CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

CS608

SOME OO TESTING ISSUES

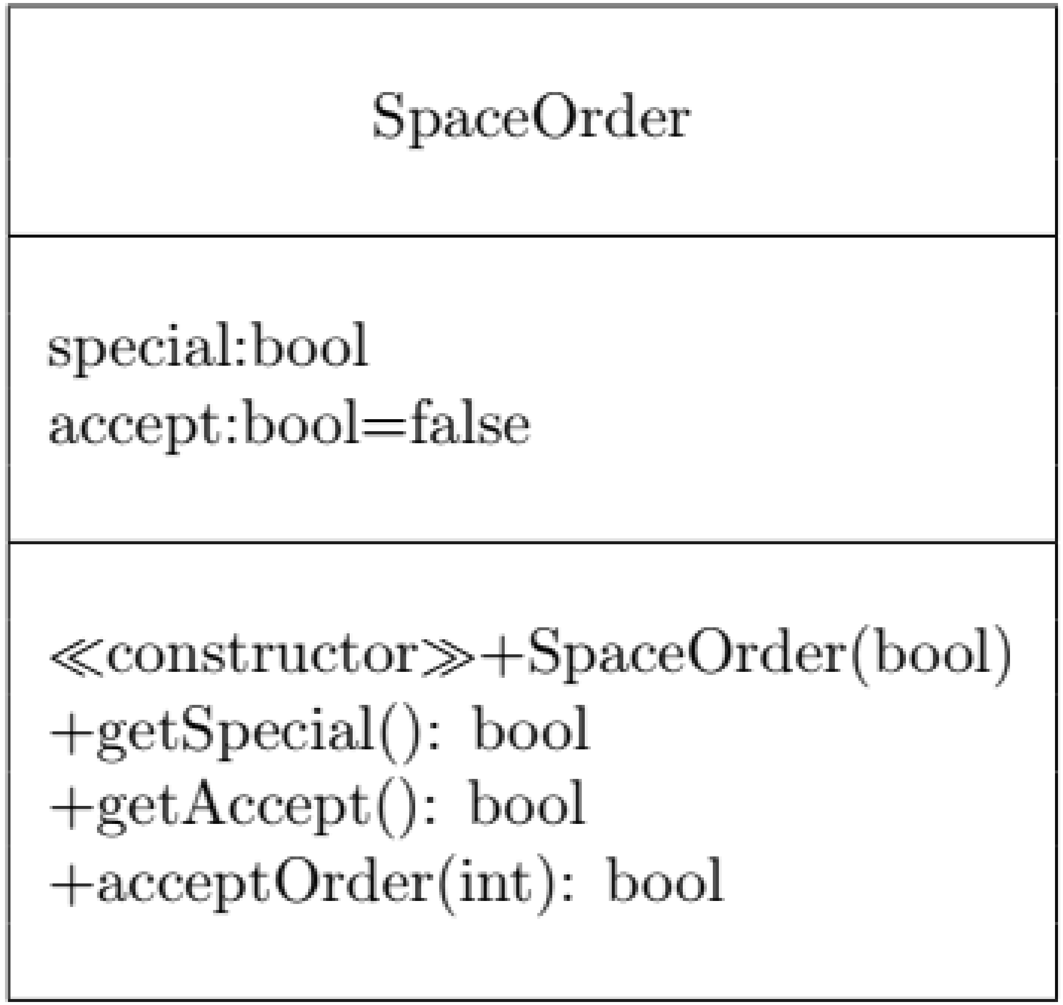
1. TESTING WITHOUT GETTERS
2. SOME INHERITANCE PITFALLS

1. TESTING WITHOUT GETTERS

- First recap of testing with getters
- Then look at testing without getters

TESTING WITH GETTERS

- With getters, easy to check attribute values after a method call



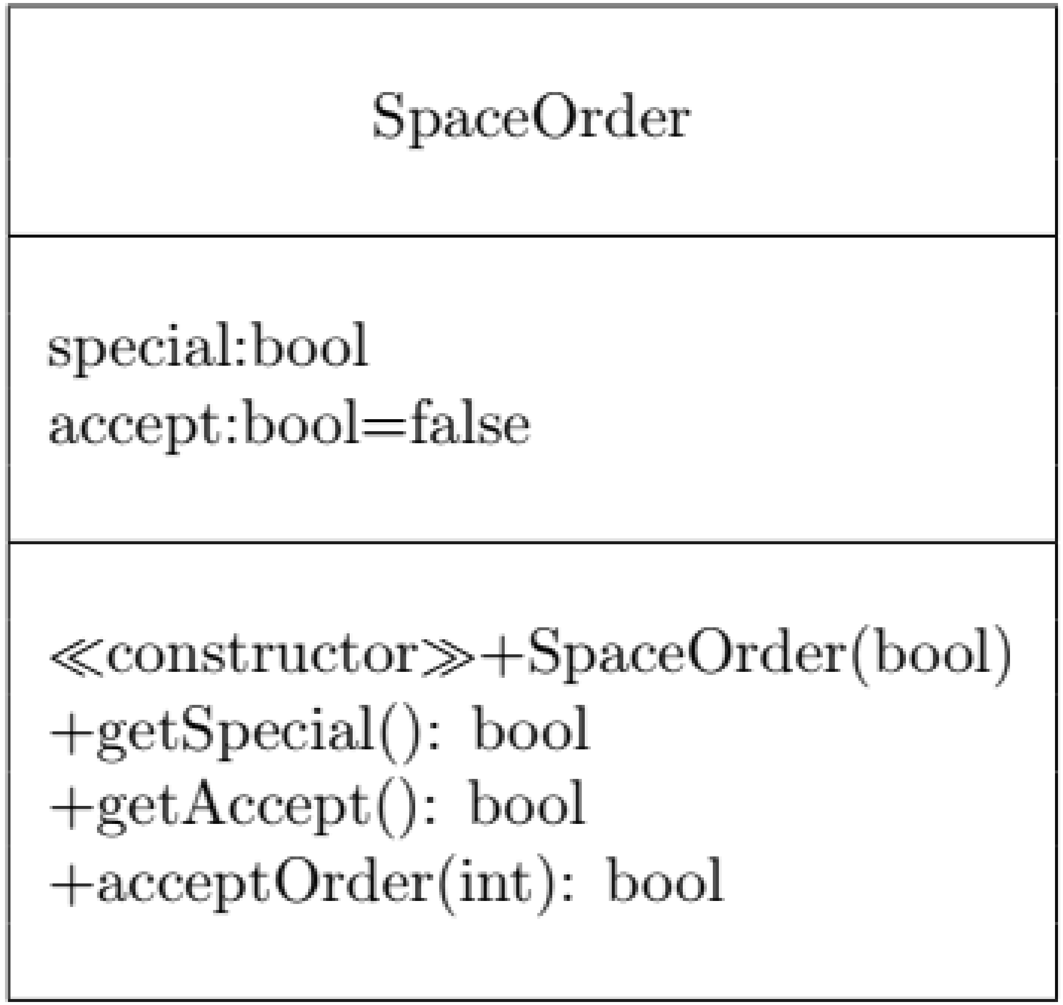
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

CHECK IF SET CORRECTLY

- SpceOrder() sets special and accept
  - getSpecial()
  - getAccept()



