P2: Exercise Session 7

Claudio Corrodi

Sokoban Objects

```
public abstract class SokobanObject {
    public void collide(Player player) {
      print(this + " collides with " + player);
    }
}
public class Box extends SokobanObject { … }
public class Bomb extends SokobanObject { … }
```

Static and dynamic types

```
public abstract class SokobanObject {
    public void collide(Player player) {
      print(this + " collides with " + player);
    }
}
public class Box extends SokobanObject { … }
public class Bomb extends SokobanObject { … }
```

```
Box box = new Box(...);
Bomb bomb = new Bomb(...);
SokobanObject o = box;
```

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

box: Box
bomb: Bomb
o: SokobanObject

Static and dynamic types

```
public abstract class SokobanObject {
    public void collide(Player player) {
      print(this + " collides with " + player);
    }
}
public class Box extends SokobanObject { … }
public class Bomb extends SokobanObject { … }
```

```
Box box = new Box(...);
Bomb bomb = new Bomb(...);
SokobanObject o = box;
```

Dynamic type of a variable: Type of the object bound to the variable at runtime (may change during runtime)
box: Box
bomb: Bomb
o: Box

Static and dynamic types

```java
public abstract class SokobanObject {
    public void collide(Player player) {
      print(this + " collides with " + player);
    }
}
public class Box extends SokobanObject { … }
public class Bomb extends SokobanObject { … }
```

```java
Box box = new Box(...);
Bomb bomb = new Bomb(...);
SokobanObject o = box; o = bomb;
```

Dynamic type of a variable: Type of the object bound to the variable at runtime (may change during runtime)
box: Box
bomb: Bomb
o: Bomb

Overloading

```
public class Player {
   public void collideWith(Box box) {
      box.collide(this);
   }
   public void collideWith(Bomb door) {
      door.collide(this);
   }
}
```

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

Overloading

```
public class Player {
   public void collideWith(Box box) {
      box.collide(this);
   }
   public void collideWith(Bomb door) {
      door.collide(this);
   }
}
```

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

```
Player player = new Player();
Box box = new Box(...);
Bomb bomb = new Bomb(...);

player.collideWith(box);
player.collideWith(bomb);
```

Overloading

```
public class Player {
   public void collideWith(Box box) {
      box.collide(this);
   }
   public void collideWith(Bomb door) {
      door.collide(this);
   }
}
```

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

```
Player player = new Player();
Box box = new Box(...);
Bomb bomb = new Bomb(...);

player.collideWith(box);
player.collideWith(bomb);
```

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

Overloading

```java
public class Player {
   public void collideWith(Box box) {
      box.collide(this);
   }
   public void collideWith(Bomb door) {
      door.collide(this);
   }
}
```

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

```java
Player player = new Player();
Box box = new Box(...);
SokobanObject o = box;

player.collideWith(o);
```

Does not compile: Static type of o is SokobanObject and the call does not match any of the overloaded methods.

Overloading

```
public class Player {
    public Integer collideWith(SokobanObject o) {
        ...
    }
    public String collideWith(SokobanObject o) {
        ...
    }
}
```

Different return types, but same signature does not work!
(this can not be compiled)

Overriding

```java
public abstract class SokobanObject {
   public void collide(Player player) {
      print(this + " collides with " + player);
   }
}
public class Box extends SokobanObject {
   @Override
   public void collide(Player player) {
      super.collide(player);
      player.collideWith(this);
   }
}
```

@Override indicates that we are redefining an inherited method.

Overriding

```
public abstract class SokobanObject {
    public void collide(Player player) {
        print(this + " collides with " + player);
    }
}
public class Box extends SokobanObject {
    @Override
    public void colllide(Player player) {

    }
}
```

Typo! Does not compile!

@Override indicates that we are redefining an inherited method.

Overriding

```
public abstract class SokobanObject {
   public void collide(Player player) {
      print(this + " collides with " + player);
   }
}
public class Box extends SokobanObject {
   @Override
   public void collide(Player player) {
      super.collide(player);
      player.collideWith(this);
   }
}
```

“super” can be used to call the overridden method.

Changing types when overriding

```
public abstract class SokobanObject {
    public abstract SokobanObject interact(Player player);
}
```

```
public class Box extends SokobanObject {
    @Override
    public SokobanObject interact(Player player) {
        return null;
    }
}
```

Changing types when overriding

```
public abstract class SokobanObject {
    public abstract SokobanObject interact(Player player);
}
```

```
public class Box extends SokobanObject {
    @Override
    public Box interact(Player player) {
        return null;
    }
}
```

Return types can be more specific when overriding methods (Box must be a subtype of SokobanObject).

