Lecture 10
Abstract classes, interfaces, packages, lists

Inheritance Recap

- 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 Recap

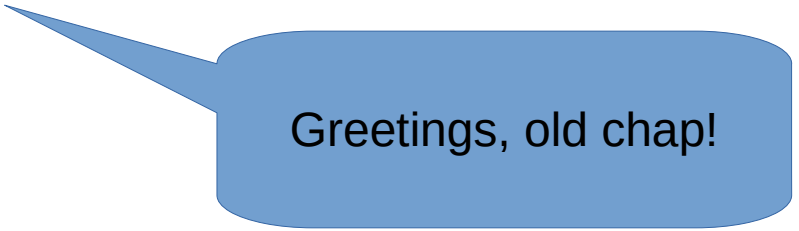
- A subclass can have methods not present in its superclass
-
```
class Person {
  public void talk() {
    System.out.println("Hi!");
  }
}
class BritishPerson extends Person {
  public void drinkTea() {
    System.out.println(
      "I do say, this blend is most delightful!"
    );
  }
}
…
BritishPerson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

Inheritance Recap

- 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()!
```

Inheritance Recap

- 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();
```



Inheritance Recap

- 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*.

Inheritance Recap

Code Example: Adding a Boss Enemy

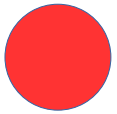
Abstract Classes

- 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

Abstract Classes

Code Example: Abstract Fighters

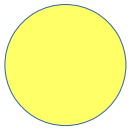
Interface

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;
   }
}
```

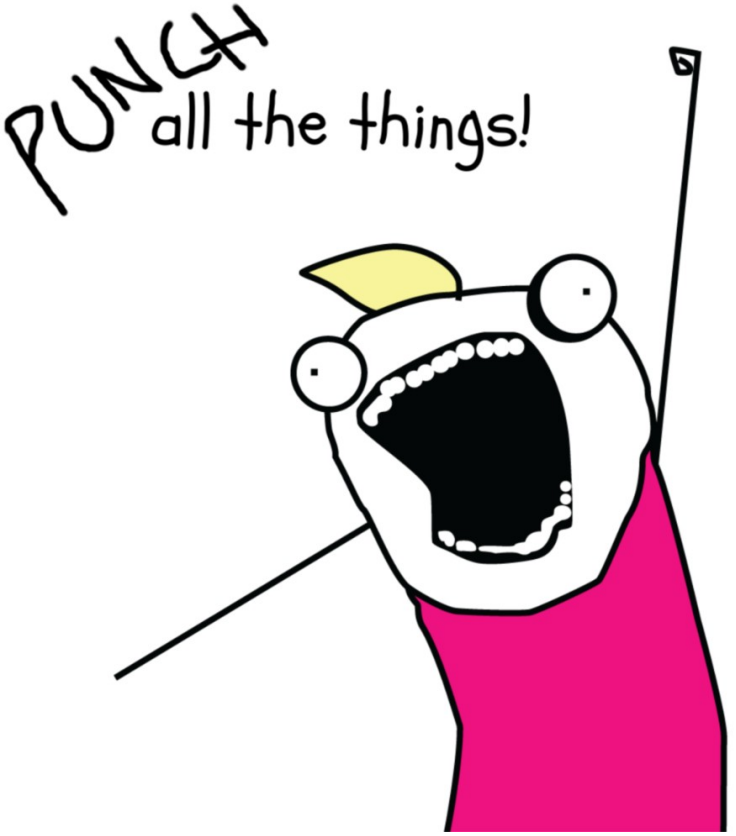
Implementing Interfaces

A class can **implement** an interface:

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

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

Implementing Interfaces

Code Example



Abstract Classes vs Interfaces

Before Java 8:

- Abstract classes may contain instance variables, non-abstract methods
- Interfaces can only contain abstract methods (and constants)
- A class can only extend one class (abstract or non-abstract)
- A class can implement more than one interface

Multiple Inheritance

- The Diamond Problem (Deadly Diamond of Death):

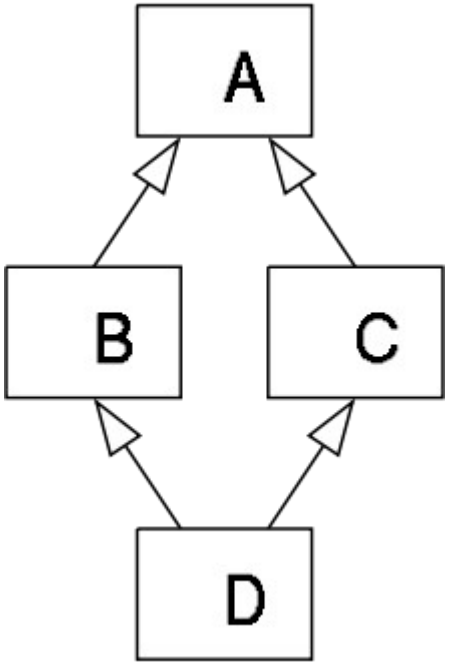


Image from Wikipedia

A has a method f
B and C both override f
We call f on an object of type D
Which code is executed?

