

Lecture 5

Classes and Objects

Exam preparation

- Last lecture before exam on **Friday 13/12**
 - Recap of things to appear on the exam
 - Send me your suggestions: antonek@chalmers.se
- Exercises on **6/12**: solving old exams together
 - First half: solve exam questions in groups
 - Second half: each group presents their question to the class
- Everyone registered on the course on **Mon. 2/12** will be registered for the exam

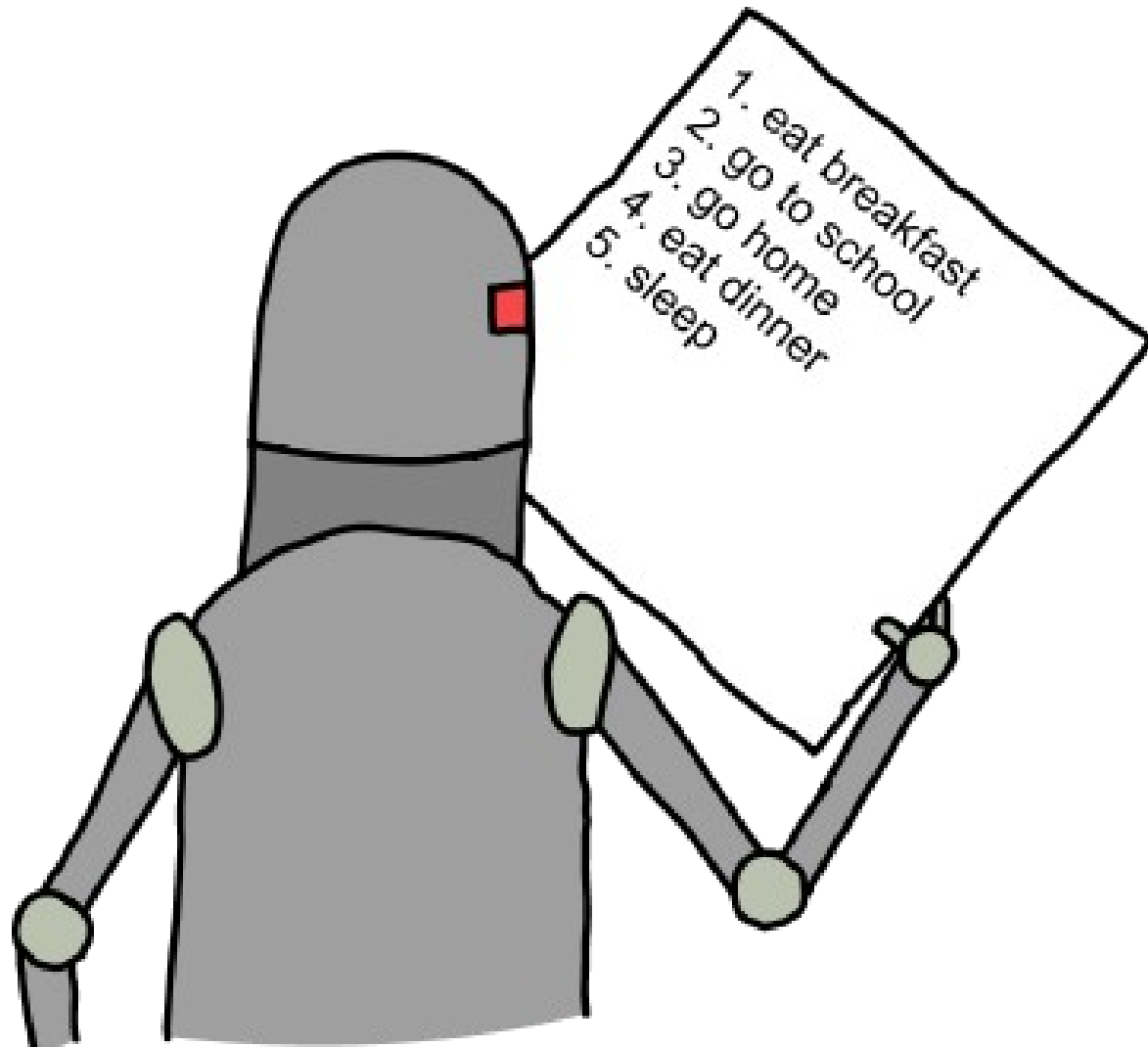
Questions about the material?

- I'm always available during the Friday exercise sessions
- There's also a TA, of course
- Please drop by if you have questions!

