P2: Inheritance

Claudio Corrodi

March 31, 2017
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;
Static and dynamic types

```java
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;
```

**Static type** of a variable: Type declared in the program, never changes

- wall: Wall
- floor: Floor
- tile: Tile
Static and dynamic types

```java
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}
```

```java
Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;
```

**Dynamic type** of a variable: Type of the object bound to the variable at runtime (may change over time) changes

wall: Wall
floor: Floor
tile: Wall
Static and dynamic types

```java
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;  // tile = floor;
```

Dynamic type of a variable: Type of the object bound to the variable at runtime (may change over time) changes

wall: Wall
floor: Floor
.tile: Floor
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) { ... }

    public void renderTile(Floor floor) { ... }
}
```

Methods within a class can have the same name if they have different parameter lists.

```java
Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);
renderer.renderTile(wall);
renderer.renderTile(floor);
```

Method is selected based on the static type of the arguments.

```java
Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = floor;
renderer.renderTile(tile);
```

Does not compile: Static type of tile is Tile and there is no method named renderTile that takes such an argument.
Overloading

public class Renderer {
    public void renderTile(Wall wall) { ... }

    public void renderTile(Floor floor) { ... }
}

Methods within a class can have the same name if they have different parameter lists.
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) { ... }

    public void renderTile(Floor floor) { ... }
}
```

Methods within a class can have the same name if they have different parameter lists.

```java
Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);

renderer.renderTile(wall);
renderer.renderTile(floor);
```
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) { ... }

    public void renderTile(Floor floor) { ... }
}
```

Methods within a class can have the same name if they have different parameter lists.

```java
Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);

renderer.renderTile(wall);
renderer.renderTile(floor);
```

Method is selected based on the static type of the arguments.
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) { ... }

    public void renderTile(Floor floor) { ... }
}
```

Methods within a class can have the same name if they have different parameter lists.

Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = floor;

renderer.renderTile(tile);

Does not compile: Static type of tile is Tile and there is no method named renderTile that takes such an argument.
Overloading

```java
public class Renderer {
    public String renderTile(Tile aTile) { ... }

    public void renderTile(Tile aTile) { ... }
}
```

Different return types, but same signature does not work! (Can not be compiled.)
public abstract class Tile {
    public void renderOn(Renderer renderer) {
        ...}
}

public class Floor extends Tile {
    @Override
    public void renderOn(Renderer renderer) {
        renderer.renderTile(this);
        // NB: Looks for a method Renderer#render(Floor),
        // since the static type of 'this' is Floor.
    }
}

@Override indicates that we are redefining an inherited method. Does not compile if there is no such method in the superclass (e.g., when there’s a typo).
Changing types when overriding

```java
public abstract class Tile {
    /** Return yourself if the argument is the same tile, null otherwise. */
    public abstract Tile matches(Tile anotherTile) { ... }
}

public class Floor extends Tile {
    public Tile matches(Tile anotherTile) { ... }
}
```
Changing types when overriding

```java
public abstract class Tile {
    /** Return yourself if the argument is the same tile, null otherwise. */
    public abstract Tile matches(Tile anotherTile) { ... }
}

public class Floor extends Tile {
    public Floor matches(Tile anotherTile) { ... }
}
```

Return types can be **more specific** when overriding methods (Floor must be a subtype of Tile).
Changing types when overriding

```java
public abstract class Tile {
    /** Return yourself if the argument is the same tile, null otherwise. */
    public abstract Tile matches(Tile anotherTile) { ... }
}

public class Floor extends Tile {
    public Floor matches(Tile anotherTile) { ... }
}
```
Changing types when overriding

```java
public abstract class Tile {
    /** Return yourself if the argument is the same tile, null otherwise. */
    public abstract Tile matches(Tile anotherTile) { ... }
}

public class Floor extends Tile {
    public Floor matches(Object anotherObject) { ... }
}
```

Accept at least what the inherited method accepts.
Calling an inherited constructor

```java
public abstract class Tile {
    protected int x, y;

    public Tile(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Floor extends Tile {
    protected Game game;
    public Floor(Game game, int x, int y) {
        super(x, y);
        this.game = game;
    }
}

Does not work: Tile does not have a default (argumentless) constructor.
Call an inherited constructor with super(...) (must be the first statement).
```
Calling an inherited constructor

```
public abstract class Tile {
    protected int x, y;

    public Tile(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Floor extends Tile {
    protected Game game;
    public Floor(Game game, int x, int y) {
        super(x, y);
        this.game = game;
    }
}
```

Does not work: Tile does not have a default (argumentless) constructor.
Calling an inherited constructor

```java
public abstract class Tile {  
    protected int x, y;
    public Tile(int x, int y) {  
        this.x = x;
        this.y = y;
    }
}

class Floor extends Tile {  
    protected Game game;
    public Floor(Game game, int x, int y) {  
        super(x, y);  
        this.game = game;
    }
}

Does not work: Tile does not have a default (argumentless) constructor.  
Call an inherited constructor with super(...) 
```
Call an inherited constructor with `super(...)` (must be the first statement).
Overloading & Overriding

- Overloading
  - Same method name, different signatures
  - Return types must match

- Overriding
  - Redefine inherited methods
  - Must call a super constructor if there is no argumentless constructor available in the superclass (with `super(...)`) 
  - Accept more, return less
  - Use `super.methodName(...)` to invoke the inherited method
Attributes and inheritance

Private attributes: Inherited, but not accessible!

```java
public class Tile {
    private int x, y;
    public Tile(int x, int y) {
        this.x = x; this.y = y;
    }
}

public class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(x + "", " + y);
    }
}
```
Attributes and inheritance

Private attributes: Inherited, but not accessible!

```java
public class Tile {
    private int x, y;
    public Tile(int x, int y) {
        this.x = x; this.y = y;
    }
}

public class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(x + ", " + y);
    }
}
```

Does not work: Tile does not have a default (argumentless) constructor.
Private attributes: Inherited, but not accessible!

```java
public class Tile {
    protected int x, y;
    public Tile(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

```java
public class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(x + "", " + y);
    }
}
```

Protected attributes: accessible to subclass.
Private attributes: Inherited, but not accessible!

```java
class Tile {
    private int x, y;
    public Tile(int x, int y) {
        this.x = x; this.y = y;
    }
    protected int getX() { return x; }
    protected int getY() { return y; }
}
class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(getX() + "", " + getY());
    }
}
```

Using (inherited!) getter methods works too.
Shadowing attributes

public class Tile {
    public String name;
    public String getName() { return this.name; }
}

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";
System.out.println(floor.getName());
System.out.println(tile.getName());

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";
System.out.println(floor.name);
System.out.println(tile.name);

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";
System.out.println(floor.name);
System.out.println(tile.name);
Shadowing attributes

```java
public class Tile {
    public String name;
    public String getName() { return this.name; }
}

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";

System.out.println(floor.getName());
System.out.println(tile.getName());
```

public class Tile {
    public String name;
    public String getName() { return this.name; }
}

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";

System.out.println(floor.getName()); // null
System.out.println(tile.getName()); // null
public class Tile {
    public String name;
    public String getName() { return this.name; }
}

class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";

System.out.println(floor.name);
System.out.println(tile.name);
public class Tile {
    public String name;
    public String getName() { return this.name; }
}

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";

System.out.println(floor.name); // null
System.out.println(tile.name); // floor
Exercise 6: Quoridor continued

- Finish exercise 5 if necessary
  - We will give feedback together with exercise 6; ask us if you want us to take a look now

- Place walls, move players, winning conditions

- Logic for playing the game

- `main()` method for manually playing the game
  - (along with your tests of course)