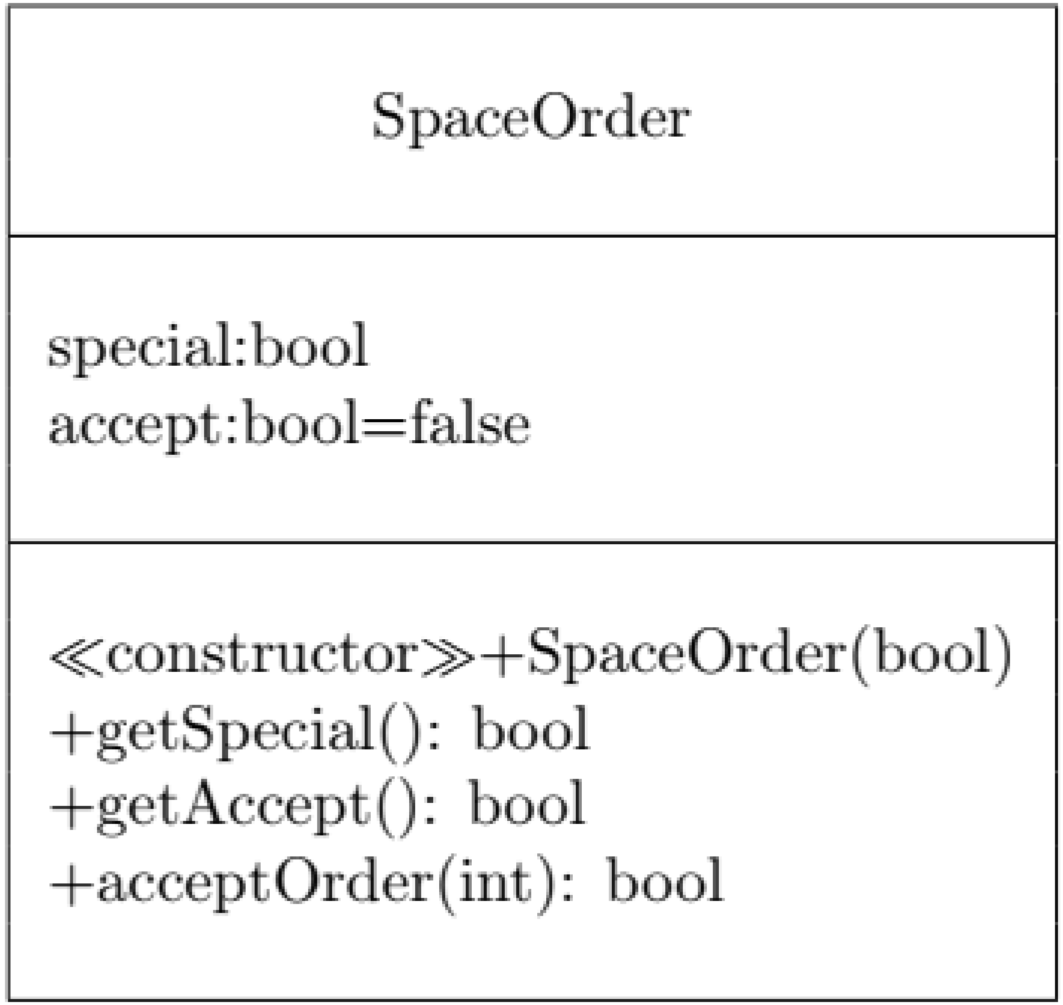
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

CHECK IF SET CORRECTLY

- SpceOrder() sets special and accept
  - getSpecial()
  - getAccept()
- acceptOrder() sets accept
  - getAccept()



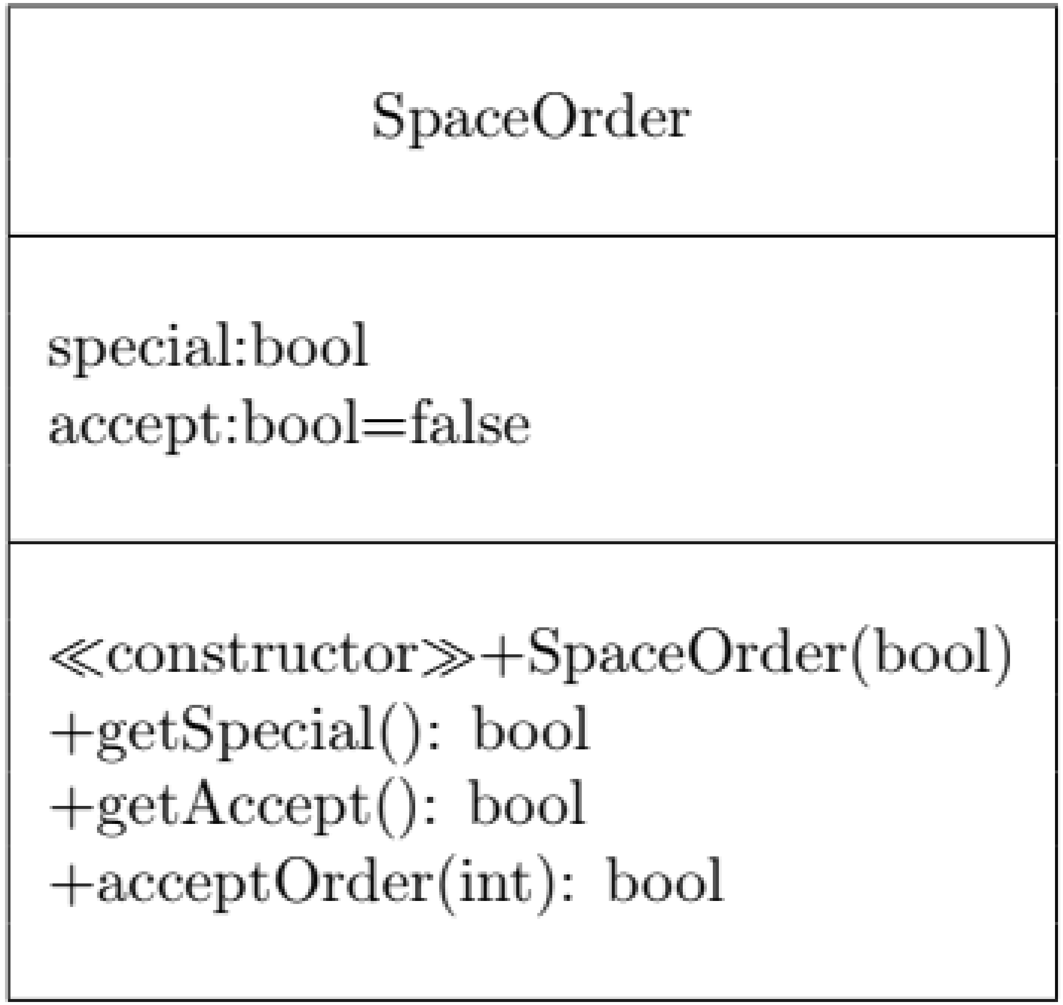
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

CHECK NOT CHANGED INCORRECTLY

- acceptOrder() sets accept but not special
    - getSpecial() – to check not changed



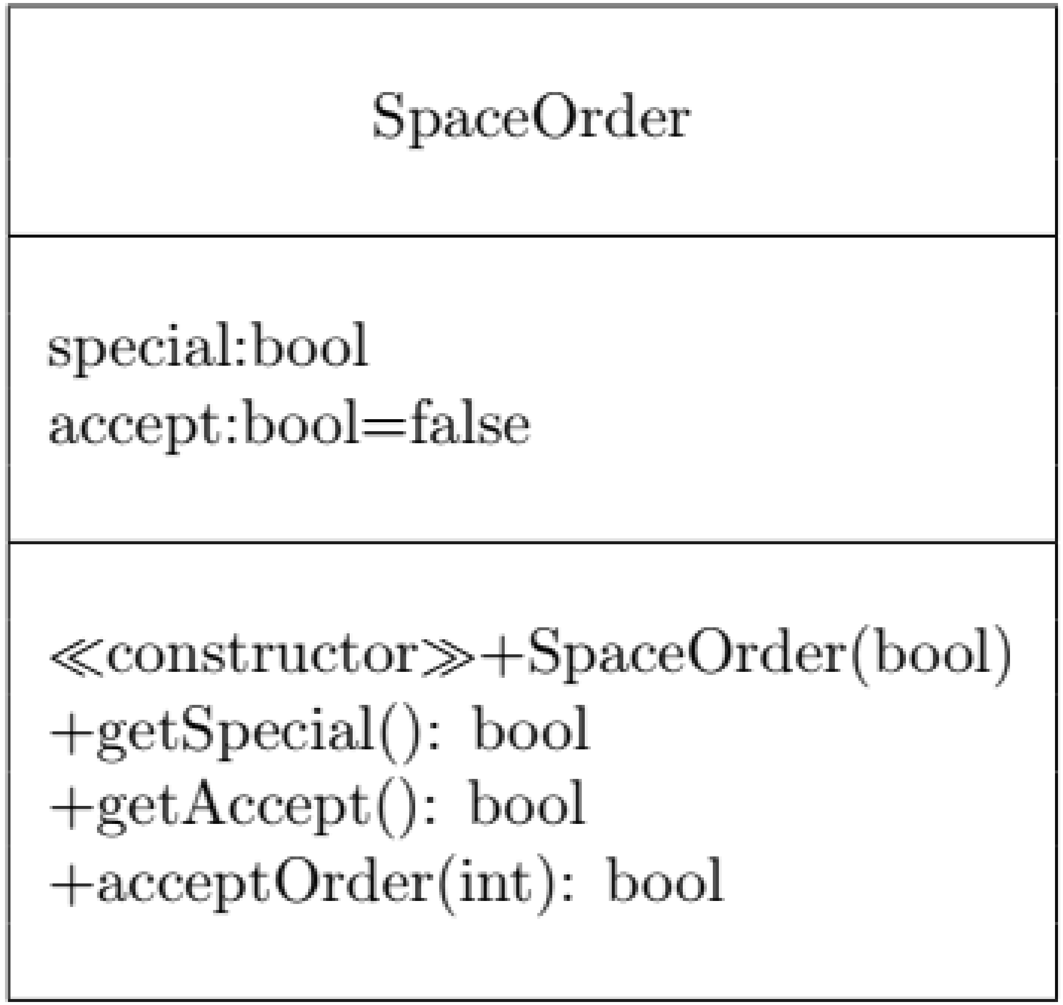
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

