Lecture 14
Wrapping Up

About the Exam

- Friday, March 13th, 08.30 to 11.00
  - It was the only available slot :(
- Will cover second half of the course
  - But **no** graphical programming!
- Some knowledge from first half still necessary
  - Kind of hard to come up with questions that don't involve methods, variables, etc…
  - Grading will be more lenient for these parts
- Everyone registered in Canvas on Feb. 28th (a Friday) will be registered for the exam
  - Spread the word!

Overloading

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

public Circle(double r) {
   ...
}
```

Overloading

```
public class Enemy {
   public void kick(int damage) {
      ...
   }
   public void kick(Player kicker) {
      ...
   }
}

...

Player john = new Player("John McClane");
Enemy hans = new Enemy("Hans Gruber");
hans.kick(5);
hans.kick(john);
```

Inheritance

- A class can *extend* another class
- `class Enemy extends Fighter { … }`
  - `Enemy` is *a subclass* of `Fighter`
  - `Fighter` is *the superclass* of `Enemy`
  - All **`public`** and **`protected`** members of `Fighter` are now also members of `Enemy`
  - Objects of type `Enemy` can be used as though they were of type `Fighter`:

```
public static void punch(Fighter f) { … }
…
punch(new Enemy(…));
```

  - An `Enemy` *IS* a `Fighter`!

Inheritance

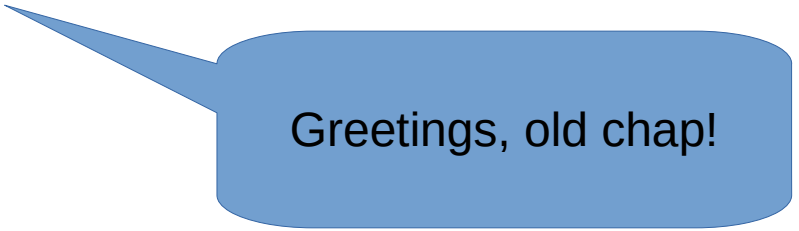
- A subclass can have methods not present in its superclass
-
```
class Person {
  public void talk() {
    System.out.println("Hi!");
  }
}
class BritishPerson extends Person {
  public void drinkTea() {
    System.out.println(
      "I do say, this blend is most delightful!"
    );
  }
}
…
BritishPerson p = new BritishPerson();
p.talk();
p.drinkTea();
```

Inheritance

- Adding a method to a subclass does *not* add it to its superclass
-
```
class Person {
  public void talk() {
    System.out.println("Hi!");
  }
}
class BritishPerson extends Person {
  public void drinkTea() {
    System.out.println(
      "I do say, this blend is most delightful!"
    );
  }
}
…
Person p = new BritishPerson();
p.talk();
p.drinkTea(); ← Compiler error! Person has no method
                drinkTea()!
```

Overriding

- A subclass can *override* its superclass' methods
-
```
class Person {
  public void talk() {
    System.out.println("Hi!");
  }
}
class BritishPerson extends Person {
  @Override
  public void talk() {
    System.out.println("Greetings, old chap!");
  }
}
Person p = new BritishPerson();
p.talk();
```



Overriding

- A subclass can *override* its superclass' methods
-

```
class Person {
  public void talk() {
    System.out.println("Hi!");
  }
}

class BritishPerson extends Person {
  public void talk() {
    System.out.println("Cheerio!");
  }
}

Person p = new BritishPerson();
p.talk();
```

Which methods are available?
Decided by *declared type*.

Which version of each method to call?
Decided by *actual type*.

Overriding vs. Overloading

- Overloading: decided at **compile time**

```
public class Enemy extends Fighter { … }

public void punch(Enemy e) {
  System.out.println("punched an enemy");
}
public void punch(Fighter e) {
  System.out.println("punched a fighter");
}

Fighter someone = new Enemy(…);
punch(someone);
```

- Output: `punched a fighter`

Overriding vs. Overloading

- Overriding: decided at **run time**

```
public class Fighter {
  public void punch() {
    System.out.println("fighter got punched");
  }
}

public class Enemy extends Fighter  {
  @Override
  public void punch() {
    System.out.println("enemy got punched");
  }
}

