INF101: Object-Oriented Programming

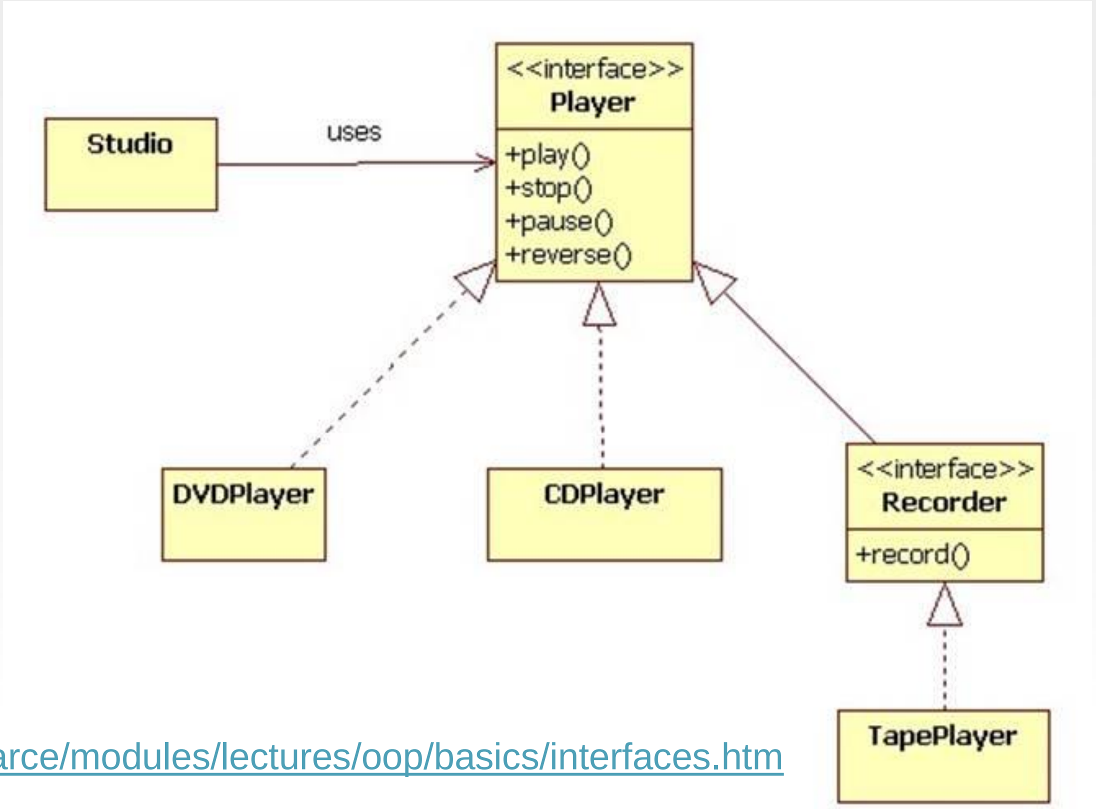
Lecture 13: Class Invariance

UNIVERSITY OF BERGEN

Welcome to INF101 – Lecture 13!

- Previously:
  - Inheritance part 2
  - Pre- and Post-conditions
  - Liskov Substitution Principle

Interfaces + Inheritance



http://www.cs.sjsu.edu/~pearce/modules/lectures/oop/basics/interfaces.htm

Inheritance in a nutshell

- Subclass inherits from superclass: all fields (state) and methods (behavior)
- You can use an object of the sub-class any place you expect a super-class object
- When you make a sub-class, its constructor must call the super-class constructor using **super** keyword
- Only one superclass allowed per class

Inheritance + Interfaces

- Remember: sub-classes get all fields and methods (including code) of the super-class that they extend
- The sub-class then also implements any interfaces that the super-class does

Interfaces, once more

- Implementing classes have to implement all interface methods
- Implementing classes also implement all the interfaces that an interface extends (inherits from)
- Many classes can implement same interface
- Interfaces don’t specify a constructor, fields, or method code
- You cannot make objects from an interface

Pre-condition

- Something that must be true before you can call a method
- Specifies what method arguments are valid
- Breaking the precondition typically results in an IllegalArgumentException (or another exception)
- “What I expect from you”

Precondition

- Best to check arguments yourself, and throw an appropriate exception if not met
- Preferably right at the beginning of the method
- Important to document preconditions with JavaDoc