CHECK NOT CHANGED INCORRECTLY

- acceptOrder() sets accept but not special
  - getSpecial() – to check not changed
- getSpecial() does not set either
  - getAccept() – to check not changed
  - getSpecial() – to check not changed



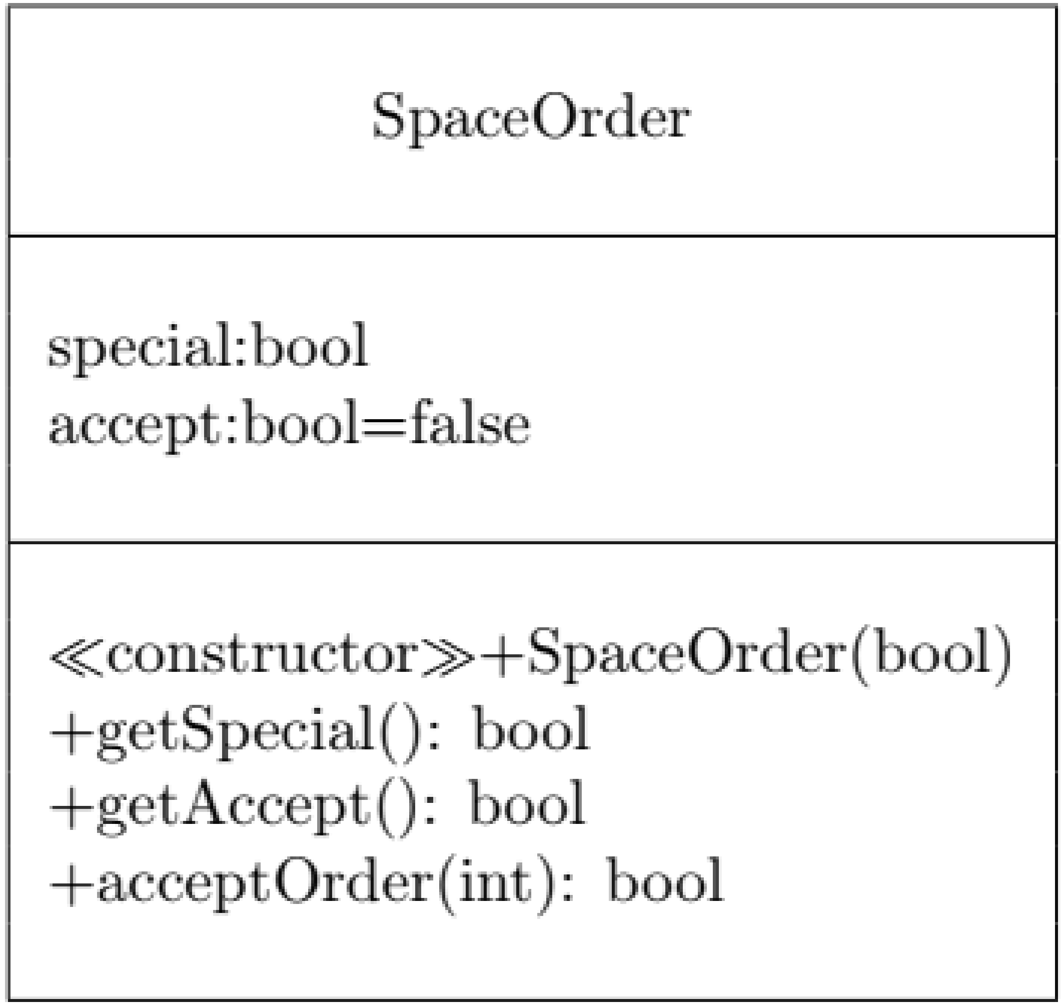
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

CHECK NOT CHANGED INCORRECTLY

- acceptOrder() sets accept but not special
  - getSpecial() – to check not changed
- getSpecial() does not set either
  - getAccept() – to check not changed
  - getSpecial() – to check not changed
- getAccept() does not set either
  - getAccept() – to check not changed
  - getSpecial() – to check not changed



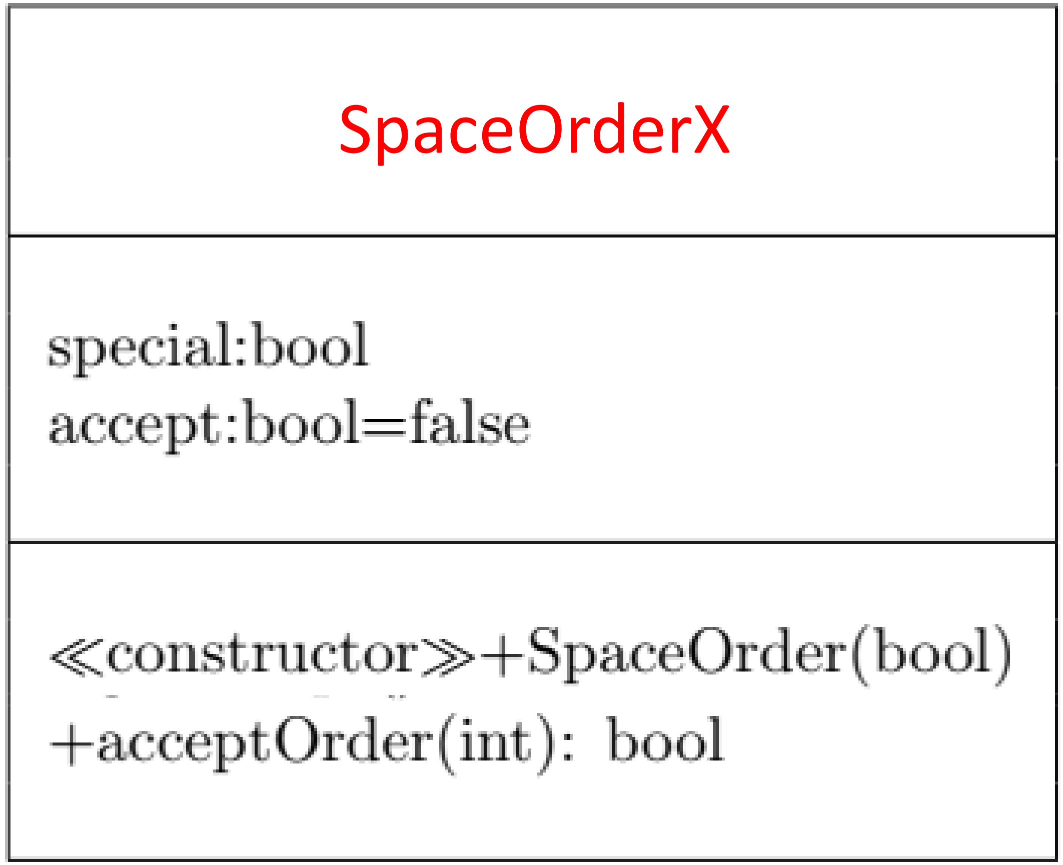
SpaceOrder

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Suggestions?



