Java for Advanced Programmers

Inheritance: Details

Bernd Kiefer
Jörg Steffen

November 24, 2022

Excursion: References

- Java distinguishes two categories of data types
  - primitive types: `int,char,double,...`
  - reference types: all classes, interfaces, arrays
- Variables for POD types contain the value itself
- Variables of reference types contain
  a reference to an object somewhere in memory
- The same holds for method parameters and fields

Special References

- The `null` reference signals that a variable / field does not point to any object

Special References

- The null reference signals that a variable / field does not point to any object
- In every method, there is a special reference this

```
class Hero {
  int x, y;
  void move(int dx, int dy) { x += dx; y += dy; }
  void rest() { move(0,0); }
}
```

Special References

- The null reference signals that a variable / field does not point to any object
- In every method, there is a special reference this

```
class Hero {
  int x, y;
  void move(int dx, int dy) { this.x += dx; this.y += dy; }
  void rest() { this.move(0,0); }
}
```

Special References

- The `null` reference signals that a variable / field does not point to any object
- In every method, there is a special reference `this`

```
class Hero {
  int x, y;
  void move(int dx, int dy) { this.x += dx; this.y += dy; }
  void rest() { this.move(0,0); }
}
```

- `this` always refers to the object the method was called with

Special References and Casts

- There is also a special reference `super`, which references the same object, as if it were the superclass
- Since every class has a superclass (minimally `Object`), `super` is always available
- `super` allows you to force the call of a superclass method
- Alternatively, you can use a cast, with which you can explicitely change the compile time type:

```
Tile t = new WallTile(0, 0);
WallTile w = (WallTile)t;
```

- `super` behaves like `((DirectSuperClass)this)`

Inheritance: Constructors

Constructors are **not** inherited

$$\begin{tikzcd}
\texttt{Tile.java} \arrow[d] \\
\texttt{WallTile.java}
\end{tikzcd}$$

Inheritance: Constructors

Constructors are **not** inherited

$$\begin{tikzcd}
\texttt{Tile.java} \arrow[d] & \texttt{Tile(int x, int y)\{...\}} \\
\texttt{WallTile.java} & \texttt{(none defined)}
\end{tikzcd}$$

Inheritance: Constructors

Constructors are not inherited

$$\begin{tikzcd}
\texttt{Tile.java} \arrow[d] & \texttt{Tile(int x, int y)\{...\}} \\
\texttt{WallTile.java} & \texttt{?} \qquad \textit{Compile Error}
\end{tikzcd}$$

- Superclass defines constructor with arguments
  $\rightarrow$ subclass must define one (not necessarily same arguments)

Inheritance: Constructors

Constructors are **not** inherited

$$\begin{tikzcd}
\texttt{Tile.java} \arrow[d] \\
\texttt{WallTile.java}
\end{tikzcd}$$

```
Tile(int x, int y){...}

WallTile(){...};
```

- Superclass defines constructor with arguments
  $\rightarrow$ subclass must define one (not necessarily same arguments)

Inheritance: Constructors

Constructors are not inherited

$$\begin{tikzcd}
\texttt{Tile.java} \arrow[d] & \texttt{Tile(int x, int y)\{...\}} \\
\texttt{WallTile.java} & \texttt{WallTile(int x, int y)\{...\};}
\end{tikzcd}$$

- Superclass defines constructor with arguments
  $\rightarrow$ subclass must define one (not necessarily same arguments)

Inheritance: Constructors

Constructors are **not** inherited

$$\begin{tikzcd} \texttt{Tile.java} \arrow[d] \\ \texttt{WallTile.java} \end{tikzcd}$$

```
Tile(int x, int y){...}

WallTile(int x, int y){ super(x,y); };
```

- Superclass defines constructor with arguments
  → subclass must define one (not necessarily same arguments)
- We would like to avoid duplication of code, but how?
- `this(...)` and `super(...)` come in handy
- Only allowed as first statement in a constructor

Access Control

- Up to now: default access for class members (fields and methods)
- All members with default access can be accessed by
  - all classes in the same package
  - all subclasses of a class
- Other access modifiers (put modifier keyword in front of member)
  - `public` all classes can access the member
  - `protected` all subclasses can access the member
  - `private` only other members of the same class have access
- All methods in an interface are (always) `public` (and abstract)

Access Hierarchy

`public` > `protected` > default > `private`

Why Access Restriction

- Hide implementation, only access to API (contract)
- Avoid tampering with internals (inconsistent state)
- Most important:
  Allows modifying implementation keeping usage unaffected

The `String` Class: A Special Form of Protection

- You've already used the `String` class (e.g.m for the `toString` method)
- You most likely did something like

```
String result = this.firstName + " " + this.lastName;
```

- `String`s are immutable, meaning: you can not change the Object
- Appending to a string, e.g. using `s += " ";` will always create a copy
- To make sure nobody can change this behaviour, you can not inherit from the `String` class, it is marked as final