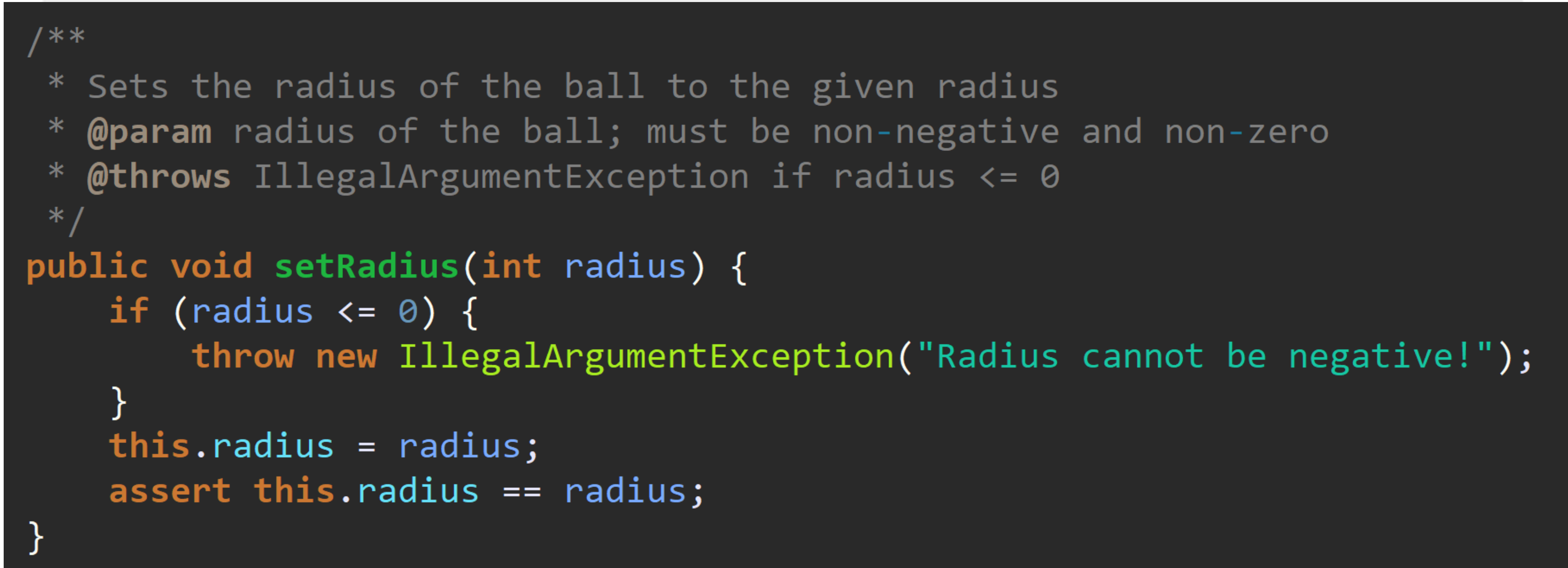
```java
/**
 * Sets the radius of the ball
 * @param radius of the ball; must be non-negative and non-zero
 * @throws IllegalArgumentException if radius <= 0
 */
public void setRadius(int radius) {
    if (radius <= 0) {
        throw new IllegalArgumentException("Radius cannot be negative!");
    }
    this.radius = radius;
}
```

Post-condition

- What the method guarantees if the pre-condition is satisfied
- Things that are true after method is complete
- “What I promise to do for you”

Post-conditions

- Can be checked using assert



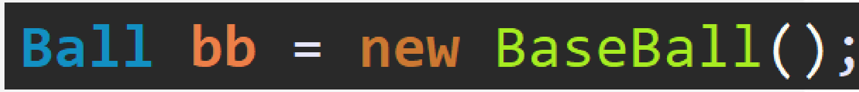
```
/**
 * Sets the radius of the ball to the given radius
 * @param radius of the ball; must be non-negative and non-zero
 * @throws IllegalArgumentException if radius <= 0
 */
public void setRadius(int radius) {
    if (radius <= 0) {
        throw new IllegalArgumentException("Radius cannot be negative!");
    }
    this.radius = radius;
    assert this.radius == radius;
}
```

Pre- and Post-conditions

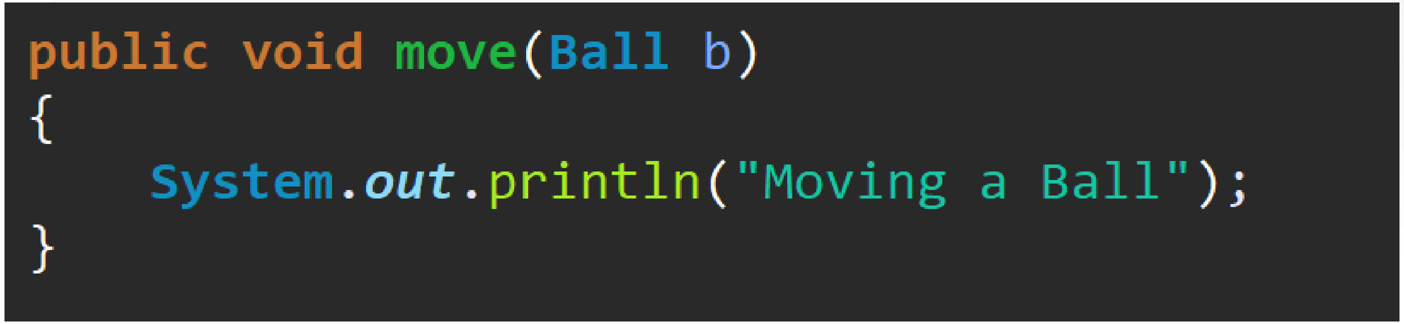
- More reading: https://docs.oracle.com/cd/E19683-01/806-7930/assert-13/index.html

The Liskov Substitution Principle

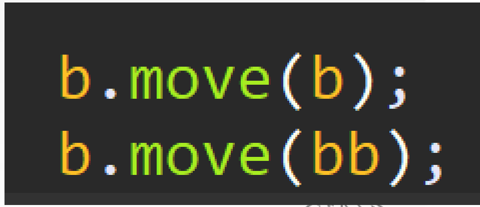
- Functions that use superclasses must be able to use objects of subclasses without knowing it



```java
Ball bb = new BaseBall();
```



```java
public void move(Ball b)
{
    System.out.println("Moving a Ball");
}
```



```java
b.move(b);
b.move(bb);
```

LSP

- Ability to replace any instance of a superclass with an instance of one of its subclasses without negative side effects

Today

- Invariance

Questions?