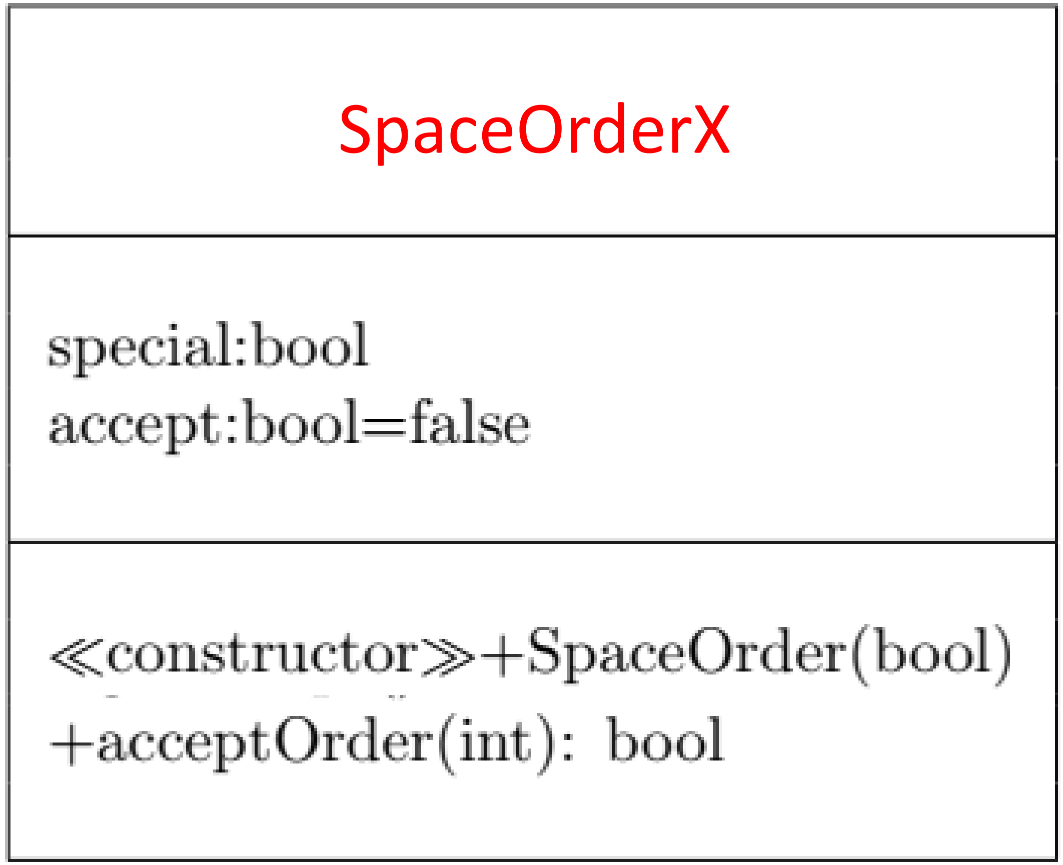
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Options:
  a) Modify the source code for test purposes
  b) Direct access to attributes in test code
  c) Tests in the source code
  d) Java Reflection



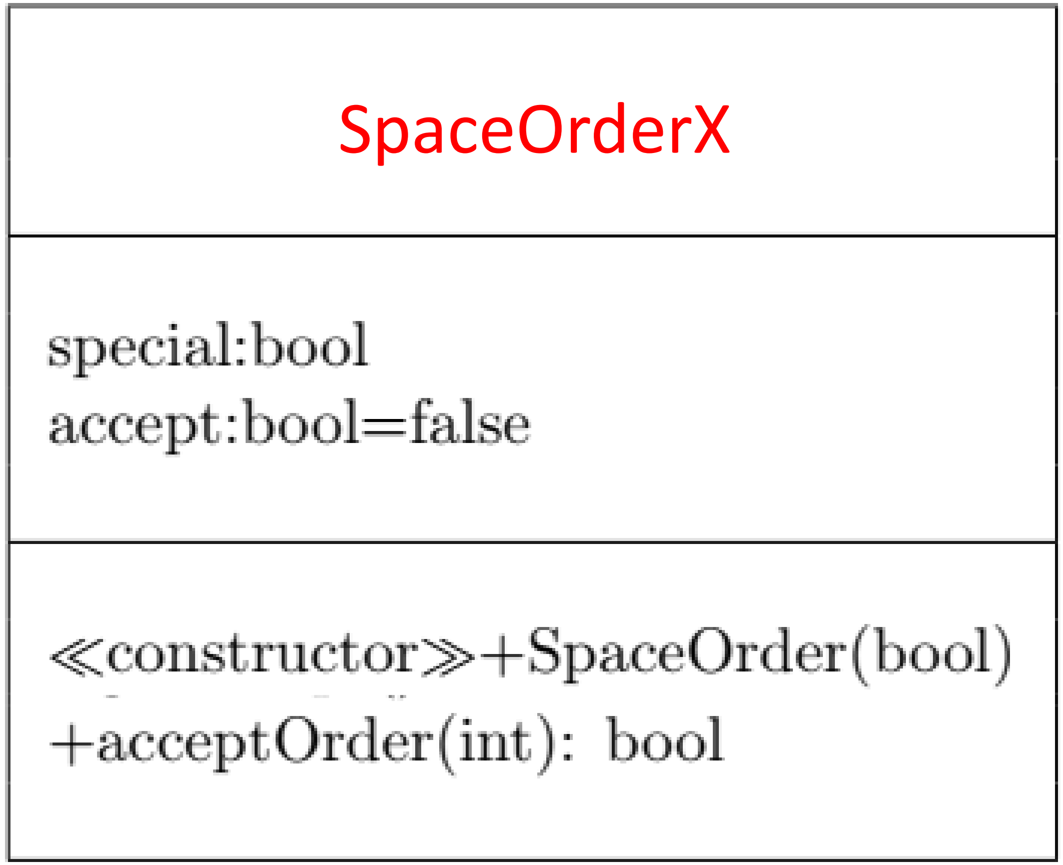
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Options:
  a) Modify the source code for test purposes - add getters
  b) Direct access to attributes in test code
  c) Tests in the source code
  d) Java Reflection



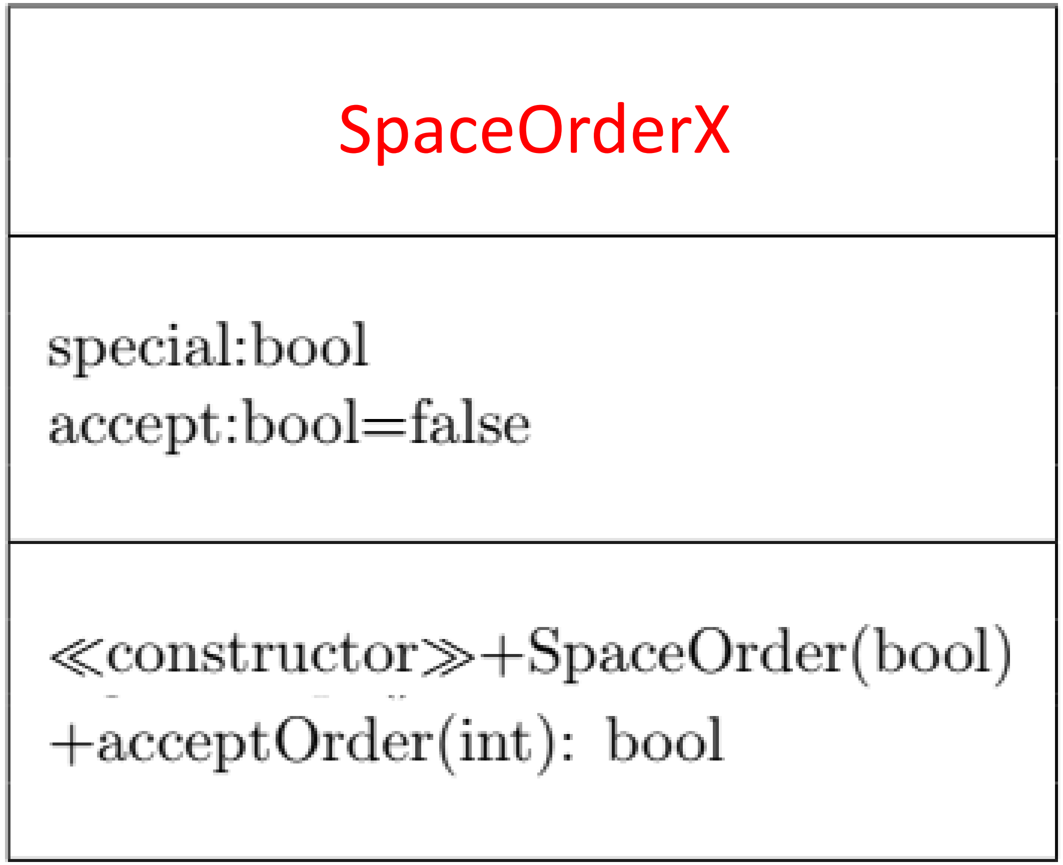
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Options:
  a) Modify the source code for test purposes
  b) Direct access to attributes in test code
     - depends on attribute access modifiers
  c) Tests in the source code
  d) Java Reflection