Exceptions Recap: A Safer Vector Library

Program as List of Instructions

- Model so far:



Spaghetti Code

```
1 C      A weird program for calculating Pi written in Fortran.
2 C      From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.
3
4      PROGRAM PI
5      DIMENSION TERM(100)
6      N=1
7      3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
8      N=N+1
9      IF (N-101) 3,6,6
10     6 N=1
11     7 SUM98 = SUM98+TERM(N)
12     WRITE(*,28) N, TERM(N)
13     N=N+1
14     IF (N-99) 7, 11, 11
15     11 SUM99=SUM98+TERM(N)
16     SUM100=SUM99+TERM(N+1)
17     IF (SUM98-3.141592) 14,23,23
18     14 IF (SUM99-3.141592) 23,23,15
19     15 IF (SUM100-3.141592) 16,23,23
20     16 AV89=(SUM98+SUM99)/2.
21     AV90=(SUM99+SUM100)/2.
22     COMANS=(AV89+AV90)/2.
23     IF (COMANS-3.1415920) 21,19,19
24     19 IF (COMANS-3.1415930) 20,21,21
25     20 WRITE(*,26)
26     GO TO 22
27     21 WRITE(*,27) COMANS
28     STOP
29     23 WRITE(*,25)
30     GO TO 22
31     25 FORMAT('ERROR IN MAGNITUDE OF SUM')
32     26 FORMAT('PROBLEM SOLVED')
33     27 FORMAT('PROBLEM UNSOLVED', F14.6)
34     28 FORMAT(I3, F14.6)
35     END
36
```

The diagram illustrates the flow of the Fortran program, highlighting the 'spaghetti code' nature of the jumps. The flow starts at line 7 (labeled 3) and proceeds to line 8, then to line 9. From line 9, a jump occurs to line 3 (labeled 3) if the condition is false, and to line 6 (labeled 6) if true. From line 6, the flow goes to line 7 (labeled 7). From line 14, a jump occurs to line 7 (labeled 7) if the condition is false, and to line 11 (labeled 11) if true. From line 11, the flow goes to line 15. From line 17, a jump occurs to line 14 (labeled 14) if the condition is false, and to line 23 (labeled 23) if true. From line 18, a jump occurs to line 23 (labeled 23) if the condition is false, and to line 15 (labeled 15) if true. From line 19, a jump occurs to line 23 (labeled 23) if the condition is false, and to line 16 (labeled 16) if true. From line 23, a jump occurs to line 19 (labeled 19) if the condition is false, and to line 21 (labeled 21) if true. From line 24, a jump occurs to line 21 (labeled 21) if the condition is false, and to line 20 (labeled 20) if true. From line 20, the flow goes to line 26. From line 26, the flow goes to line 22. From line 27, the flow goes to line 28. From line 29, a jump occurs to line 22 (labeled 22) if the condition is false, and to line 23 (labeled 23) if true. From line 30, the flow goes to line 22. From line 31, the flow goes to line 32. From line 33, the flow goes to line 34. From line 35, the flow goes to line 36.

“[T]he primary technical imperative you have as a programmer [...] is to manage complexity.”
- Steve McConnell

Programming Paradigms

- Logic programming
 - Give the computer a set of facts and a question, and it deduces the answer.
- Procedural programming
 - Group instructions into procedures, subroutines or functions.
- Functional programming
 - Functions can take functions as input and give functions as output
- Object-oriented programming
 - A program is a set of objects that send messages to each other.

Programming Principles

- Principle of Least Surprise
- DRY: Don't Repeat Yourself
- Single Responsibility Principle (“Small is Beautiful”)

Principle of Least Surprise

```
public static void doStuff(int[] a, int i, int j) {  
    a[i] ^= a[j];  
    a[j] ^= a[i];  
    a[i] ^= a[j];  
}
```

Principle of Least Surprise

```
public static void doStuff(int[] a, int i, int j) {  
    int tmp = a[i];  
    a[i] = a[j];  
    a[j] = tmp;  
}
```

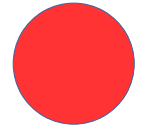
Principle of Least Surprise

```
public static void swap(int[] a, int i, int j) {  
    int tmp = a[i];  
    a[i] = a[j];  
    a[j] = tmp;  
}
```

Classes and Objects

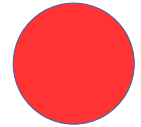
- When a program is running, the memory holds a number of **objects**
- Every object has a **class**
- You write a program by writing the code for a number of classes
- The code for class C tells us:
 - what data on object of class C stores (**fields**)
 - how to create an object of class C (**constructors**)
 - what an object of class C can do (**methods**)
- Together, the fields, constructors and methods are called the **members** of the class.