Invariance

Invariance

- Invariant: an expression that is the same every time it is calculated

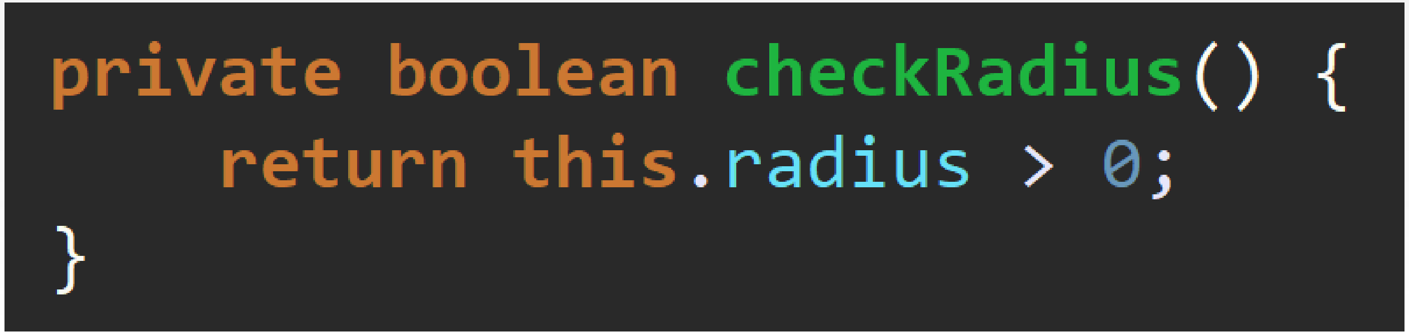
- Class Invariant: something that is always true for an object of a certain class

Class Invariant

- an expression that will be true every time a method is called or returns a value
- Class invariants indicate if an object is consistent or not (has valid field variables)
- Class invariants should be fulfilled as a pre-condition to all public methods
- All public methods have to maintain class invariance

How to check?

- We can make a method that checks class invariance
- For instance if our class invariant is that a ball radius is always positive:



```java
private boolean checkRadius() {
    return this.radius > 0;
}
```

How to check?

- Call this method at the beginning and end (before return) of every public method to assert class invariant:

```java
public void setRadius(int radius) {
    if (radius <= 0) {
        throw new IllegalArgumentException("Radius cannot be negative!");
    }
    assert this.checkRadius();
    this.radius = radius;
    assert this.radius == radius;
    assert this.checkRadius();
}
```

Well that’s a bit much

- Calling this method at the beginning of every public method to assert class invariant = overkill
- Generally not needed, unless state (fields) are directly modifiable by other classes (public)
- Another good reason to keep your fields private ☺
- More: https://docs.oracle.com/cd/E19683-01/806-7930/assert-13/index.html

How can we break class invariants?

- Invariants check validity of object’s state (data)
- This state may become invalid, for example by:
  - Invalid argument passed by method or constructor caller (should check in pre-condition!)
  - The implementation is defective (should check with assert statement before return)

Ball example again

In practice

- Document what the class invariants are (Javadoc)
- Useful: check class invariants before you return from the constructor and public methods to discover inconsistent objects as early as possible
- Remember: As long as field variables are private, only the class methods can change them (so only need to check own public methods to see if class invariant holds)

Summary

- **Precondition**: A condition that the caller of an operation agrees to satisfy
- **Postcondition**: A condition that the method itself promises to achieve
- **Invariant**: A condition that a class must satisfy anytime a client could invoke an object's method, or a condition that should always be true for a specified segment or at a specified point of a program

Questions?

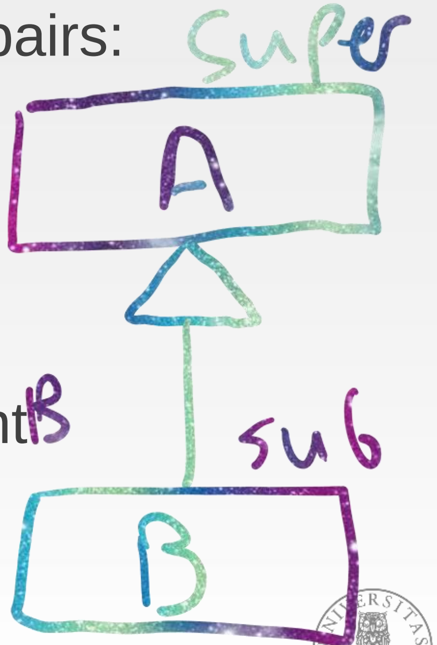
Break

Exercise

- Identify superclass and subclass for these pairs:
  - **a.** Employee, Manager
  - **b.** GraduateStudent, Student
  - **c.** Person, Student
  - **d.** Employee, Professor
  - **e.** BankAccount, CheckingAccount
  - **f.** Vehicle, Car
  - **g.** Vehicle, Minivan
  - **h.** Car, Minivan
  - **i.** Truck, Vehicle

