CSE 115/503

April 5-9, 2010

Announcements

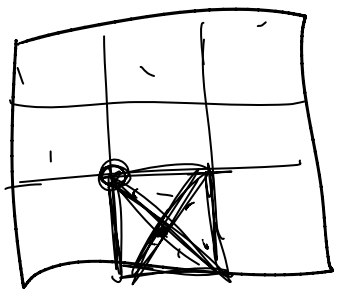
- Lab 7 due next week
- Exam 4 handed back ~~Wednesday~~ Friday
- Exam 5 Review Wednesday 4/14
- Exam 5 Friday 4/16
- Lab 8 due Monday 4/26
- Final exam review session: TBA
- Final exam Thursday 4/29

Lab 7 tips

- Get all ships to appear on screen when game starts
- Then, work on the buttons
  - Button clicked sets the ship in the holder
  - Clicking on board places ship (doesn't matter location)
  - Clicking on board places ship (where you clicked)

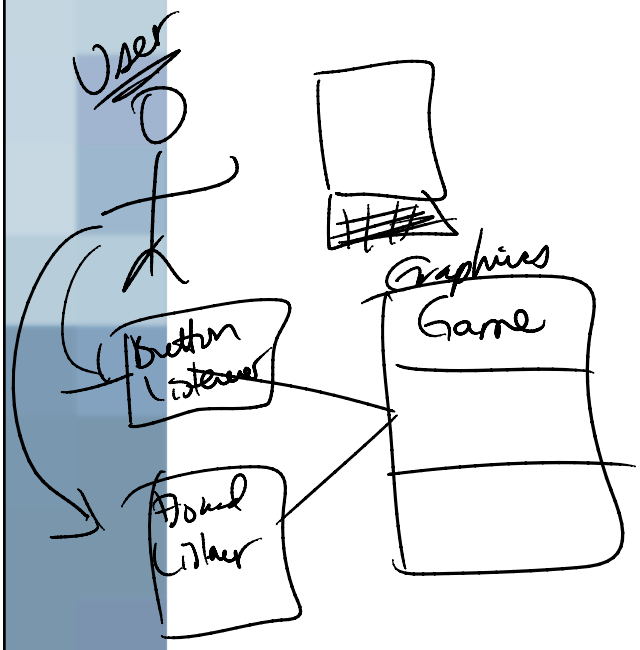
Lab 7 Tips

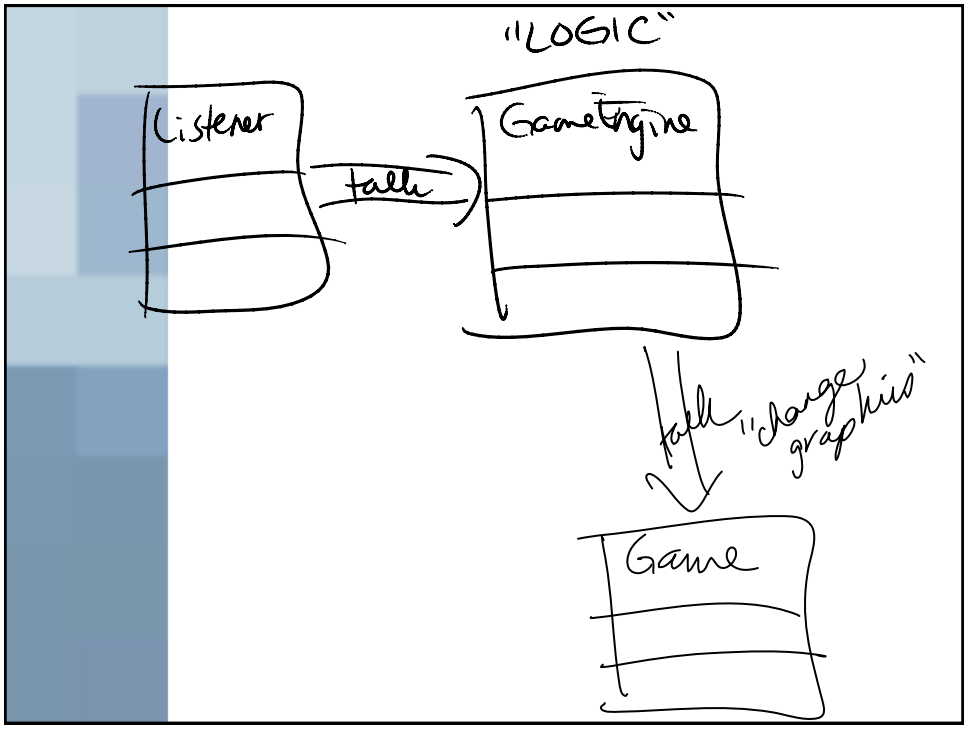
- Then, work on whether ships fall off the edges
- Then, work on ships that overlap each other