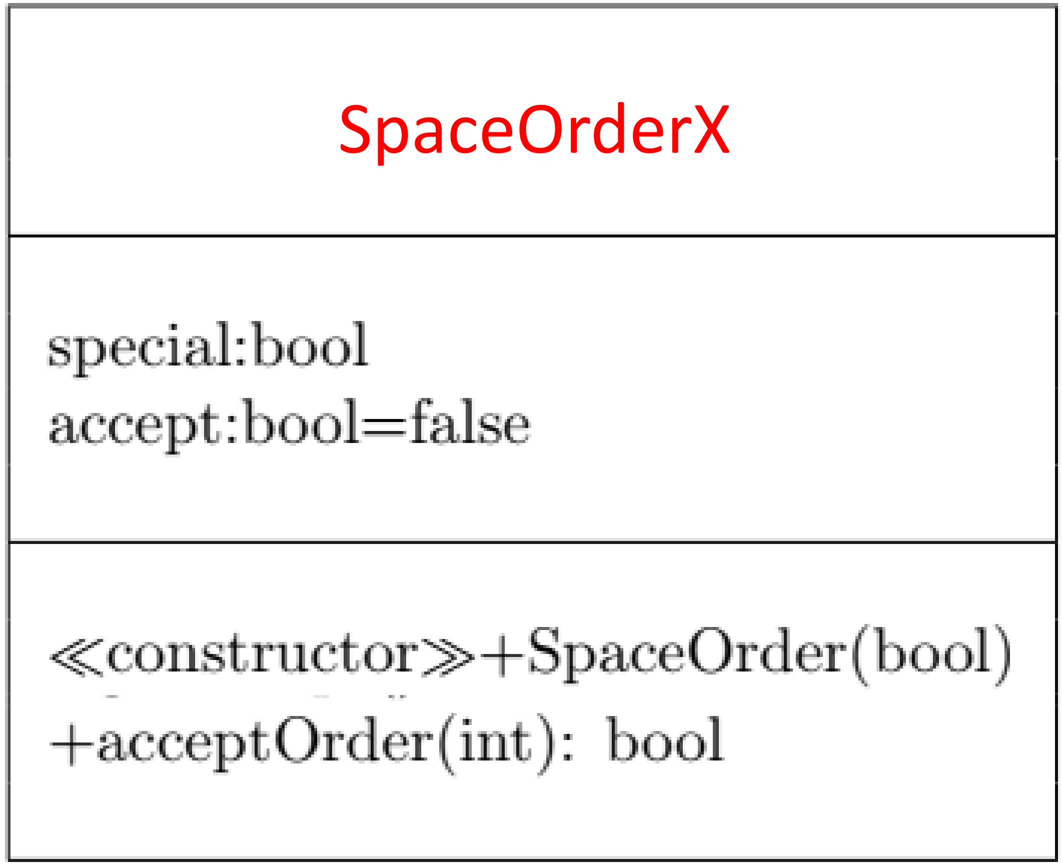
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Options:
  a) Modify the source code for test purposes
  b) Direct access to attributes in test code
  c) Tests in the source code
     - perhaps using a text embedding tool
     - or Built-In Testing (BIT)
  d) Java Reflection



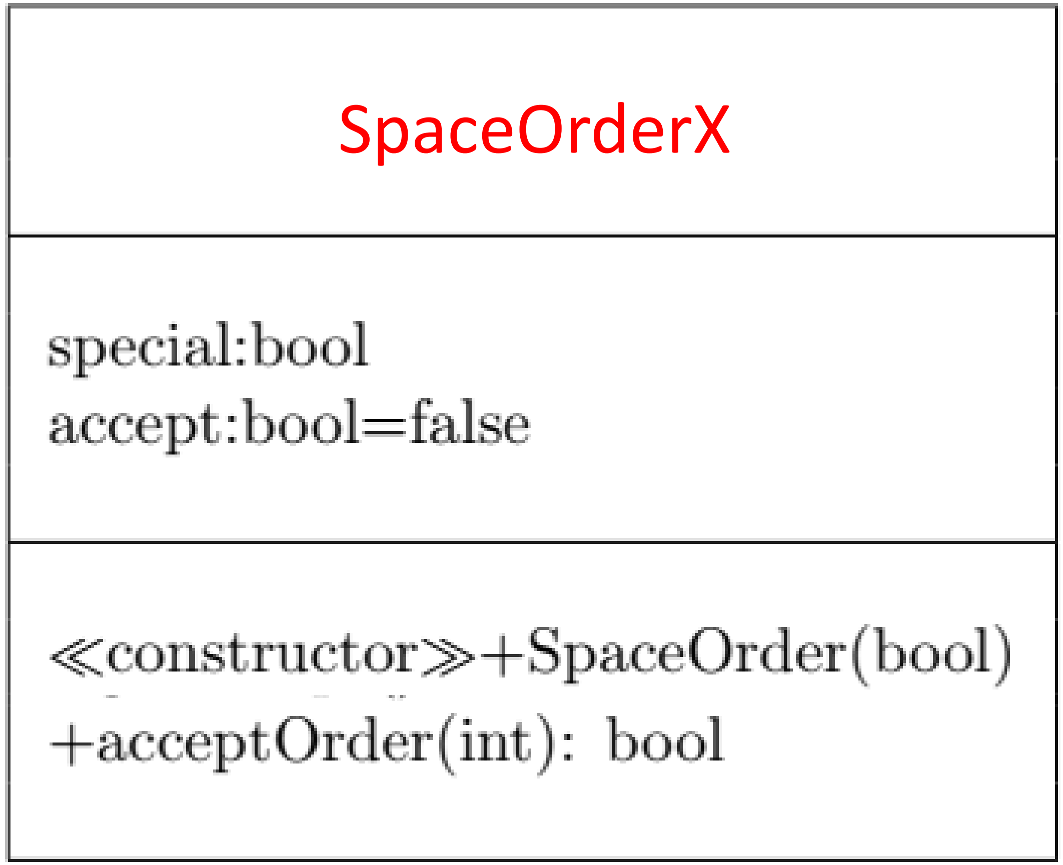
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

1. TESTING WITHOUT GETTERS

- Without getters, difficult to check attribute values after a method call
- Options:
  a) Modify the source code for test purposes
  b) Direct access to attributes in test code
  c) Tests in the source code
  d) Java Reflection
     - allows access to attributes by name



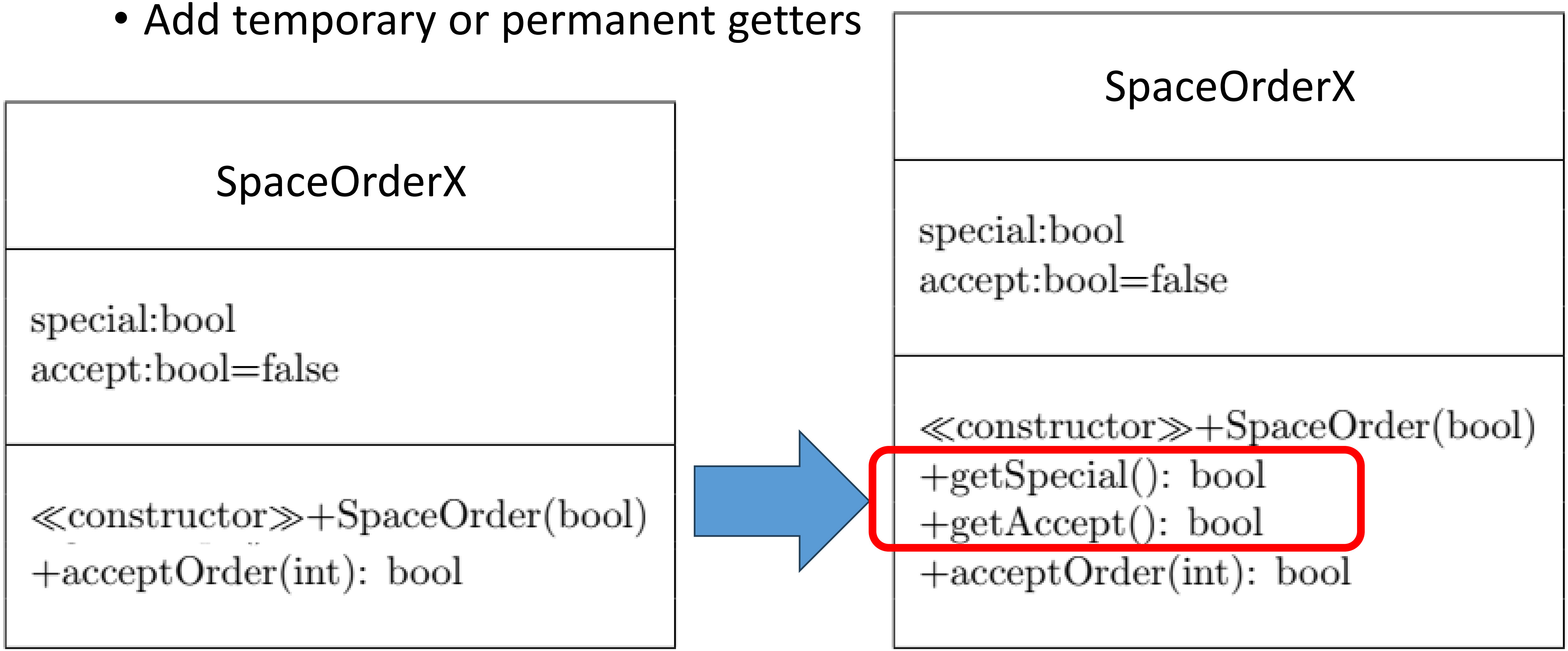
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