Exercise

- Identify superclass and subclass for these pairs:
  - **a.** Employee, A Manager B
  - **b.** GraduateStudent, B Student A
  - **c.** Person, A Student B
  - **d.** Employee, A Professor B
  - **e.** BankAccount, A CheckingAccount B
  - **f.** Vehicle, A Car B
  - **g.** Vehicle, A Minivan B
  - **h.** Car, A Minivan B
  - **i.** Truck, B Vehicle A



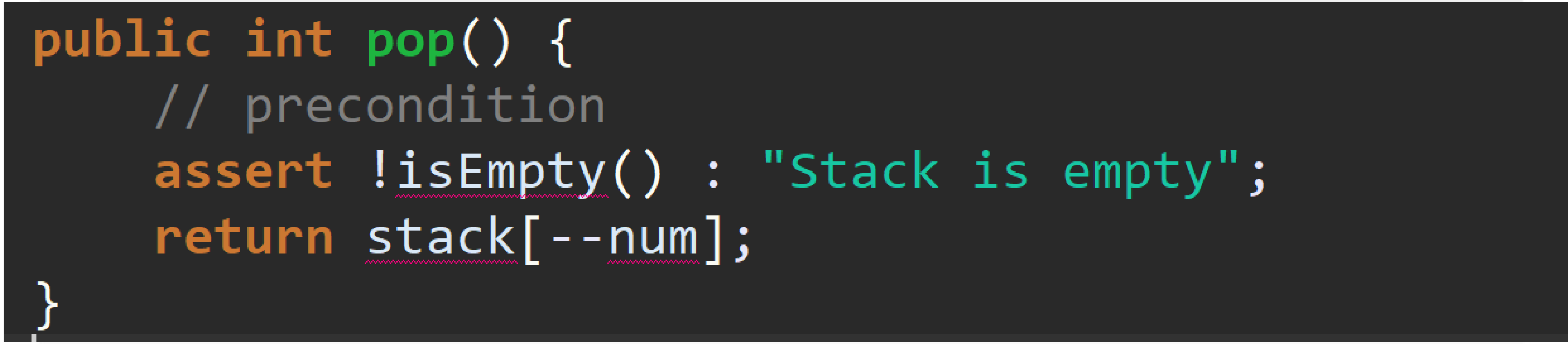
Another example: a stack of ints

- We can push things onto the stack
- We can pop things off of the stack

(example from http://www.oracle.com/us/technologies/java/assertions-139853.html)

Pop

- If we want to get an item from the stack, it should not be empty (precondition)

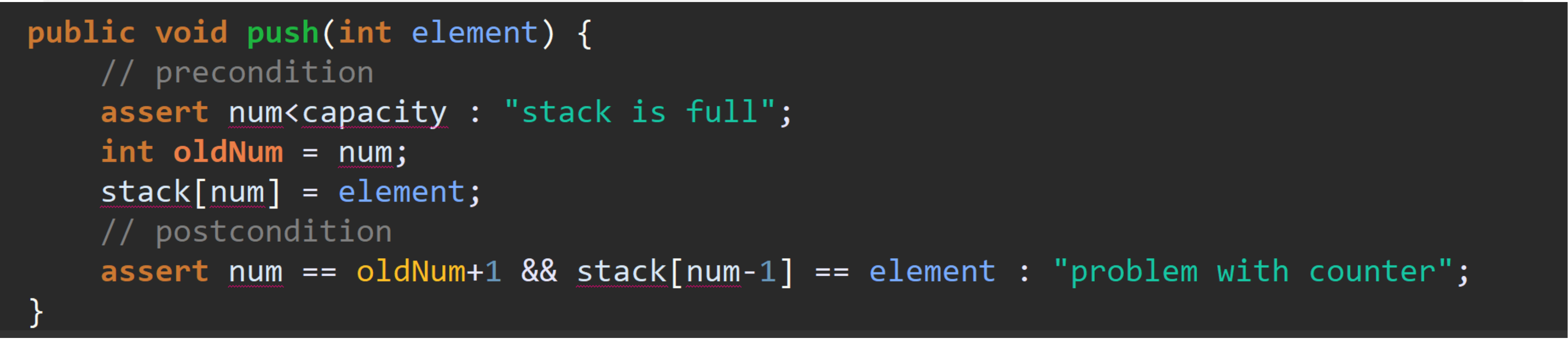


```
public int pop() {
    // precondition
    assert !isEmpty() : "Stack is empty";
    return stack[--num];
}
```

- Actually throwing an exception would be more helpful since assert can be disabled!