Fighter someone = new Enemy(…);
someone.punch();
```

- Output: `enemy got punched`

The `super` keyword

Let A be a subclass of B.

Inside the class A, the keyword `super` has two uses:

- It refers to the current object as if it were an object of class B, letting you use the methods and fields of class B.
- **As the first line of a constructor,** it invokes a constructor of B.

(Compare with the keyword `this`.)

Is-a vs Has-a

```
class Vehicle {
  public void speedUp() {…}
}
class Engine {
  public int getSize() {…}
}
```

How should we write the class Car?

We want to speedUp a car and get its engine size...

Is-a vs **Has-a**

- **Is-a** relationships are represented by **subclassing**
- **Has-a** relationships are represented by **composition**

A car **is a** vehicle
A car **has an** engine

```
class Car extends Vehicle {
   Engine engine;
   public int getEngineSize() {
      return engine.getSize();
   }
}
```

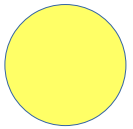
Interface

Before Java 8:

An **interface** is a collection of abstract methods:

```
interface HasMass {
   double getMass();
}
```

Note:

- All methods are public and abstract. Keywords are optional.
- Do not include them 

Implementing Interfaces

A class can **implement** an interface:

```
class PointMass implements HasMass {
   public double mass;

   @Override
   public double hasMass() {
      return mass;
   }
}
```

Implementing Interfaces

A class can **implement** an interface:

```
class RigidBody implements HasMass {
   public double volume;
   public double density;

   @Override
   public double hasMass() {
      return volume * density;
   }
   …
}
```

Implementing Multiple Interfaces

A class can only **extend** one **class** (abstract or non-abstract)

but it can **implement** many **interfaces**:

```
class FilledSquare
   extends Square
   implements Moveable, Drawable, ...
```

Abstract Classes

- An abstract class is an incomplete class!
- It may contain abstract methods – methods with no definition!
- The intention is that we create subclasses that implement these abstract methods in different ways.
- We cannot create an instance of an abstract class – only an instance of a completed subclass.

Abstract Classes - Example

```
abstract class Shape {
  public abstract double area();
}

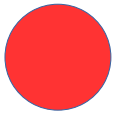
class Circle extends Shape {
  private double radius;

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

  @Override
  public double area() {
    return Math.PI * this.radius * this.radius;
  }
}

class Square extends Shape { … }
```

Abstract Classes - Rules



- A class is declared abstract with the abstract keyword
- A method is declared abstract with the abstract keyword
- If a class contains an abstract method, it must be an abstract class
- An abstract class C cannot be instantiated.
  new C(…) will not compile
- Abstract classes can contain everything that a non-abstract class can contain:
  instance variables, non-abstract methods, class methods, class variables

Recursion

- A method *calling itself* until some condition is met
- A recursive method `m` should have:
  - A conditional (if, switch, etc.) statement which decides which *case* to execute
  - 1+ base case(s), which return without recursing
  - 1+ recursive case(s), which call `m` with *smaller input*
    - For some notion of “smaller”

Recursive Example: Fibonacci

```
public int fib(int n) {
   if(n < 2) {
      return 1;
   }
   return fib(n-1) + fib(n-2);
}
```

Recursive Example: Fibonacci

```
public int fib(int n) {
  if(n < 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

- Don't worry about *how* `fib` works when you call it recursively
- Instead, just *assume* that it *will* solve the problem for the smaller input
- Then combine solutions for smaller inputs into solution for "your" input

Recursion vs Iteration

- It is always possible to rewrite a recursive function so that it is not recursive.
- Iterative methods are usually faster and use less memory
- Recursive methods can be easier to read, modify, test and debug
- Very useful for “backtracking” solutions

Generic Methods

A method signature may have type parameters:

```
public static <T> List<T> replicate(int copies, T elem) {
  List<T> list = new ArrayList<T>();
  for(int i = 0; i < copies; i++) {
    list.add(elem);
  }
  return list;
}
```

Inside the method, we may use type parameters like any other type.

- Variables can have type S, T or S[] or List<S> or …

However, we cannot write new S();

Generic Classes

Classes may also have type parameters:

```
public class Pair<S, T> {
    …
}
```

Now we have classes

- `Pair<Integer, Integer>`
- `Pair<String, Double>`
- `Pair<Pair<Integer, Integer>, Double>`
- etc.

A type parameter must be instantiated with a class (not a primitive data type).

Abstract classes and interfaces may have type parameters.

