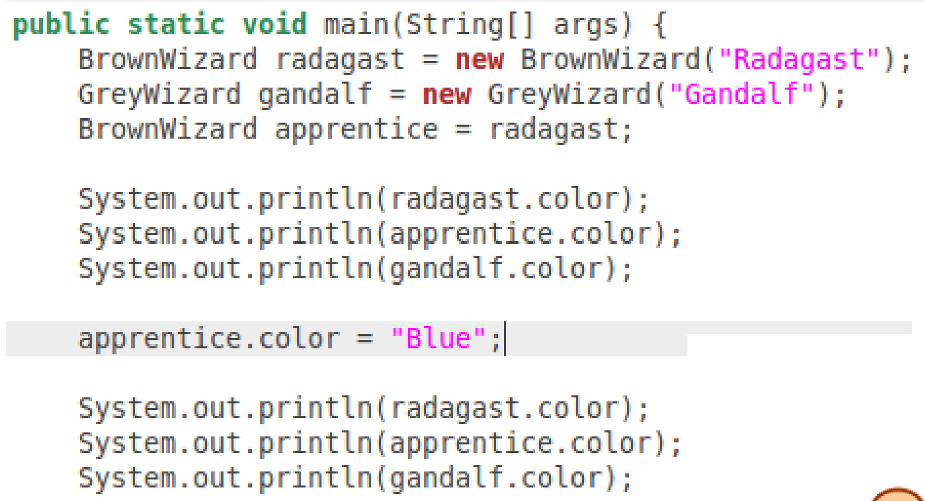
Java Basics: Polymorphism, Abstract Class, Interface, and Package

Professor Sudip Misra
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur
http://cse.iitkgp.ac.in/~smisra/

Object References

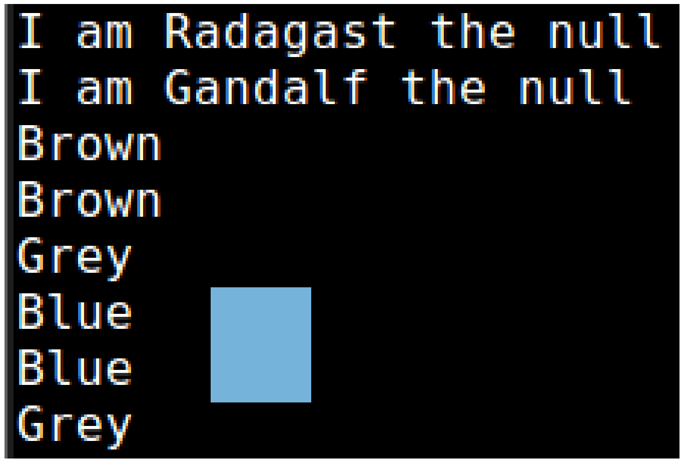
- Before looking at the final feature of OOP, polymorphism, let us look at how object references work
- Consider the code:
  - `GreyWizard gandalf = new GreyWizard(“Gandalf”);`
- This can be split into two statements:
  - Declaration:
    - `GreyWizard gandalf;`
  - `Assignment:`
    - `gandalf = new GreyWizard(“Gandalf”);`

OBJECT REFERENCES (CONT'D)