Push

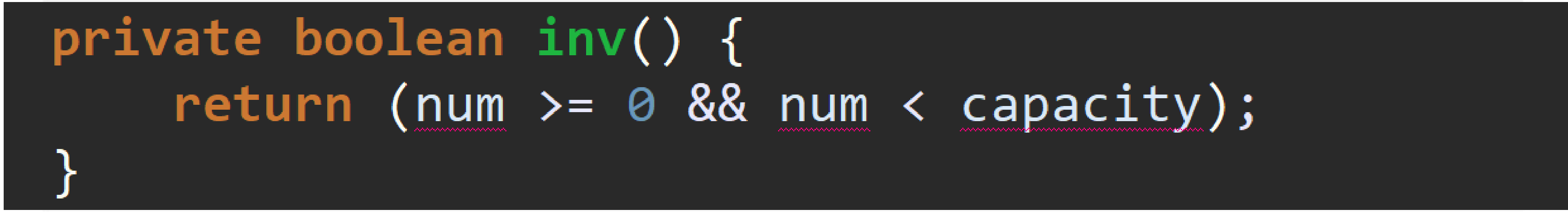
- If we want to push onto the stack, it should not be full (precondition), but also, the new index is the old index + 1, and the new element should be added (postconditions)



```
public void push(int element) {
    // precondition
    assert num<capacity : "stack is full";
    int oldNum = num;
    stack[num] = element;
    // postcondition
    assert num == oldNum+1 && stack[num-1] == element : "problem with counter";
}
```

Class Invariant

- For the stack, the number of elements in the stack should be greater than or equal to zero, and the number of elements should not be greater than the maximum capacity



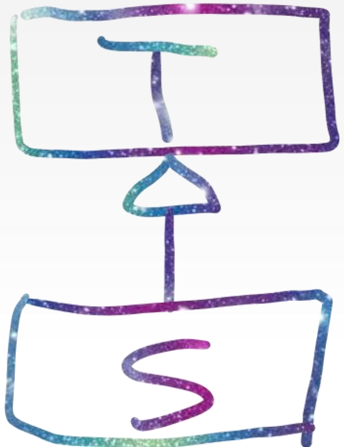
```
private boolean inv() {
    return (num >= 0 && num < capacity);
}
```

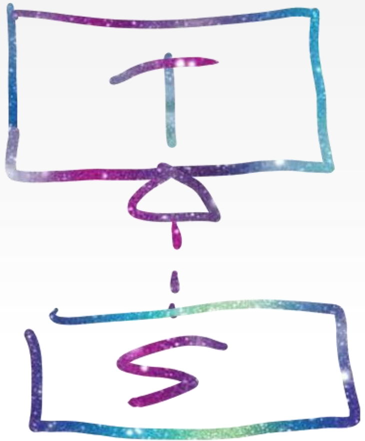
- Assert before every public method and constructor return

Back to Liskov substitution principle

Substitutability

- If S is a subtype of T, objects of type T may be replaced/substituted with objects of type S and it will work just as well
- For both inheritance and interfaces



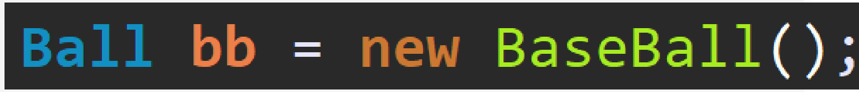


In practice

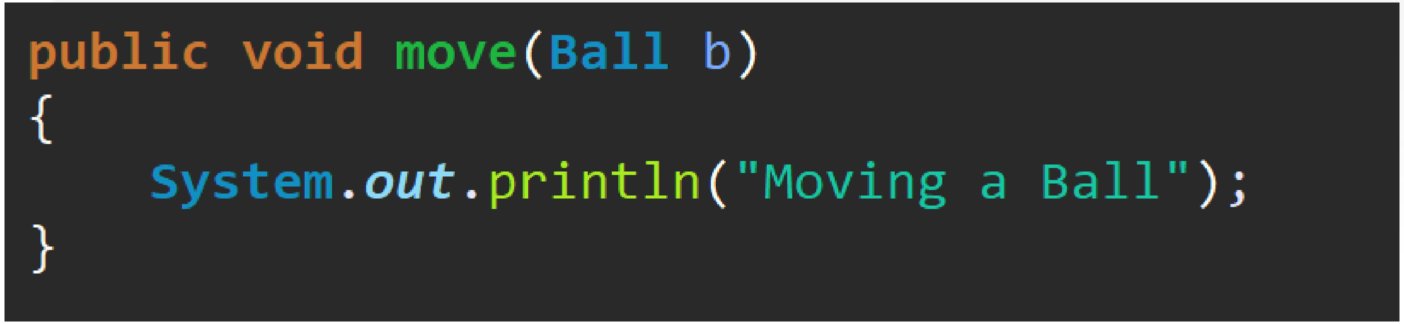
- Pre-conditions in the subclass can not be stronger than pre-conditions in the superclass (but can be weaker)
- Behavior: subclass has to do at least what the superclass does, so able to pass all the tests for the superclass

The Liskov Substitution Principle

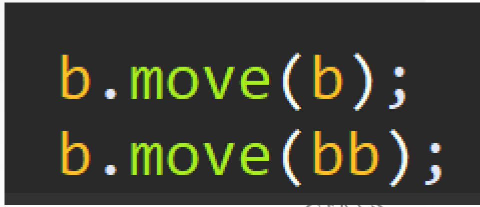
- Functions that use superclasses must be able to use objects of subclasses without knowing it



```java
Ball bb = new BaseBall();
```



```java
public void move(Ball b)
{
    System.out.println("Moving a Ball");
}
```



```java
b.move(b);
b.move(bb);
```

LSP

- Ability to replace any instance of a superclass with an instance of one of its subclasses without negative side effects

How does this relate to class invariant?