Solutions from Various Languages:

- C++: We have to say which we want when we call f (B::A.f() or C::A.f())
- Python: We give the superclasses an order when we construct D
- PHP (before 2012): No multiple inheritance

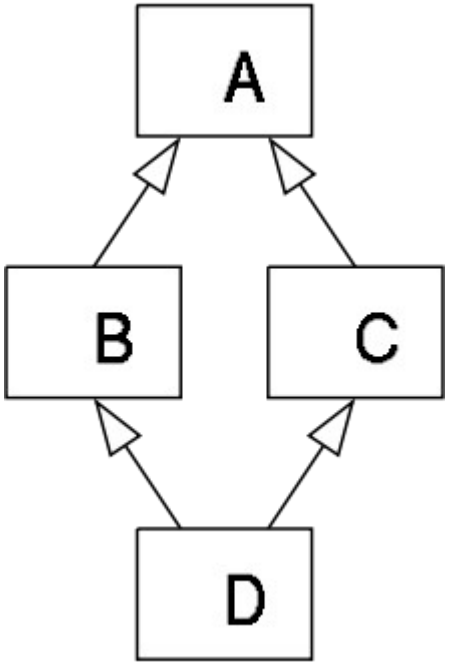
Implementing Multiple Interfaces

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

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

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

Multiple Inheritance



So f cannot be defined (non-abstract) in both B and C. When we call f() on an object, it must have just one definition.

Abstract Classes vs Interfaces

Java 7

	Abstract Class	Interface
Instance variables	Yes	No
Abstract methods	Yes	Yes
Public non-abstract methods	Yes	No
Constants	Yes	Yes
Constructors	Yes	No
Private methods	Yes	No

Abstract Classes vs Interfaces

Java 8

	Abstract Class	Interface
Instance variables	Yes	No
Abstract methods	Yes	Yes
Public non-abstract methods	Yes	**Yes**
Constants	Yes	Yes
Constructors	Yes	No
Private methods	Yes	No

- Public non-abstract methods are added with the `default` keyword

Abstract Classes vs Interfaces

Java 9

	Abstract Class	Interface
Instance variables	Yes	No
Abstract methods	Yes	Yes
Public non-abstract methods	Yes	Yes
Constants	Yes	Yes
Constructors	Yes	No
Private methods	Yes	**Yes**

So what about the Diamond Problem?

- If interfaces I1 and I2 both implement default method f (i.e. non-abstract method):
  class C implements I1, I2 { … }
  will not compile.
- I expect in future versions of Java: no difference between abstract classes and interfaces

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();
   }
}
```

Packages

Classes are collected into **packages**

- Package names are all lower case
- Put related classes in the same package

The first line of the file gives the name of the package

```
package graphics;
```

- The package name must be the same as the directory name

Namespaces

Suppose class Circle is in package graphics

Inside graphics: we refer to it as Circle

Outside graphics: we must **import** it to use it

- `import graphics;`
  - We can now refer to the class as graphics.Circle
- `import graphics.Circle;`
  - We can now refer to the class as Circle

Access Modifiers

- A class or interface may be `public` or package-private (no modifier)
  - Public: Can be accessed anywhere in the program
  - Package-private: Can only be accessed within the same package
- A file can contain at most one public class, which must have the same name as the file (minus .java)

Access Modifiers

- A member (method or instance variable) can be `public`, `protected`, `private` or package-private (no modifier)

Access Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other packages*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Image from opensourceforgeeks.blogspot.com

Subpackages

Packages can contain subpackages:

```
package graphics.3d;
```

This must be in the directory graphics/3d

- **Reading**
  - Java Direkt med Swing 10.7-10.8, 3.1-3.2
- **Exercises**
  - Java Direkt med Swing chapter 10
  - The Object-Oriented Zoo 2.0 (see course website)
  - Play around with this week’s code examples – can you turn the pieces into a simple, text-based game?