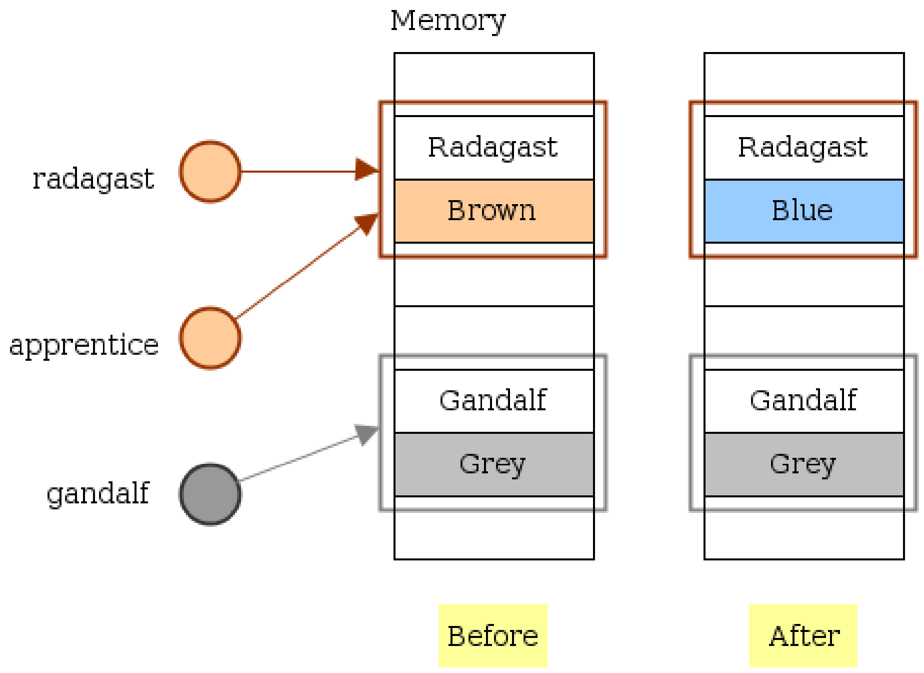


```
public static void main(String[] args) {
    BrownWizard radagast = new BrownWizard("Radagast");
    GreyWizard gandalf = new GreyWizard("Gandalf");
    BrownWizard apprentice = radagast;

    System.out.println(radagast.color);
    System.out.println(apprentice.color);
    System.out.println(gandalf.color);

    apprentice.color = "Blue";

    System.out.println(radagast.color);
    System.out.println(apprentice.color);
    System.out.println(gandalf.color);
```

The handles radagast and apprentice both point to the same object in memory



```
I am Radagast the null
I am Gandalf the null
Brown
Brown
Grey
Blue
Blue
Grey
```



Object References (Cont'd)

- Parameter passing in Java pass-by-value only
  - Values of actual arguments are copied into formal parameters
  - Object references are also passed by value
    - Value of the object references are copied into formal parameters
- Unlike C/C++, no pointers in Java
  - Users cannot manipulate memory arbitrarily => ensures security

See: `http://leepoint.net/JavaBasics/methods/method-commentary/methcom-20-passby.html`

Polymorphism

- Greek for "many forms"
- Ability of a class (object) to respond differently under different conditions
  - "One interface, multiple methods"
- Two types:
  - Compile-time (static) polymorphism
    - Achieved using method overloading
      - Methods with different signature in the same class
  - Run-time (dynamic) polymorphism
    - Achieved using method overriding
      - Methods with exactly the same signature in parent & child classes
- Unlike C++, no operator overloading in Java

Polymorphism: Upcasting

- A superclass object reference can point to a subclass:
  - `Wizard wizard = new GreyWizard();`
  - Meaningful, because a GreyWizard is also a Wizard
  - Referred to as "upcasting"
- A subclass reference **cannot** point to a superclass type:
  - `BrownWizard bWizard = new Wizard("Gandalf");`
  - The above code returns "incompatible types" error during compilation
  - Meaningful, because a Wizard is not necessarily a BrownWizard

Access Principles

- Two key principles to remember when members/methods are accessed via types other than the corresponding class
  - Members: The type of the **object reference**, not the type of the object that it refers to, determines the members that can be accessed
  - Methods: The type of the **object pointed to**, not the type of the reference, determines the methods that can be accessed

Access Principles (Cont'd)

- `BrownWizard bWizard = new GreyWizard("Gandalf");`
  - Since the type of the **object reference** is `BrownWizard`, only members known to `BrownWizard` can be accessed
  - On the other hand, since the type of the **object referred** to is `GreyWizard`, methods known to the `GreyWizard` class can be invoked

Polymorphism of Instance Variables ?

- Technically, NO
  – Only method overriding involves run-time/late binding
  – Variables are compile-time bound to their types
  – When upcasting, the type of a variable is bound to the parent class
- So, polymorphism is only related to methods
- "... you can only override behavior and not structure." [1]

[1]: http://stackoverflow.com/a/427790/147021

Modify Wizards

- Let us introduce an instance variable, `level`, in the `BrownWizard` class

```
protected int level = 1;
```

- Also, inside the constructor of `GreyWizard`, set:

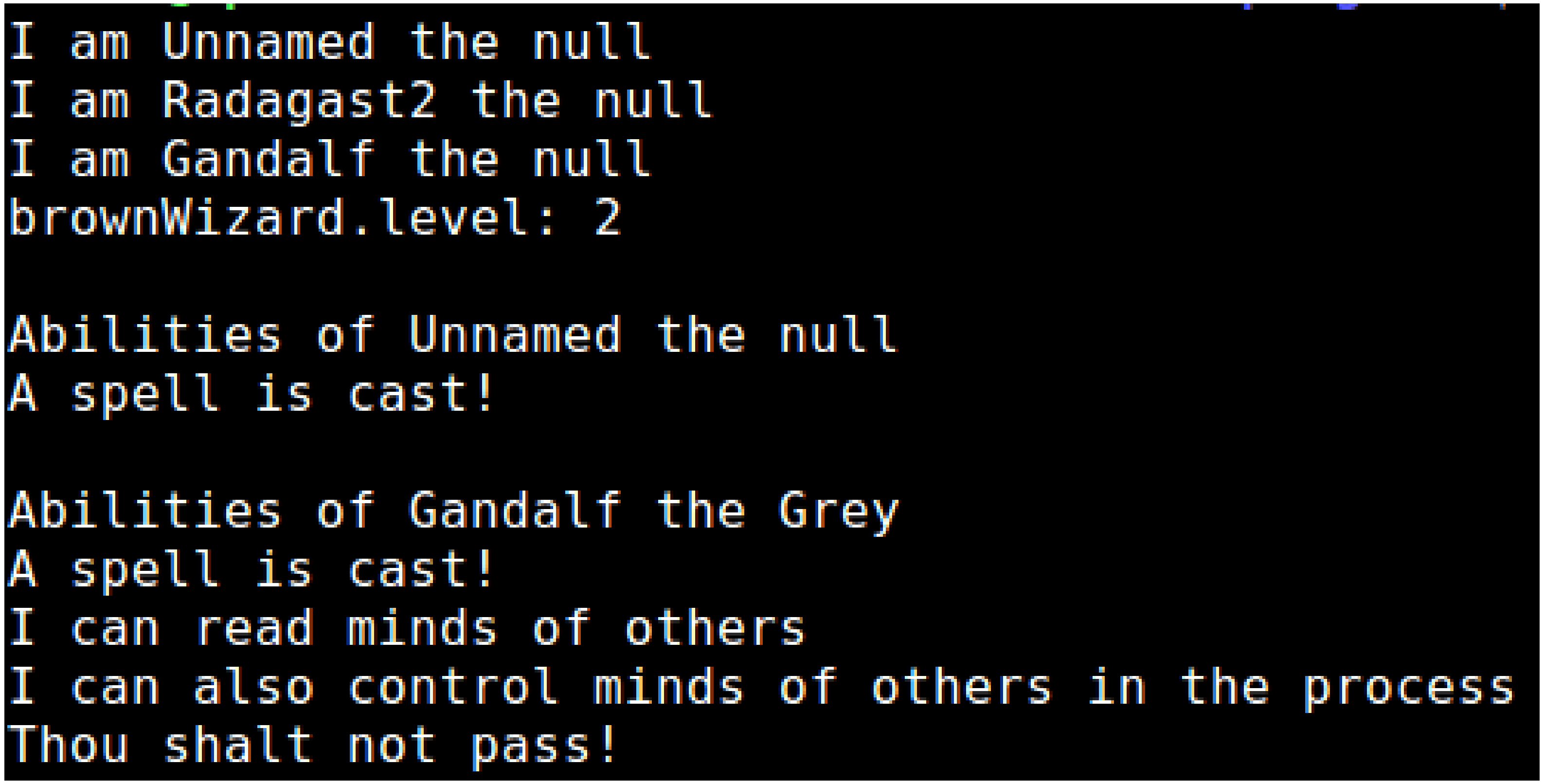
```
level = 2;
```

- In the following, let us look at some of the behaviors induced by different object references

Run-time Polymorphism: Examples

```java
public static void main(String[] args) {
    Wizard wizard = new Wizard("Unnamed");
    Wizard wizard2 = new BrownWizard("Radagast2");
    BrownWizard brownWizard = new GreyWizard("Gandalf");

    // This is not possible since level is not known to Wizard
    // System.out.println("wizard.level: " + wizard.level);

    // This is OK
    System.out.println("brownWizard.level: " + brownWizard.level);

    wizard.printAbilities();

    // Which abilities would it print?
    brownWizard.printAbilities();
```

Run-time Polymorphism: Output



```
I am Unnamed the null
I am Radagast2 the null
I am Gandalf the null
brownWizard.level: 2

Abilities of Unnamed the null
A spell is cast!

Abilities of Gandalf the Grey
A spell is cast!
I can read minds of others
I can also control minds of others in the process
Thou shalt not pass!
```