a) MODIFY THE SOURCE CODE

- Add temporary or permanent getters



SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+acceptOrder(int): bool

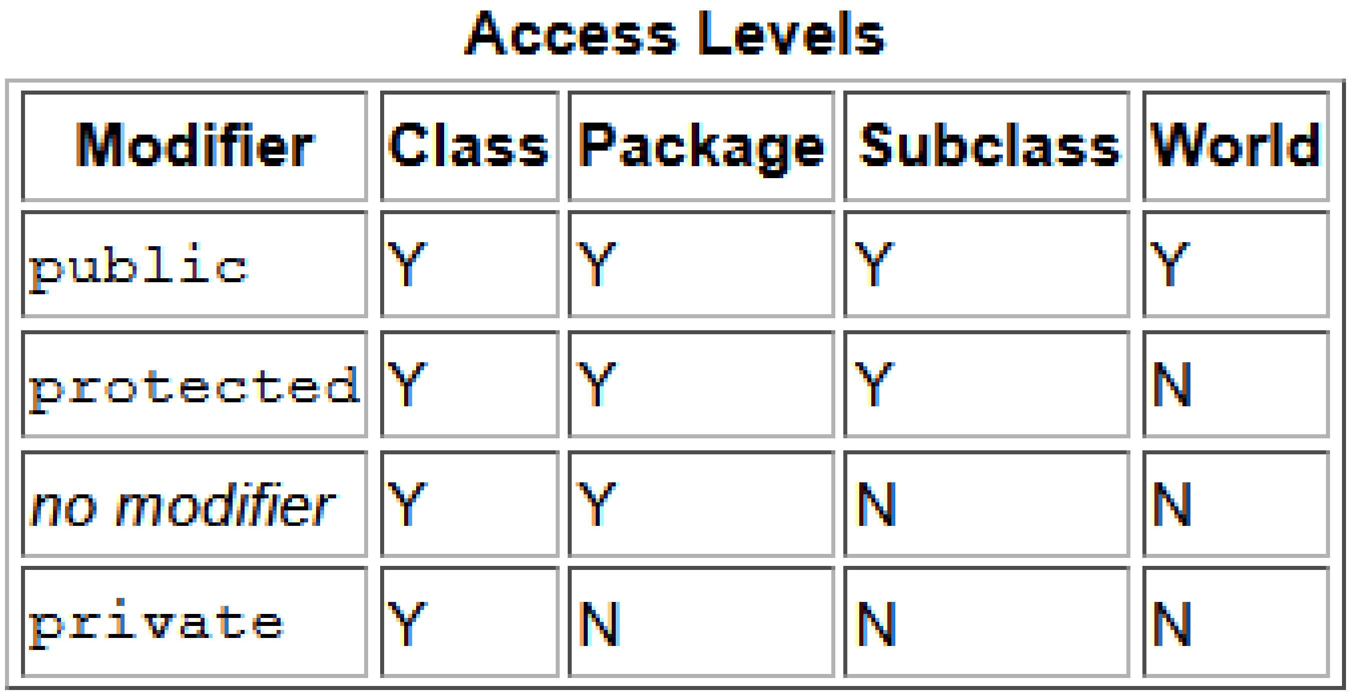
SpaceOrderX

special:bool
accept:bool=false

≪constructor≫+SpaceOrder(bool)
+getSpecial(): bool
+getAccept(): bool
+acceptOrder(int): bool

b) DIRECT ACCESS

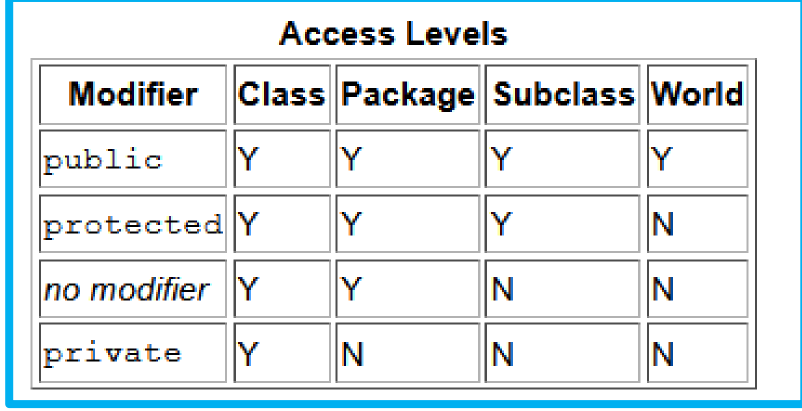
- Depends on the **access modifiers** for the attributes



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

DIRECT ACCESS



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

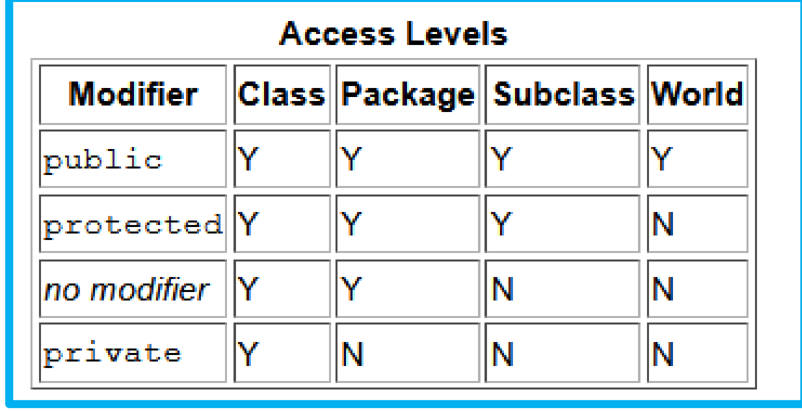
- Depends on the **access modifiers** for the attributes
  - public: no restrictions
  - protected: test must be in same class, package, or subclass
  - default: test must be in same class or package
  - private: test must be in same class

```
public class SpaceOrderX {

   boolean special;
   boolean accept=false;

}
```

Test in an Accessible Class



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

- **public**, **protected**, or **default**
- Access the attribute by name
- Not ideal – assumes that the attribute is simply stored
- Requires interpretation of source code
- Tests may fail if the internal representation of an attribute is changed

Example

```
package cs608;
import static org.testng.Assert.*;
import org.testng.annotations.*;

public SpaceOrderXTest() {}

   @DataProvider(name="constructorData")
   public Object[][] getConstructorData() {
      return new Object[][] {
         // TID,  special, e_special
         {  "T1",    true,      true},
         {  "T2",   false,     false},
      };
   }
   @Test(dataProvider="constructorData")
   public void testConstructor(String tid,
         boolean special, boolean expectedSpecial)
   {
      SpaceOrderX o = new SpaceOrderX(special);
      assertEquals( o.special, expectedSpecial );
   }
}
```

How it May Fail

- Perhaps in the original code, special was stored as a boolean as shown

```
public class SpaceOrderX {

   protected boolean special;
   protected boolean accept=false;

   public SpaceOrderX(boolean isSpecial) {
      special = isSpecial;
   }

   // code hidden here

}
```

How it May Fail

- But then refactored as an enum
- With a different attribute name: type
- This is quite valid: it still matches the UML design
    - The implementation of attributes must be "compatible with" the design
- And the API is unchanged

```
package cs608;

public class SpaceOrderY {

   enum OrderType {SPECIAL, NORMAL};

   OrderType type;

   public SpaceOrderY(boolean isSpecial) {
      if (isSpecial) type=OrderType.SPECIAL;
      else type=OrderType.NORMAL;
   }

   public boolean acceptOrder(int space) {

}
```

But the Test Fails to Compile

```
$javac-d bin -cp libraries-win\* SpaceOrderYTest.java
SpaceOrderYTest.java:25: error: cannot find symbol
      assertEquals( o.special, expectedSpecial );
                     ^
  symbol:   variable special
  location: variable o of type SpaceOrderY
1 error
```

Another Example: Tests Fail Incorrectly

- Temperature class:
  - setTemp(int degreesC)
  - getTemp(int degreesC)
  - And attribute temp in degrees Centigrade
- The class is refactored:
  - setTemp(int degreesC)
  - getTemp(int degreesC)
  - And attribute temp in degrees Farenheit

Another Example: Tests Fail Incorrectly

- Temperature class:
  - setTemp(int degreesC)
  - getTemp(int degreesC)
  - And attribute temp in degrees Centigrade
- The class is refactored:
  - setTemp(int degreesC)
  - getTemp(int degreesC)
  - And attribute temp in degrees Farenheit
- The API is the same
- The tests all compile
- But accessing temp directly produces incorrect test failures