Lists

- Different types of list implement the `List<T>` interface
- `LinkedList<T>`
  - Fast append, slow indexing
- `ArrayList<T>`
  - Fast indexing, slow append
- Use the right one for your use case

Lists

- Lists are handy when:
  - We don’t know in advance how many elements we will need
  - We want to add and remove elements later (not just overwrite old ones)
- They are not so good when:
  - We need very high performance
  - We need to minimize memory usage

Maps (“avbildningstabeller”)

The interface `Map<K, V>` has the following methods:

- `boolean put(K key, V value)` (optional) – associates value with key
- `boolean containsKey(Object o)` – true if the map contains an entry with the key o
- `E get(K key)` – returns the element associated with the given key
- `int size()` - number of mappings in the map
- ...

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

`equals` has a friend: `hashCode`

- Generates a **hash** of the object
- If `a.equals(b)`, then

  `a.hashCode() == b.hashCode()`

  - Does **NOT** apply the other way around!
    (E.g. `a.hashCode() == b.hashCode()` does not imply `a.equals(b)`)
- Used to speed up comparisons
  - ```
    if(a.hashCode() == b.hashCode()) {
      return a.equals(b);
    } else {
      return false;
    }
    ```
    ```

`equals` has a friend: `hashCode`

- Simplest valid implementation of equals:

```
@Override
public int hashCode() {
  return 0;
}
```

- A more useful implementation:

```
@Override
public int hashCode() {
  int hash = this.name.hashCode();
  hash = hash*97 + this.skill.hashCode();
  hash = hash*97 + this.age;
  ...
  return hash;
}
```

HashMap

- Efficient implementation of `Map<K, V>` based on hashing
- Unordered, but useful in most circumstances when you want a map
- Performance and correctness depends on `K.hashCode()` being correct and well-written

File Handling

- A text editor that can’t save or load files is pretty useless
- So far we’ve used `java Program < file.txt`
- But this is very inflexible
  - What if we want to read more than one file?
  - What if we don’t know which file to read when we start the program?

File Handling

- We can use the `File` class to work with files

```
File file = new File("my_file.txt");
if(file.exists()) {
   System.out.println("The file exists!");
   file.delete();
   System.out.println("Now it's gone!");
} else {
   System.out.println("The file does not exist!");
}
```

- `File` lives in package `java.io`.
- https://docs.oracle.com/javase/7/docs/api/java/io/File.html

Reading Files

- We can construct a `Scanner` from a `File`

```
File file = new File("my_file.txt");
try {
  Scanner scan = new Scanner(file);
  while(scan.hasNextLine()) {
    System.out.println(scan.nextLine());
  }
  scan.close();
} catch (FileNotFoundException e) {
  System.out.println("The file does not exist!");
  System.exit(1);
}
```

Writing Files

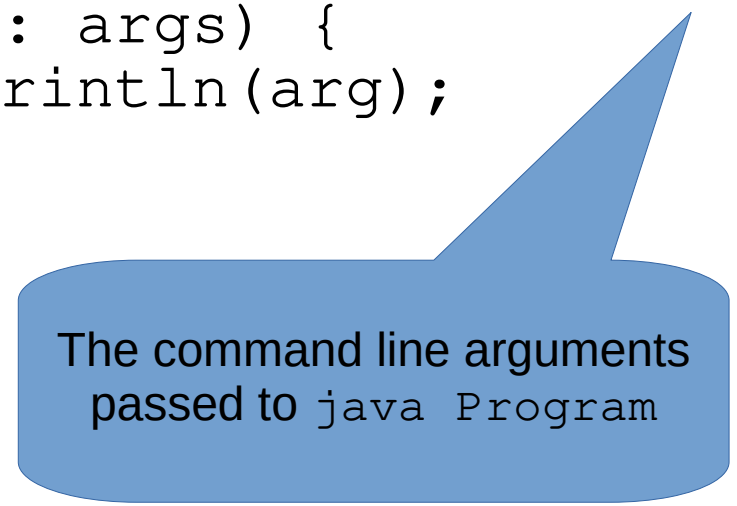
- We can construct a `FileWriter` from a `File`
- ...which we then use to construct a `PrintWriter`