Run-time Polymorphism: Explanation

- No polymorphism of instance variables
- `brownWizard.level` prints **2** (not 1) because the object is of type `GreyWizard`, and in `GreyWizard` we "overwrite" level to 2
  - `wizard.printAbilities()` prints only a single ability (spell casting), as expected
  - `brownWizard.printAbilities()` prints the abilities of a `GreyWizard`, because the type of the object referred to here is `GreyWizard`

Abstract Classes & Methods

- Abstract method: One which is only declared, but not defined
- Abstract class: One that contains one or more abstract methods
- Can we instantiate an abstract class?
  - No
- Can we extend an abstract class?
  - Yes!
- Child of an abstract class?
  - Concrete, if it implements all the abstract methods
  - Abstract, otherwise

Abstract Classes & Methods (Cont'd)

```java
public abstract class AbstractWorld {

    private double x, y;

    // An abstract class can have constructor(s)
    public AbstractWorld() {
        // Code
    }

    // Non-abstract methods are allowed
    public void setDimensions(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public abstract void drawWorld();
}
```

Abstract Classes & Methods (Cont'd)

```java
public class MiddleEarth extends AbstractWorld {

    public MiddleEarth() {

    }

    public void drawWorld() {
        System.out.println("Behold ye, Middle Earth before thee!");
    }

    public static void main(String[] args) {
        MiddleEarth earth1 = new MiddleEarth();
        MiddleEarth earth2 = new MiddleEarth();

        earth1.drawWorld();
        earth2.drawWorld();
    }
}
```

Abstract Classes & Methods (Cont'd)

- Output of execution:

```
Behold ye, Middle Earth before thee!
Behold ye, Middle Earth before thee!
```

- What is the problem here?
  - We can create several instances of MiddleEarth
  - In reality, there can be only a single world (instance)
- How to fix it?

Singleton Class

- A class that has at most a single instance
- Make constructor private
- Return a static instance
  - Lazy initialization: Do not initialize until it is required
- Let us convert MiddleEarth into a singleton class

MiddleEarth Revisited

```
public class MiddleEarth extends AbstractWorld {

    private static MiddleEarth middleEarth = null;

    // Prevent anyone else from instantiating this class
    private MiddleEarth() {}

    public static MiddleEarth getInstance() {
        // Lazy instantiation
        if (middleEarth == null) {
            middleEarth = new MiddleEarth();
        }

        return middleEarth;
    }

    public void drawWorld() {
        System.out.println("Behold ye, Middle Earth stands before thee!");
    }
}
```

MiddleEarth Revisited (Cont'd)

```java
public class MiddleEarthTest {
    public static void main(String[] args) {
        // Invalid
        //MiddleEarth earth = new MiddleEarth();

        MiddleEarth earth = MiddleEarth.getInstance();

        earth.drawWorld();
    }
}
```

What is Interface?

- Abstraction and polymorphism to the fullest extent
- A general “contract”
- Classes respect the contract by **implementing** interfaces
- A class can implement any number of interfaces
  - But can extend only a single class
  - Interfaces help simulate multiple inheritance
- An interface can be implemented by any number of classes
- An interface can extend **multiple** interfaces

What does an Interface Contain?