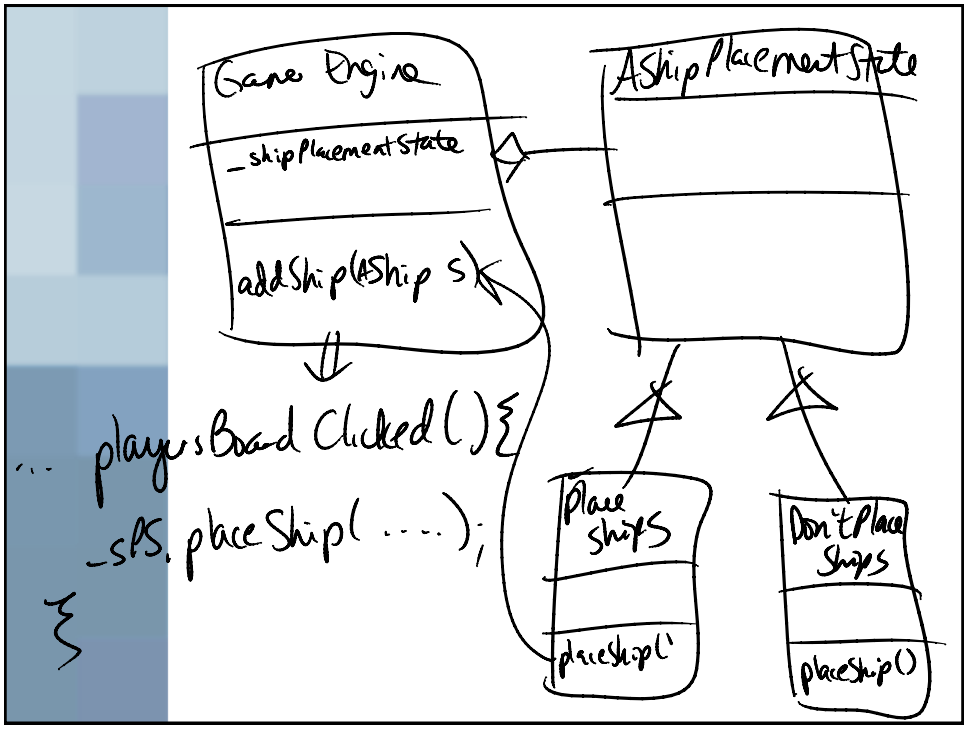


Lab 7 tips

- Remember, that when a user clicks on the screen with the mouse, the location that was clicked is carried in a MouseEvent object that is used as the parameter to the mouseClicked method. You can ask this object for its getPoint() to find out where exactly the user clicked.




"LOGIC"
Listener
talk
GameEngine
talk "change graphics"
Game


GameEngine
_shipPlacementState
addShip(AShip s)
AShipPlacementState
... playersBoardClicked(){
_sPS.placeShip(....);
}
PlaceShips
placeShip()
Don'tPlaceShips
placeShip()

Types

- Declare a variable whose type is java.util.Collection and then assign it an instance of a java.util.LinkedList
- Why is this allowed?
  - Collection is a supertype of LinkedList
  - LinkedList is a subtype of Collection