- All methods (including constructor) in the subclass have to maintain the class invariant of the superclass
- If the subclass has an own class invariant which is stronger than the superclass, all methods (including inherited!) must uphold this. If the inherited do not: have to @override

Example

- Class Bunny inherits from Animal, and has an extra field: teethlength
- If teethlength >= 0, then Bunny class invariant = Animal class invariant + teethlength >= 0
- All methods in Animal don’t know about teethlength, so are safe

Example continued

- But if we have a special relation between teeth and weight (from Animal), we have to override all methods to maintain this:
  - The eat() method in Animal can change the weight without adjusting the Bunny teethlength

Exercise

In an object-oriented traffic simulation system, we have the classes listed below.

Draw an inheritance diagram that shows the relationships between these classes.

- Vehicle
- Car
- Truck
- Sedan
- Coupe
- PickupTruck
- SportUtilityVehicle
- Minivan
- Bicycle
- Motorcycle

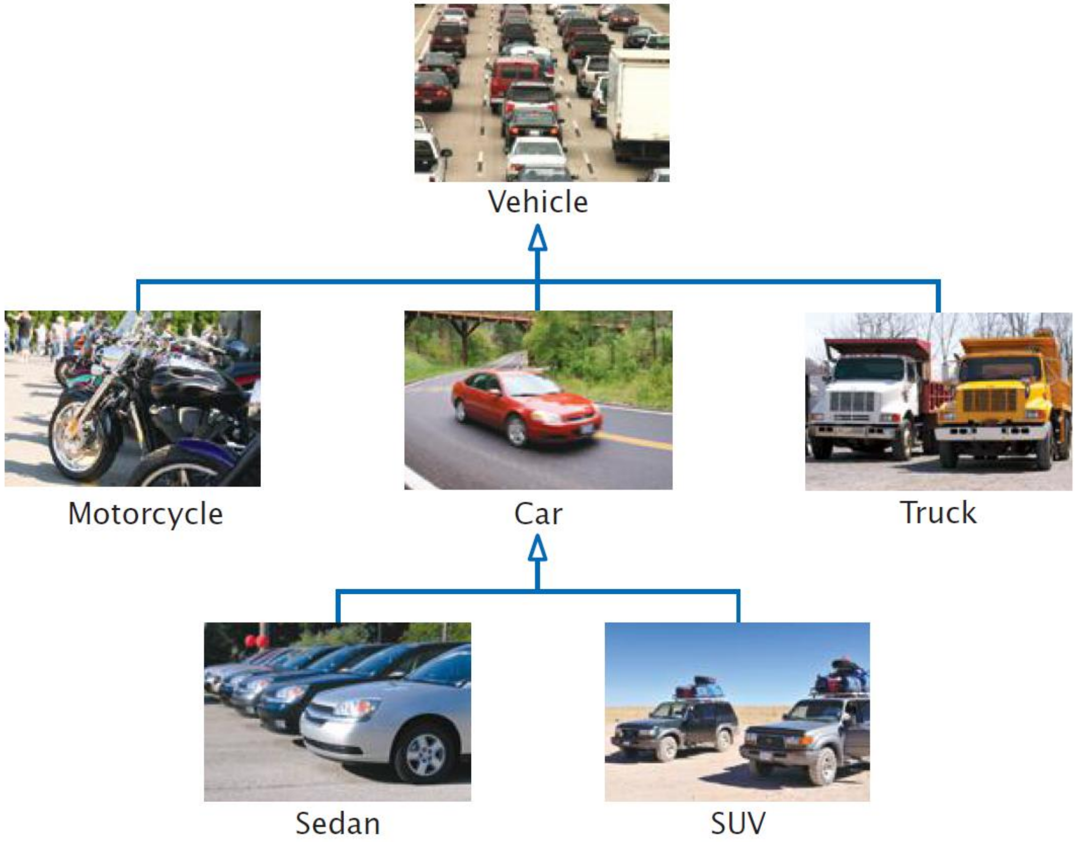
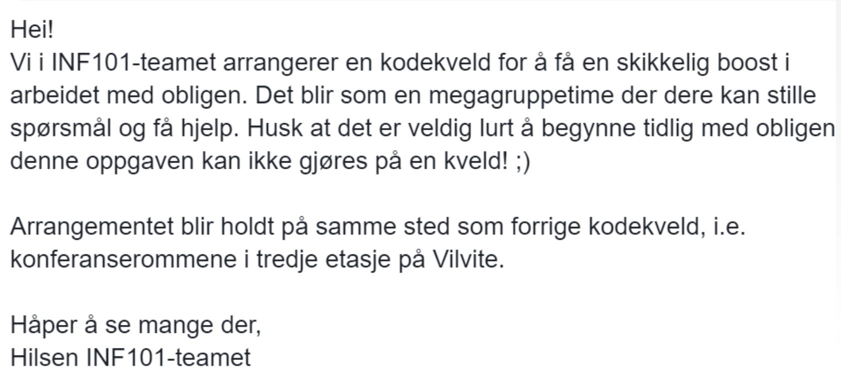


Figure 1 An Inheritance Hierarchy of Vehicle Classes

Questions?