(c) Tests in the Same Class

- Not desirable – don't want to include tests in the final product
- Use text manipulation tools to auto-embed the test code (include statements, annotations, data providers, test methods) in the class for testing
  - Create a copy of the code
  - Run a tool to auto-insert the tests into the classes
  - Run the tests, using the class as the test class
  - Note: TestNG requires a constructor with no parameters
- Or use BIT (built-in-testing) which can be disabled at runtime
  - Discussed in next lecture

Example: source code

```java
package cs608;

public class SpaceOrderX {

   protected boolean special;
   protected boolean accept=false;

   public SpaceOrderX(boolean isSpecial) {
      // Code not shown
   }

   public boolean acceptOrder(int space) {
      // Code not shown
   }

   public boolean getAccept() {
      // Code not shown
   }
}
```

Example: constructor test code

```
import static org.testng.Assert.*;
import org.testng.annotations.*;

   public SpaceOrderX() {}

   @DataProvider(name="constructorData")
   public Object[][] getConstructorData() {
      return new Object[][] {
         // TID,  special, e_special
         {  "T1",    true,      true},
         {  "T2",   false,     false},
      };
   }

   @Test(dataProvider="constructorData")
   public void testConstructor(String tid,
         boolean special, boolean expectedSpecial)
   {
      SpaceOrderX o = new SpaceOrderX(special);
      assertEquals( o.special, expectedSpecial );
   }
```

Example: class with tests auto-inserted

```java
package cs608;

import static org.testng.Assert.*;
import org.testng.annotations.*;

public class SpaceOrderX {

   protected boolean special;
   protected boolean accept=false;

   public SpaceOrderX() {}

   public SpaceOrderX(boolean isSpecial) {
      // Code not shown
   }

   public boolean acceptOrder(int space) {
      // Code not shown
   }

   public boolean getAccept() {
      // Code not shown
   }

   @DataProvider(name="constructorData")
   public Object[][] getConstructorData() {
      return new Object[][] {
         // TID,  special, e_special
         {  "T1",    true,      true},
         {  "T2",   false,     false},
      };
   }

   @Test(dataProvider="constructorData")
   public void testConstructor(String tid, boolean special, boolean expectedSpecial) {
      SpaceOrderX o = new SpaceOrderX(special);
      assertEquals( o.special, expectedSpecial );
   }
}
```

Example: Test Execution

```
$java -cp libraries-win\*;bin org.testng.TestNG –testclass cs608.SpaceOrderX

PASSED: testConstructor("T1", true, true)
PASSED: testConstructor("T2", false, false)

===============================================
    Command line test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Command line suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

(d) JAVA REFLECTION

- You can access object attributes at runtime using Java Reflection

Original Test

```
import static org.testng.Assert.*;
import org.testng.annotations.*;

   public SpaceOrderX() {}

   @DataProvider(name="constructorData")
   public Object[][] getConstructorData() {
      return new Object[][] {
         // TID,  special, e_special
         {  "T1",    true,       true},
         {  "T2",   false,      false},
      };
   }

   @Test(dataProvider="constructorData")
   public void testConstructor(String tid,
         boolean special, boolean expectedSpecial)
   {
      SpaceOrderX o = new SpaceOrderX(special);
      assertEquals( o.getSpecial(), expectedSpecial );
   }
```

Focus: original test method

```
   @Test(dataProvider="constructorData")
   public void testConstructor(String tid,
         boolean special, boolean expectedSpecial)
   {
      SpaceOrderX o = new SpaceOrderX(special);
      assertEquals( o.getSpecial(), expectedSpecial );
   }
```

Test method with reflection

```
@Test(dataProvider="constructorData")
public void testConstructor(String tid,
                            boolean special,
                            boolean expectedSpecial)
{
   SpaceOrderX o = new SpaceOrderX(special);
   assertEquals((boolean)readAtt(o,"special"),expectedSpecial);
}
```

Implementation of readAtt()