Classes - Syntax



```
public class ClassName {  
    private type1 field1;  
    private type2 field2;  
    ...  
    private typen fieldn;  
  
    public ClassName(type1 field1, ..., typen fieldn) {  
        this.field1 = field1;  
        ...  
        this.fieldn = fieldn;  
    }  
  
    public type method1(type arg, ..., type arg) {  
        ...  
    }  
  
    public type method2(type arg, ..., type arg) {  
        ...  
    }  
}
```

Classes - Syntax

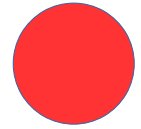


```
public class ClassName {  
    private type1 field1;  
    private type2 field2;  
    ...  
    private typen fieldn;  
  
    public ClassName(type1 field1, ..., typen fieldn) {  
        this.field1 = field1;  
        ...  
        this.fieldn = fieldn;  
    }  
  
    public type method1(type arg, ..., type arg) {  
        ...  
    }  
  
    public type method2(type arg, ..., type arg) {  
        ...  
    }  
}
```



Fields: hold data

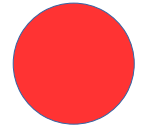
Classes - Syntax



```
public class ClassName {  
    private type1 field1;  
    private type2 field2;  
    ...  
    private typen fieldn;  
  
    public ClassName(type1 field1, ..., typen fieldn) {  
        this.field1 = field1;  
        ...  
        this.fieldn = fieldn;  
    }  
  
    public type method1(type arg, ..., type arg) {  
        ...  
    }  
  
    public type method2(type arg, ..., type arg) {  
        ...  
    }  
}
```

Methods: operations on fields and arguments

Classes - Syntax



```
public class ClassName {  
    private type1 field1;  
    private type2 field2;  
    ...  
    private typen fieldn;  
  
    public ClassName(type1 field1, ..., typen fieldn) {  
        this.field1 = field1;  
        ...  
        this.fieldn = fieldn;  
    }  
}
```

Constructors: special methods describing how
to create new instances of the class

```
public type method2(type arg, ..., type arg) {  
    ...  
}  
}
```

Java Classes

- A `public` class must be declared in a file named `ClassName.java`
- There can be at most one `public` class in a file.
- This class can be called from other files. The other classes cannot.

Methods

- Definition:

```
[public | private] typereturn methodName (type1 arg1, ..., typen argn)  
{  
    block  
}
```

- type_{return} can be void if it does not return a value.

- Method can be **called**:

```
o.methodName (exp1, ..., expn) ;
```

```
x = o.methodName (exp1, ..., expn) ;
```

where o is an object of the class

Constructors

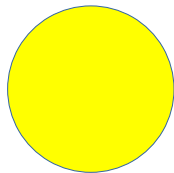
- Definition:

```
public ClassName (type1 arg1, ..., typen argn) {  
    block  
}
```

- Called using the `new` keyword:

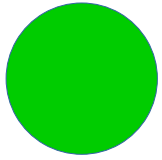
```
x = new ClassName (exp1, ..., expn) ;
```

- Note that name must be the same as name of class
- “block” should set the values of all the instance variables



Access Modifiers

- `public` and `private` are **access modifiers**
- `public` methods can be used by any class
- The `private` methods of class C can be used inside the class C
- When designing a class:
 - decide what public methods you want
 - add private methods when you need them to help write the public methods
- All instance variables should be `private`



Getters and Setters

Sometimes we want methods that just read or write the value of an instance variable.

We use `getVar` and `setVar` for methods that read or write to `var`

```
private T var;
```

```
public T getVar() {  
    return this.var;  
}
```

```
private void setVar(T val) {  
    this.var = val;  
}
```



Getters and Setters

- Don't overuse getters and setters!
- Only expose the parts of your class that *need* to be exposed!
- Don't let “bad” values slip in through setters!

```
private void setRadius(double r) {  
    if(r < 0) {  
        throw new IllegalArgumentException("negative radius");  
    }  
    this.radius = r;  
}
```

toString()

Every class has a method called `toString()`

This returns the **string representation** of the object.

This decides what is printed when the object is passed to `println()`

You can write your own method, or use Java's default.