```
File file = new File("my_file.txt");
try {
  FileWriter fileWriter = new FileWriter(file);
  PrintWriter writer = new PrintWriter(fileWriter);
  writer.println("Hello, I'm a line of text!");
  writer.println("And so am I!");
  writer.close();
} catch (IOException e) {
  System.out.println("Something went wrong!");
  System.exit(1);
}
```

- `FileWriter` and `PrintWriter` live in package `java.io`.

Command Line Arguments

```
public class Program {
  public static void main(String[] args) {
    for(String arg: args) {
      System.out.println(arg);
    }
  }
}
```




```
java Program Hello, I am the arguments!
```

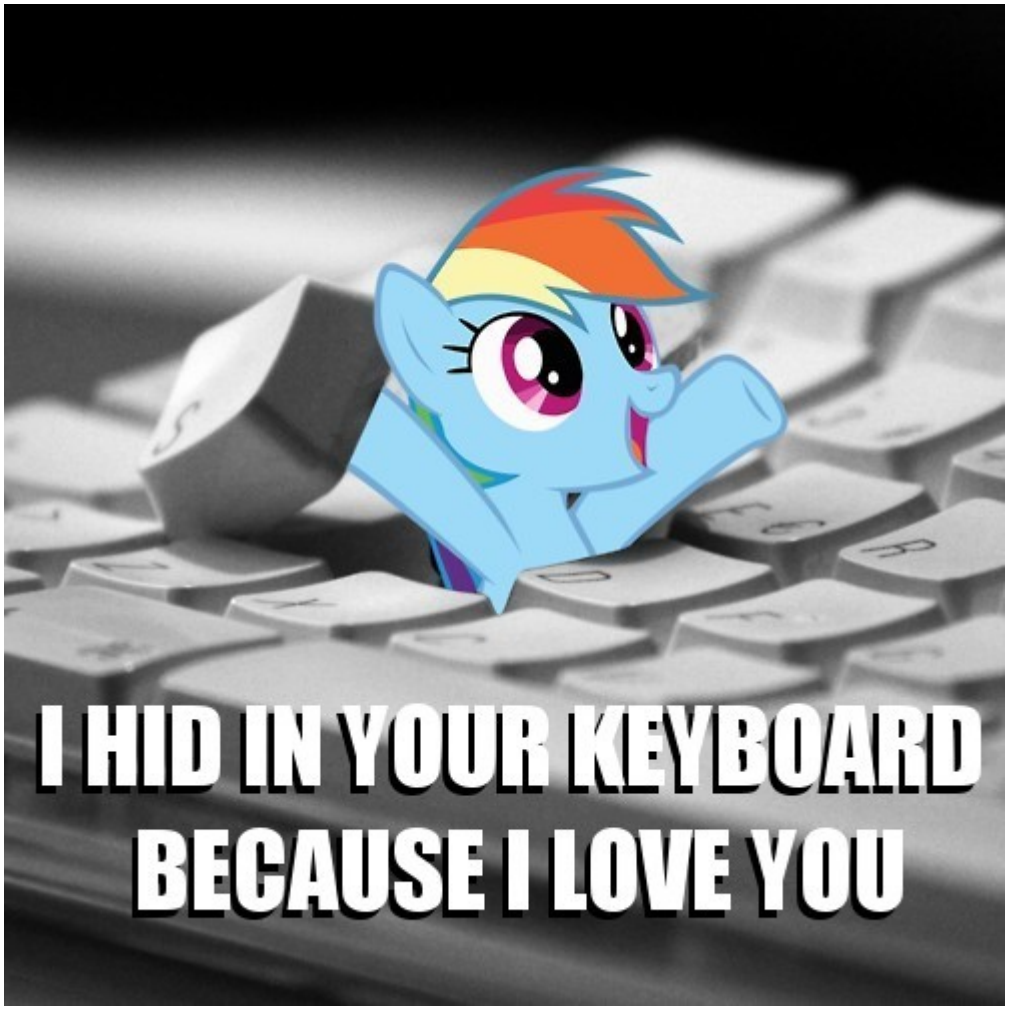
Prints:

```
Hello,
I
am
the
arguments!
```

Reading and Exercises

- Reading
  - Everything from lectures 9 through 13
- Exercises
  - Everything from lectures 9 through 13
  - Bonus exercises
  - Old exams
    - But check course website for old exam errata!

Good luck on the exam!



Thanks for a great course!