Changing types when overriding

```
public abstract class SokobanObject {
    public abstract SokobanObject interact(Player player);
}
```

```
public class Box extends SokobanObject {
    @Override
    public Box interact(Player player) {
        return null;
    }
}
```

Changing types when overriding

```java
public abstract class SokobanObject {
    public abstract SokobanObject interact(Player player);
}
```

```java
public class Box extends SokobanObject {
    @Override
    public Box interact(Object object) {
        return null;
    }
}
```

Accept **at least** what the inherited method accepts.

Calling an inherited constructor

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

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

```java
public class Box extends SokobanObject {
    private final Game game;

    public Box(Game game, int x, int y) {
        this.game = game;
    }
```

Calling an inherited constructor

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

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

```
public class Box extends SokobanObject {
    private final Game game;

    public Box(Game game, int x, int y) {
        this.game = game;
```

Does not work: SokobanObject does not have a default constructor.

Calling an inherited constructor

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

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

```
public class Box extends SokobanObject {
    private final Game game;

    public Box(Game game, int x, int y) {
        this.game = game;
        super(x, y);
    }
```

Calling an inherited constructor

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

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

```java
public class Box extends SokobanObject {
    private final Game game;

    public Box(Game game, int x, int y) {
        this.game = game;
        super(x, y);
```

Still bad: call to super constructor must be first statement

Calling an inherited constructor

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

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

```java
public class Box extends SokobanObject {
    private final Game game;

    public Box(Game game, int x, int y) {
        super(x, y);
        this.game = game;
    }
```

This is how it’s done

Overloading & Overriding

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

- Overriding
  - Redefine inherited methods
  - Use “super.methodname(...)” (or “super(...)” in constructors)
  - Must call a super constructor if there’s no argumentless constructor available in the superclass
  - Accept more, return less

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

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

public class Box extends SokobanObject {
    public Box(int a, int b) {
        super(a, b);
        print(x + ", " + y);
    }
}
```

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

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

public class Box extends SokobanObject {
    public Box(int a, int b) {
        super(a, b);
        print(x + ", " + y);
    }
}
```

Does not compile!
x and y are not accessible

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

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

public class Box extends SokobanObject {
    public Box(int a, int b) {
        super(a, b);
        print(x + ", " + y);
    }
}
```

Now we have access

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

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

    protected int getX() { return x; }
    protected int getY() { return y; }
}

public class Box extends SokobanObject {
    public Box(int a, int b) {
        super(a, b);
        print(getX() + ", " + getY());
    }
}
```

This works too

“Overriding” attributes

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

public class Box extends SokobanObject {
    public String name;
    public String getName() { return this.name; }
}
```

“Overriding” attributes

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

public class Box extends SokobanObject {
    public String name;
    public String getName() { return this.name; }
}
```

```
Box box = new Box();
SokobanObject obj = box;
obj.name = "box";

System.out.println(box.getName());
System.out.println(obj.getName());
```

“Overriding” attributes

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

public class Box extends SokobanObject {
    public String name;
    public String getName() { return this.name; }
}
```

```java
Box box = new Box();
SokobanObject obj = box;
obj.name = "box";

System.out.println(box.getName()); → null
System.out.println(obj.getName()); → null
```

“Overriding” attributes

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

public class Box extends SokobanObject {
    public String name;
    public String getName() { return this.name; }
}
```

```java
Box box = new Box();
SokobanObject obj = box;
obj.name = "box";

System.out.println(box.name);
System.out.println(obj.name);
```

“Overriding” attributes

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

public class Box extends SokobanObject {
    public String name;
    public String getName() { return this.name; }
}
```

```java
Box box = new Box();
SokobanObject obj = box;
obj.name = "box";

System.out.println(box.name);   → null
System.out.println(obj.name);   → “box”
```

Refactoring in Eclipse

Demo

Exercise 7: More Sokoban!

- Third stage: new elements
  - Pushable bombs
  - Breakable walls
  - Push a bomb into a wall, both disappear
- Refactoring and quality of code tasks
  - Packages, mutability, encapsulation, contracts, …

Comments

- Better commit quality! :-)
  - (with exceptions)
- Good code quality
  - JavaDoc, contracts, tests, …
  - Keep it up!
- Exercise 7 is not that big… use it to catch up!