It is recommended to write your own when you write a class.

```
public String toString() {  
    ...  
}
```


Example: Ponies!



this

The keyword `this` refers to the current object.

```
public void setRadius (double r) {  
    this.radius = r;  
}
```

...

```
myCircle.setRadius (42.0);
```



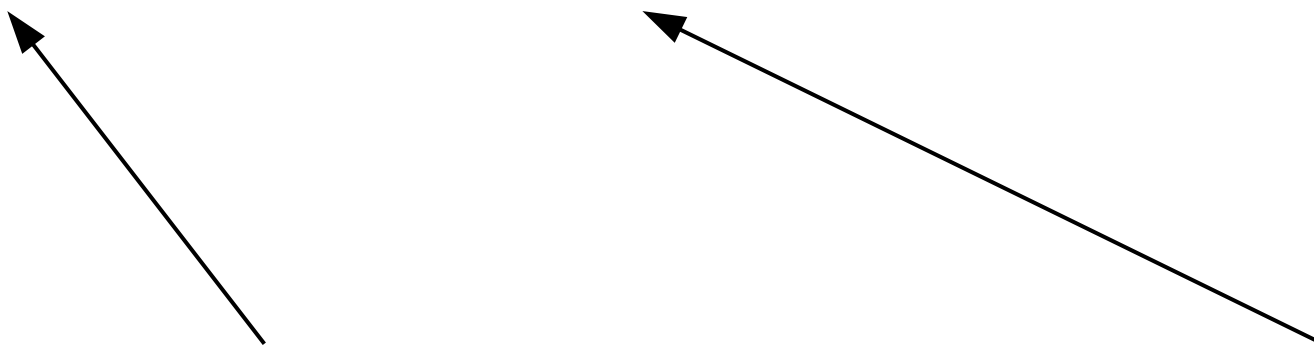
this

The keyword `this` refers to the current object.

```
public Circle(double r) {  
    this.radius = r;  
}
```

...

```
Circle myCircle = new Circle(42.0);  
// myCircle.getRadius() == 42.0
```



The diagram consists of two arrows. One arrow originates from the `new` keyword in the line `Circle myCircle = new Circle(42.0);` and points to the `this` keyword in the line `this.radius = r;` within the `Circle` constructor. The second arrow originates from the `Circle` class name in the same instantiation line and points to the `Circle` class name in the constructor signature `Circle(double r)`.

this

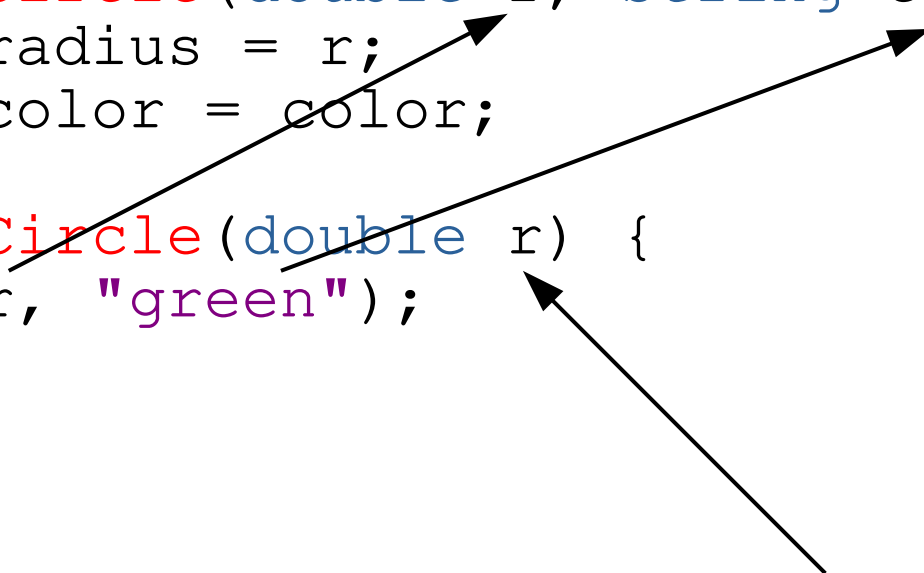
The keyword `this` can also be used to *chain* constructors

```
public Circle(double r, String color) {  
    this.radius = r;  
    this.color = color;  
}
```

```
public Circle(double r) {  
    this(r, "green");  
}
```

...

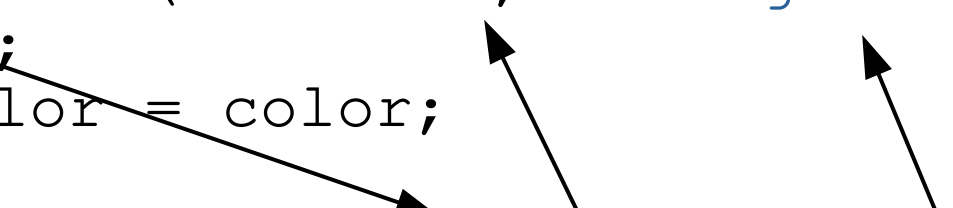
```
Circle myCircle = new Circle(42.0);  
// myCircle is now a green circle with radius 42
```



this

Chaining must happen *first* in the constructor

```
public Circle(double r, String color) {  
    this(r);  
    this.color = color;  
}
```



```
public Circle(double r) {  
    this.radius = r;  
}
```