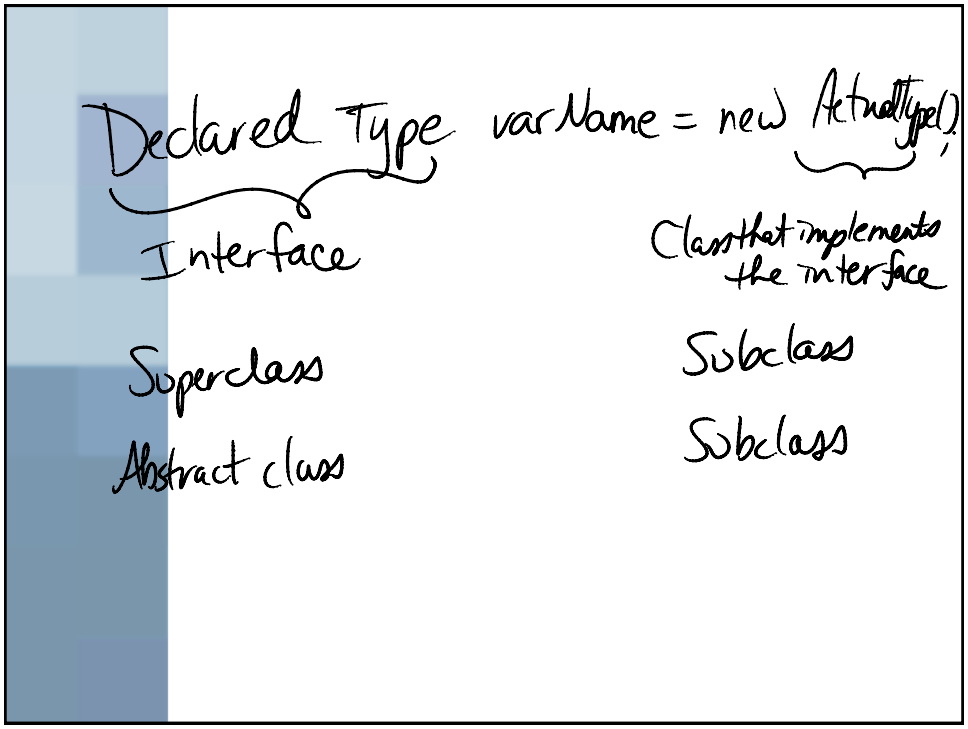
Types – Another Example

- graphics.IGraphic circle = new graphics.Ellipse();
- What are the implications of doing this?
  - circle has a declared type of IGraphic and an actual type of Ellipse

* When a variable's declared type is different than its actual type, the only methods we are allowed to call are those from the declared type. The methods that execute are those from the actual type.

Previous Slide

- Important concept
- Backbone of subtype polymorphism
- Polymorphism is an extremely powerful form of selection that can be used inside object-oriented programs



Interfaces

- Purely abstract entities
- No implementation at all
- Can not create an instance of one
- Contain method headers followed by ;
- Can contain constants, but not instance variables or private methods

Interfaces

- Classes can implement an interface or more than one interface

Inheritance

- Superclass/subclass relationship
- Subclass inherits all public members, but never private ones.
- Subclasses also inherit protected members.
  - Protected is an additional access control modifier that specifies access within the classes and within the class' subclasses.

Inheritance

- A class can extend exactly one other class.
- All classes in Java use inheritance.
- If no superclass explicitly given, the class extends java.lang.Object

Inheritance

- Interfaces can use inheritance as well.
- Interfaces can extend one or more other interfaces.

Abstract classes

- Straddle the middle between interfaces (no implementation) and concrete classes (fully implemented).
- Can not create an instance of an abstract class.
- Abstract classes can implement one or more interfaces.
- Abstract classes can extend exactly one other class.

Why would we use abstract classes or interfaces?

- It allows us to group classes into groups of related entities.
- It allows us to specify certain functionality classes need to have and ensure its implementation via compiler enforcement.

For abstract classes…

- It allows us to share some implementation amongst subclasses and yet still require additional implementation through use of declared abstract methods.

Abstract classes

- Have keyword abstract in their class header.
- Abstract methods have keyword abstract in their method header and have no method body, but rather a ; where the body should be.

Constructor chaining

When a class extends another (concrete or abstract) class, the constructor of the subclass is obligated to call a constructor of the superclass.

Constructor chaining

- No problem if there is a superclass constructor that takes no parameters. Java will call the constructor automatically in this case.
- If all the constructors of the superclass need parameters, then the subclass must explicitly call one of the superclass' constructors using super.

Example from Lab 7

```
public class State extends AShipPlacementState {
    public State() {
        //need to call super with appropriate
        //arguments – an IGameEngine
    }
}
```

Example from Lab 7

```
public class State extends AShipPlacementState {
    public State() {
        super(); //won't work – needs arguments
    }
}
```

Example from Lab 7

```
public class State extends AShipPlacementState {
    public State() {
        super(new GameEngine());
         //will work, but not for Lab 7
    }
}
```

Example from Lab 7

```
public class State extends AShipPlacementState {
    public State(IGameEngine engine) {
        super(engine);
         //will work, good idea for Lab 7
    }
}
```

Example from Lab 7

```
public class State extends AShipPlacementState {
    public State() {
        GameEngine e = new GameEngine();
        super(e);
         //won't work – super must be first line in
         //constructor
    }
}
```

Overriding

- When you use inheritance and inherit methods from the superclass, you can choose to override (change) the method in the subclass.

Accessors & Mutators

- Get & Set methods
- Accessor = get method
- Mutator = set method

Accessors

- Get in name (usually)
- Returns a value
- Therefore, return type that is not void
- Do not take parameters
- Body contains:

```
return value;
```

Where value is the value being returned.

Mutators

- Set in name (usually)
- Change values
- Therefore, parameters needed to specify what new value is
- Return type is void
- Body contains:

```
_instanceVar = paramName;
```