Java for Advanced Programmers

Subclassing / Inheritance and Delegation

Bernd Kiefer
Jörg Steffen

November 11, 2022

Our (in)famous example

```
Kisse eats a doppelganger corpse.  Kisse turns into a zruty!

Johanna the Heroine          St:25 Dx:17 Co:18 In:8 Wi:17 Ch:10  Neutr
Dlvl:5  $:11 HP:267(267) Pw:38(38) AC:-8 Xp:22/20003621
```

Subclassing / Inheritance

(More general)
Super Class

(More specific)
Subclass

Subclassing / Inheritance

(More general)
Super Class

(More specific)
Subclass

`Tile.java`

`WallTile.java`

Subclassing / Inheritance

(More general)
Super Class

(More specific)
Subclass

Tile.java

```
int x, y;
boolean canBeTamperedWith();
```

WallTile.java

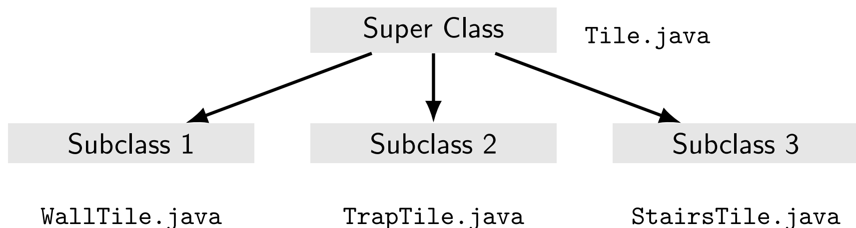
```
boolean canBeDestroyed();
boolean isIntact();
```

Subclassing / Inheritance

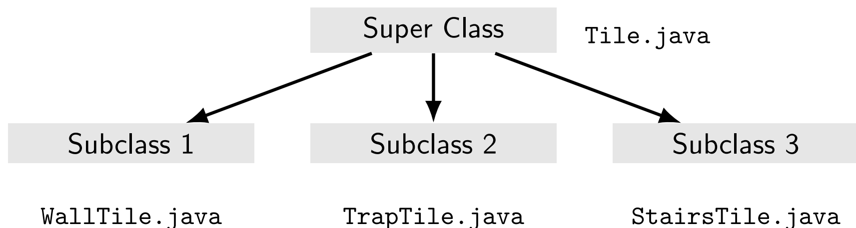
(More general) Super Class	`Tile.java`	`int x, y;` `boolean canBeTamperedWith();`
(More specific) Subclass	`WallTile.java`	`boolean canBeDestroyed();` `boolean isIntact();`

- Subclass inherits fields and methods of super class
- `WallTile` is a `Tile`:
  - A variable of type `Tile` can refer to a `WallTile` object
  - A method requiring a `Tile` parameter can be passed a `WallTile` object

Subclassing / Inheritance



Subclassing / Inheritance



- Common functionality can be put into the super class
- Super class defines interface, subclasses implement them (differently)
- This is one way of extending existing functionality, we'll learn about another later
- Remember design by contract: Subclasses should never break the contract of the super class (aka Liskov substitution principle)

A Simple Example with Overriding

```
class Tile {
  int x, y;

  /** Return true if the state of this tile can be changed by applying force */
  boolean canBeTamperedWith() { return false; }
}
```

A Simple Example with Overriding

```
class Tile {
  int x, y;

  /** Return true if the state of this tile can be changed by applying force */
  boolean canBeTamperedWith() { return false; }
}


class WallTile extends Tile {
  boolean intact;

  boolean canBeTamperedWith() { return true; }

  /** Return true if this piece of wall is still intact */
  boolean isIntact() { return intact; }
}
```

Compile-Time and Run-Time Type

```
boolean applyForce(Hero h, Force f, Tile t) {
  if (t.canBeTamperedWith()) {
    return t.applyForce(f);
  } else {
    tell(h, "A vain effort");
  }
  return false;
}
...

Position attackWhere = h.getPosition().add(h.getAttackDirection());
Tile t = map.getTile(attackWhere); // Returns some kind of Tile

if (applyForce(h, h.getAttackForce(), t)) {
  // update tile display
  ...
}
```

Abstract classes and methods

- In a class, you can specify a method signature only (no body)
- Such methods must be marked with the keyword abstract:
  `abstract boolean canBeTamperedWith();`
- Every class with an abstract method is automatically abstract (and must be marked as such)
- You can not create objects of an abstract class (no valid constructor)
- Interfaces are special classes with only abstract methods and no fields

Classes and Interfaces

```
interface PossibyChangeable {
  abstract boolean canBeTamperedWith(); // abstract is optional here
}

abstract class Tile implements PossiblyChangeable {
  int x, y;
}

// WallTile can stay the same, and it's a ``concrete'' class!
```

- A class can `extend` only one class, but `implement` any number of interfaces!
- An interface can `extend` any number of interfaces
- This is how Java “solves” multiple inheritance

Extending Functionality: Delegation

- Instead of overriding an existing class, use the functionality of an object
- That object is stored in a field of the class
- This can be used to extend the functionality of the original class, or provide something completely different

```
class DisarmableExplosive {
  Explosive ex;

  void explode(Hero h) { ex.explode(h); }

  void disarm(Hero h) { ex = null; }
}
```

- Delegation is by far more flexible (and often more appropriate) than Inheritance