...

```
Circle myCircle = new Circle(42.0, "blue");  
// myCircle is now a blue circle with radius 42
```



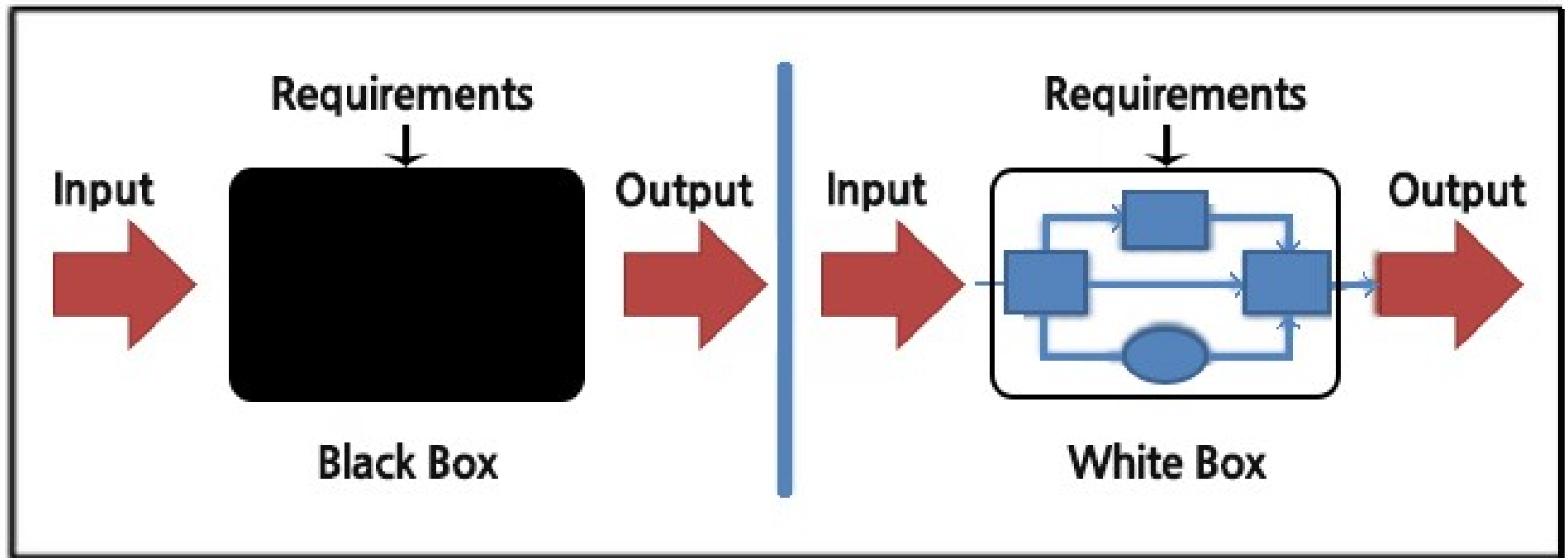
Programming Principles

- Principle of Least Surprise
- DRY: Don't Repeat Yourself
- Single Responsibility Principle (“Small is Beautiful”)

Class Methods and Instance Methods

- Instance methods:
 - defined without `static`
 - belong to the object, not the class
 - can use `this`
- Class methods:
 - defined using the keyword `static`
 - belong to the class, not to the object
 - can't touch `this`

Black Boxes and White Boxes



Black Boxes and White Boxes

- When **using** a class:
 - the class is a **black box**
 - the methods say what the objects can do
 - we do not think about how they do it
- When **writing** a class:
 - the class is a **white box**
 - we write methods that do what is required
 - we do not think about why they will be called

Example: Hangman

Programming Principles

- Principle of Least Surprise
- DRY: Don't Repeat Yourself
- Single Responsibility Principle (“Small is Beautiful”)

Example: Refactoring Hangman

Readings and Exercises

- **Readings:**

- *Java Direkt med Swing* 2.7, 2.9, 3.2, 3.4
- *Code Complete* Chapter 7

- **Exercises**

- If you did not complete last week's exercises, I recommend doing those first:

Java Direkt med Swing Exercises 2.1, 2.2, 2.3

- More practice with classes:

Java Direkt med Swing Exercises 2.4, 3.1, 3.2

- Exercises making use of everything we have done so far:

Java Direkt med Swing Exercises 3.6, 3.7, 3.12