Wednesday: Kodekveld!

- 6 PM
- Vilvite Koferansrom A/B (C/D maybe)

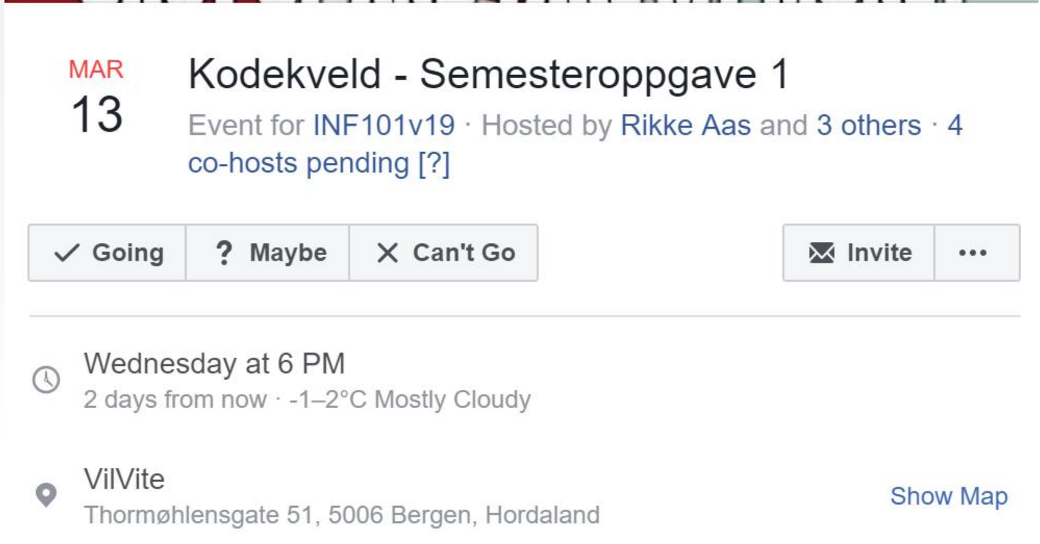


Hei!
Vi i INF101-teamet arrangerer en kodekveld for å få en skikkelig boost i arbeidet med obligen. Det blir som en megagruppetime der dere kan stille spørsmål og få hjelp. Husk at det er veldig lurt å begynne tidlig med obligen denne oppgaven kan ikke gjøres på en kveld! ;)

Arrangementet blir holdt på samme sted som forrige kodekveld, i.e. konferanserommene i tredje etasje på Vilvite.

Håper å se mange der,
Hilsen INF101-teamet

EN HEL KVELD MED GRUPPELEDERNE I INF101?
SHUT UP AND TAKE MY MONEY



MAR
13

Kodekveld - Semesteroppgave 1

Event for INF101v19 · Hosted by Rikke Aas and 3 others · 4 co-hosts pending [?]

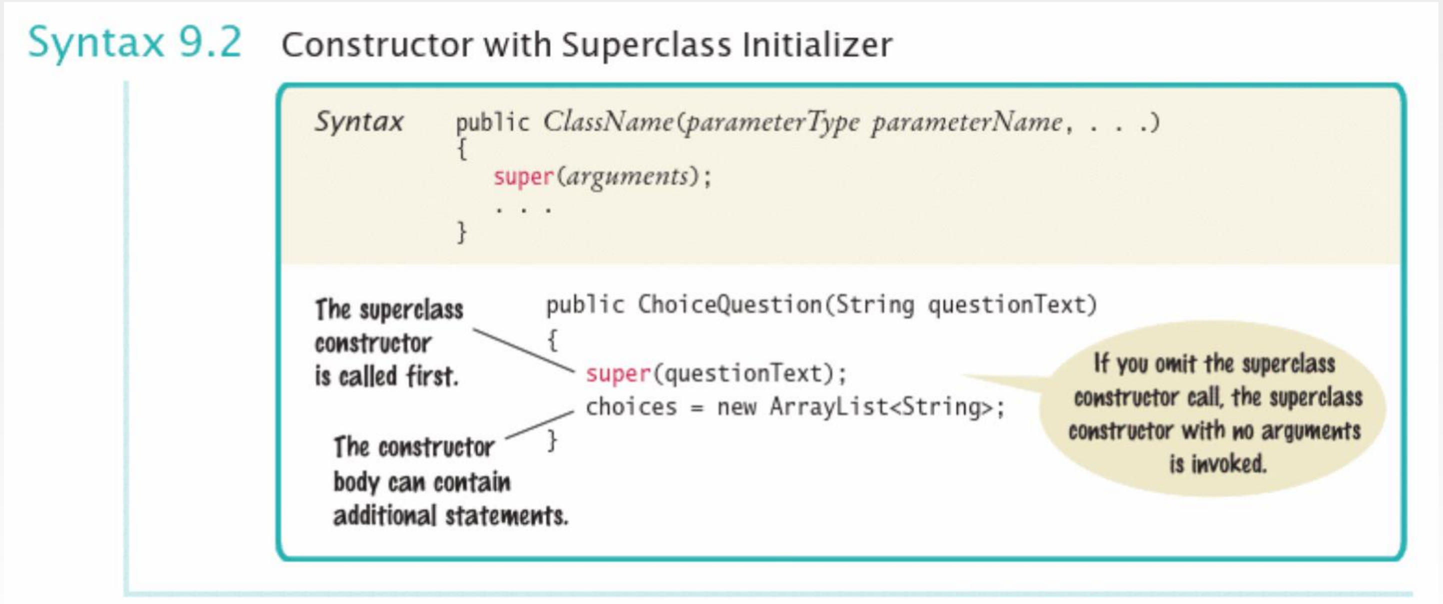
Wednesday at 6 PM
2 days from now · -1–2°C Mostly Cloudy