```
// Test helper methods

// Read a non-public attribute using reflection

private Object readAtt(Object o, String attName) {
   try {
      Class c = o.getClass();
      Object v = c.getDeclaredField(attName).get(o);
      return v;
   } catch (Exception ex) {
      return null; // will cause test to fail
   }
}
```

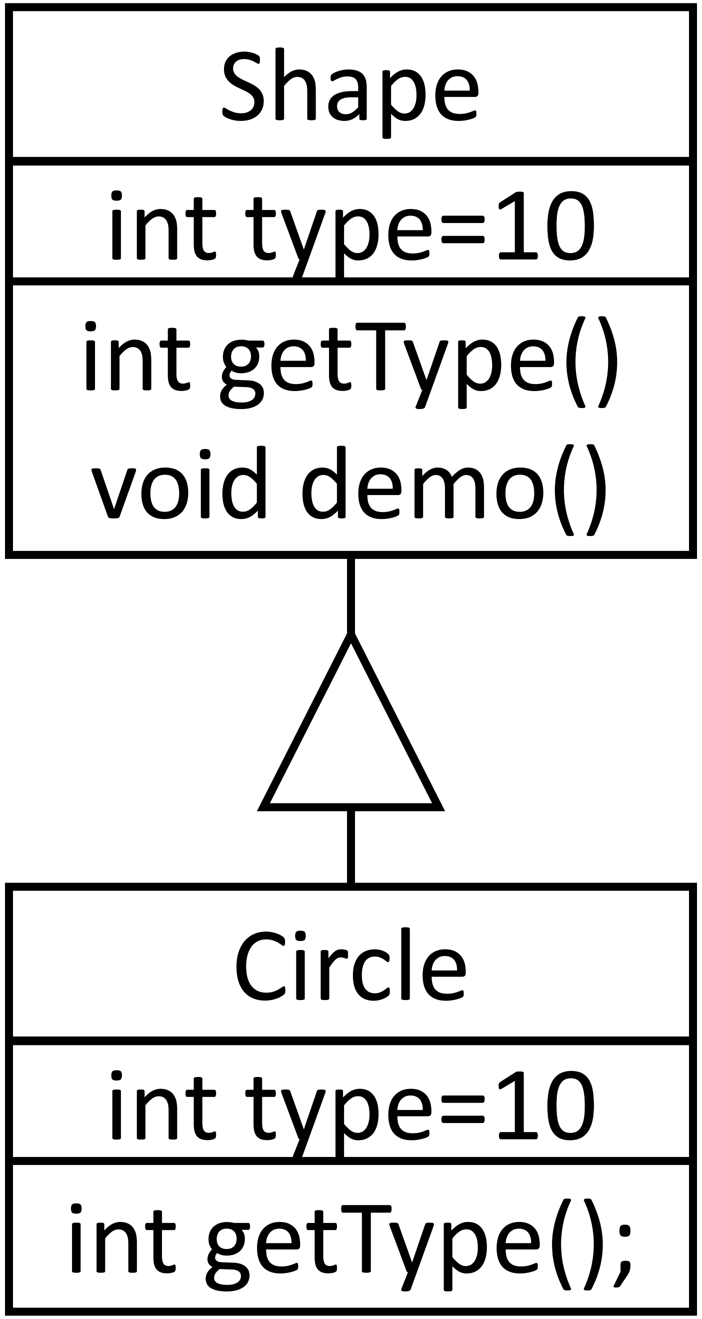
2. INHERITANCE

2. INHERITANCE PITFALLS

- Not every language implements inheritance in exactly the same way
- There are many general OO hazards
- And many language-specific language hazards
- We will look at one:
  - Accessing **methods** and **attributes** from an inherited method
- This is why OO inheritance testing is important
  - Inexperienced programmers can get "unexpected" results

Class Circle extends Class Shape

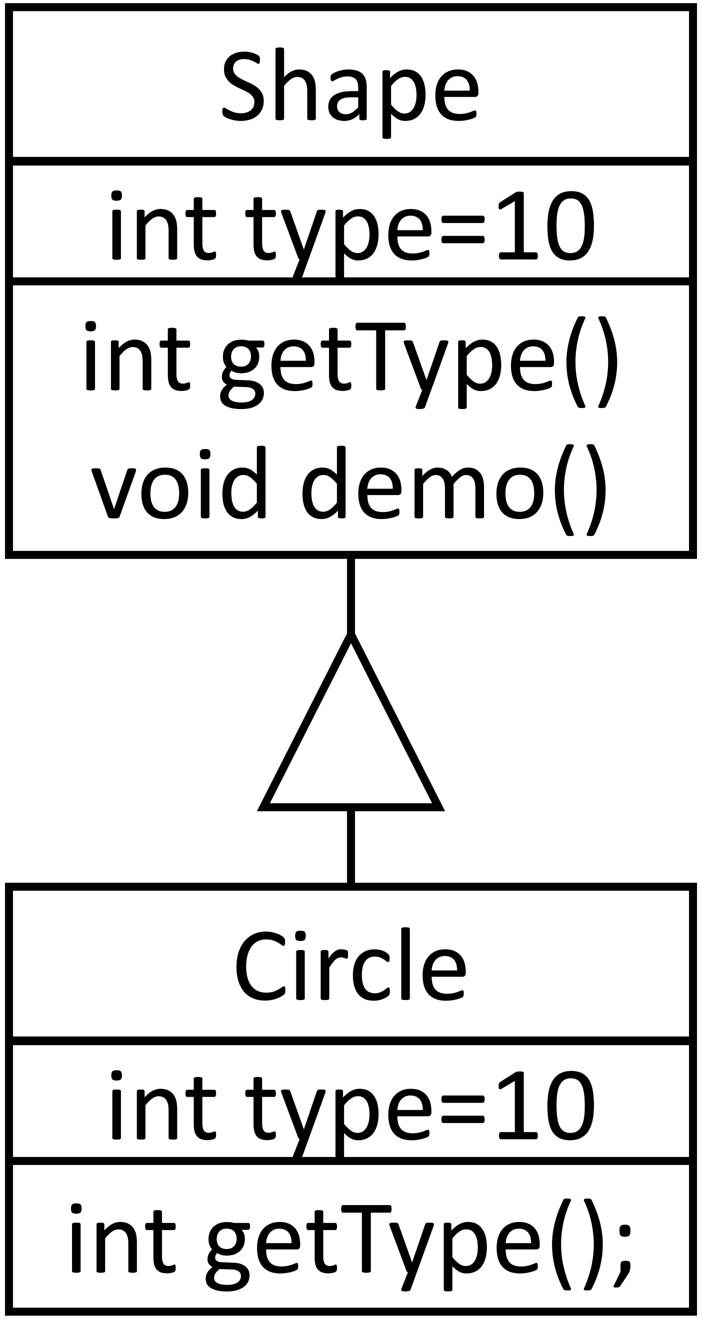
- Using Shape
  - s=new Shape()
  - s.getType() calls Shape.getType()
  - s.demo() calls Shape.demo()



Class Circle extends Class Shape

- Using Shape
  - s=new Shape()
  - s.getType() calls Shape.getType()
  - s.demo() calls Shape.demo()

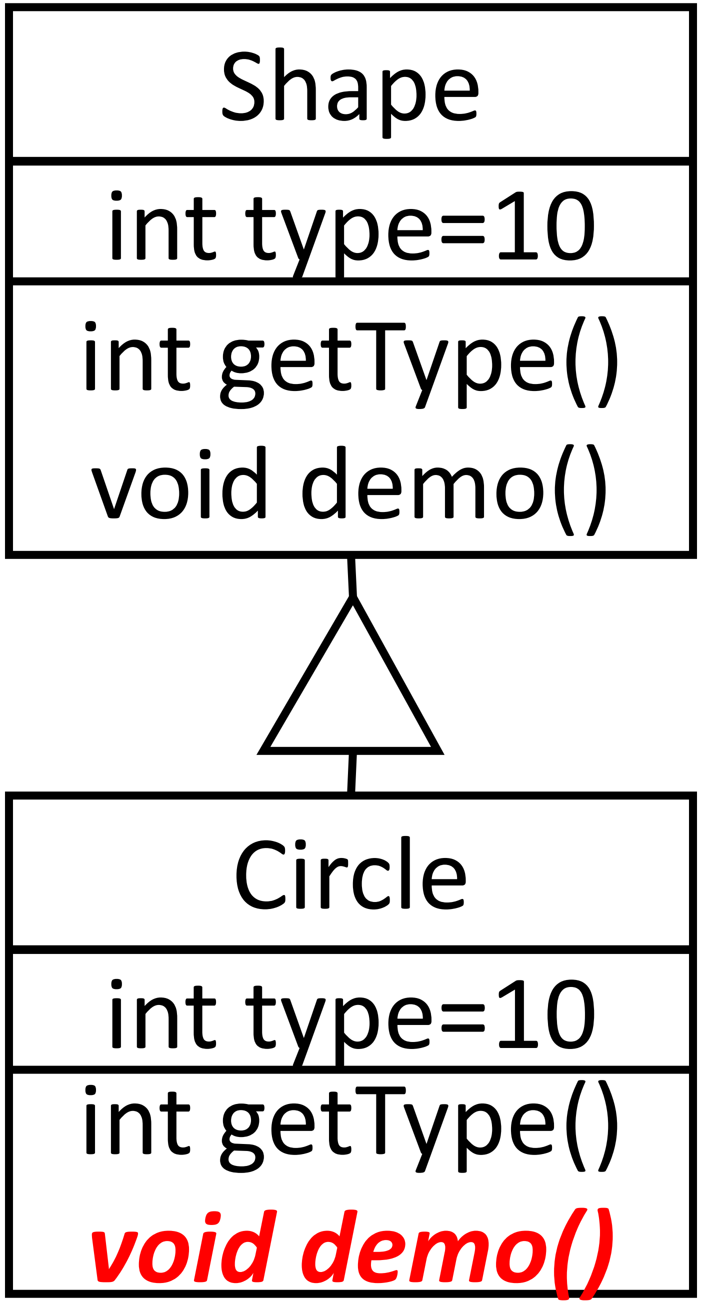
- Using Circle
  - c=new Circle()
  - c.getType() calls Circle.getType()
  - c.demo() calls Shape.demo() (inherited) in "**Circle Context**"



Class Circle extends Class Shape

- Using Shape
  - s=new Shape()
  - s.getType() calls Shape.getType()
  - s.demo() calls Shape.demo()

- Using Circle
  - c=new Circle()
  - c.getType() calls Circle.getType()
  - c.demo() calls Shape.demo() (inherited) in "**Circle Context**"



Shape

int type=10

int getType()

void demo()

Circle

int type=10

int getType()

void demo()

Class Shape

```
class Shape {
   int type=10;
   int getType()
   {
      return type;
   }
   void demo() {
       System.out.println("   getType() returns "+getType());
       System.out.println("   type equals "+type);
   }
}
```

What happens When you call s.demo()?

```
class Shape {
   int type=10;
   int getType()
   {
      return type;
   }
   void demo() {
       System.out.println("   getType() returns "+getType());
       System.out.println("   type equals "+type);
   }
}
```

What happens When you call s.demo()

```
$java DemoShape
Executing class cs608.Shape.demo()
   getType() returns 10
```

```
void demo() {
       System.out.println("   getType() returns "+getType());
       System.out.println("   type equals "+type);
}
```

What happens When you call s.demo()

```
$java DemoShape
Executing class cs608.Shape.demo()
   getType() returns 10
   type equals 10
```

```
void demo() {
   System.out.println("   getType() returns "+getType());
   System.out.println("   type equals "+type);
}
```

What happens When you call c.demo()?

```
class Circle extends Shape {
    int type=20;
    int getType()
    {
        return type;
    }
}
```

What happens When you call c.demo()?

```
class Circle extends Shape {
    int type=20;
    int getType()
    {
        return type;
    }
}
```

```
void demo() {
    System.out.println("   getType() returns "+getType());
    System.out.println("   type equals "+type);
}
```

What happens When you call c.demo()

```
$java DemoCircle
Executing class cs608.Circle.demo()
```

What happens When you call c.demo()

```
$java DemoCircle
Executing class cs608.Circle.demo()
   getType() returns 20
```

What happens When you call c.demo()

```
$java DemoCircle
Executing class cs608.Circle.demo()
   getType() returns 20
   type equals 10
```

Why?

- When method calls are invoked, the Java VM works its way up the inheritance stack from the **current** class to find a matching method
- And executes that
  - Shape.demo() calls Shape.demo()
  - Shape.demo() invokes getType() which calls Shape.getType()
  - Circle.demo() calls Shape.demo()
  - Shape.demo() in "Circle context" invokes getType() which calls Circle.getType()
- BUT attributes are accessed directly
  - Shape.getType() accesses Shape.type
  - Circle.getType() accesses Circle.type

Why?

- A superclass can invoke methods in a subclass when called from "subclass context"
- But a superclass cannot access subclass attributes, even when called from "subclass context"

Why?

- A superclass can invoke methods in a subclass when called from "subclass context"
  - So when demo() is called on a Circle object, it calls **method** Circle.getType()
- But a superclass cannot access subclass attributes, even when called from "subclass context"
  - So when demo() is called on a Circle() object, it accesses **attribute** Shape.type

Implications for Testing

- You need to make sure that inherited methods work correctly
- They may behave differently depending on whether the coder has used attributes or getters
- So, in this example, demo() works differently depending on whether the coder has used **type** or **getType()**
    - Which is correct depends on the specification for demo()