base

Hints:

- look for variables which have a declared type that is not exactly the same as the runtime type
- for these variables, examine the methods that are being invoked; look here to see whether these are polymorphic

YORK UNIVERSITÉ UNIVERSITY

What is the point of polymorphism?

- allows applications to have *elegant* designs
  - *elegant* in the mathematical sense: “pleasingly ingenious and simple”, not the everyday sense of the word
- applications don’t need to use selection (if statements) to conditionally invoked methods – rather, the runtime type determines which methods get invoked.

Sometimes non-polymorphic methods must be used.

For instance,

`isAlive()` and `move()` are methods that apply only to `ProjectileSprites`

to use them, we use a manual cast

17

Methods that are NOT polymorphic?

Sometimes non-polymorphic methods must be used. For instance,

`isAlive()` and `move()` are methods that apply only to `ProjectileSprites`

18

Methods that are NOT polymorphic?

Use the relational operator `instanceof`

```
for (Sprite s : getSprites()) {
      if (s instanceof ProjectileSprite) {
            ProjectileSprite ps = (ProjectileSprite) s;
            ps.move();
      }
}
```


YORK UNIVERSITÉ UNIVERSITY