VilVite
Thormøhlensgate 51, 5006 Bergen, Hordaland

What can you do?

- Work on the labs linked from our wiki (6 labs and semester assignment are up!)
- Summary on inheritance/pre-conditions/Liskov: https://retting.ii.uib.no/inf101/inf101.v19/wikis/arv-forkrav-invariant-substitusjonsprinsippet
- Big Java – Late Objects: Chapter 8: https://ebookcentral-proquest-com.pva.uib.no/lib/bergen-ebooks/reader.action?docID=2055777&query=big+java+late+objects

Big Java – Late Objects: Lots of nice examples!

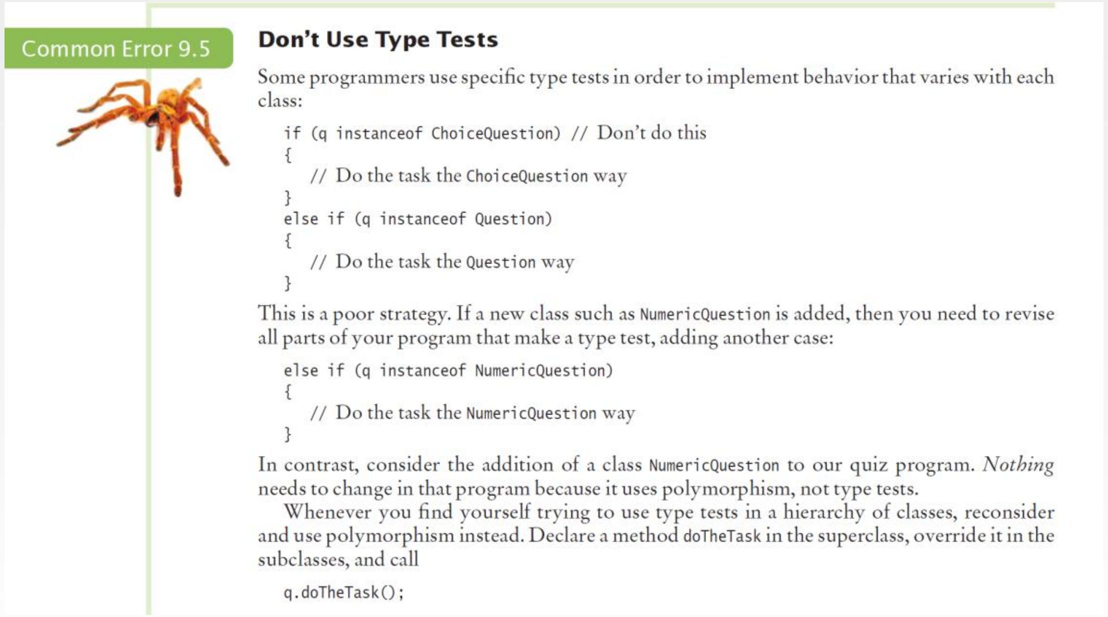

Syntax 9.2 Constructor with Superclass Initializer

Syntax
```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>;
}
```

The superclass constructor is called first.
The constructor body can contain additional statements.
If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

Big Java – Late Objects: Lots of nice examples!



Common Error 9.5

Don't Use Type Tests

Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if (q instanceof ChoiceQuestion) // Don't do this
{
   // Do the task the ChoiceQuestion way
}
else if (q instanceof Question)
{
   // Do the task the Question way
}
```

This is a poor strategy. If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
else if (q instanceof NumericQuestion)
{
   // Do the task the NumericQuestion way
}
```

In contrast, consider the addition of a class `NumericQuestion` to our quiz program. *Nothing* needs to change in that program because it uses polymorphism, not type tests.

Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead. Declare a method `doTheTask` in the superclass, override it in the subclasses, and call

```
q.doTheTask();
```

What’s next?

- Wednesday: Kodekveld
- Next week: no lectures to give you time to complete the compulsory assignment ☺
- Come to the group sessions for help with the practical parts of the course

Thanks!