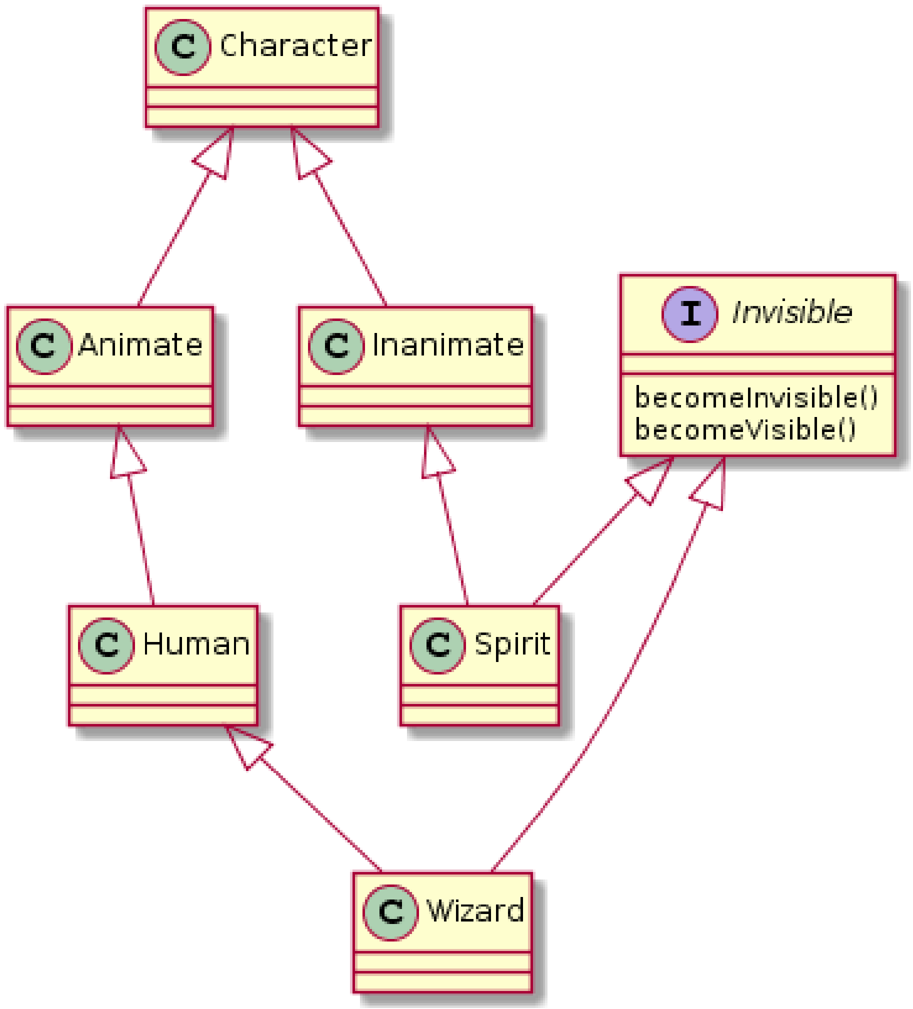
- Method signatures (name, parameters, return type)
  - **No** implementation*
- Static constants
- All methods are abstract implicitly
  - No need to mention as abstract
- An interface is abstract implicitly
  - Cannot instantiate
- Methods & constants are publicly implicitly (unless the interface is private)

* This somewhat changed in Java 8

How to name an Interface?

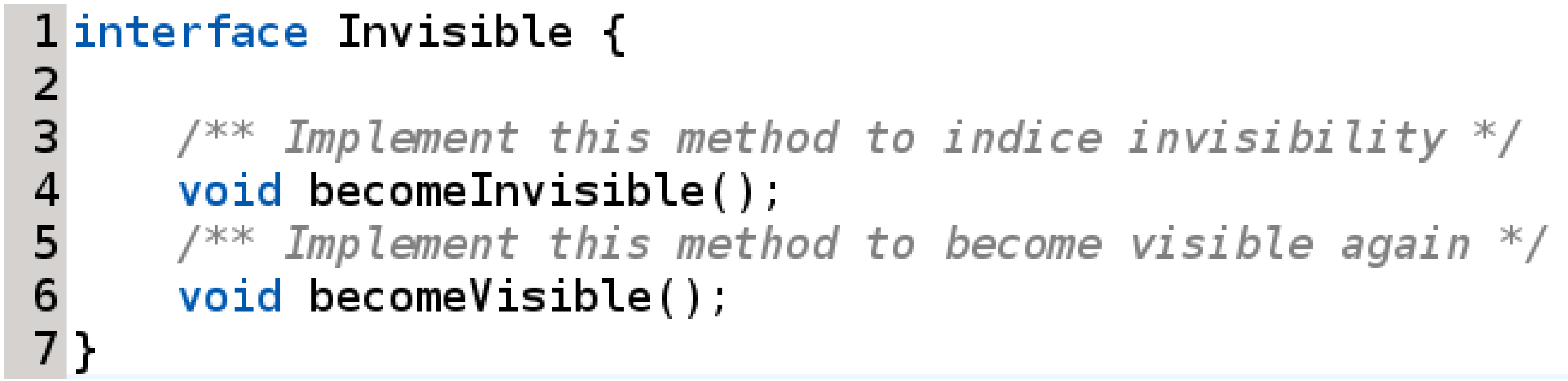
- Generally adjectives are used to indicate the behavior of the contract (what it does)
- Some prefix it with “I”, e.g., IEvent
  - Often not preferred
- Some examples from the Java library:
  - Comparable
  - Runnable
  - Renderer
  - ActionListener
  - Painter

Interface: Example



THE INVISIBLE INTERFACE

- Humans, in general, cannot become invisible
  - But Wizards can
- Spirits generally are invisible
- Does not make sense invisibility to be a behavior of all Characters



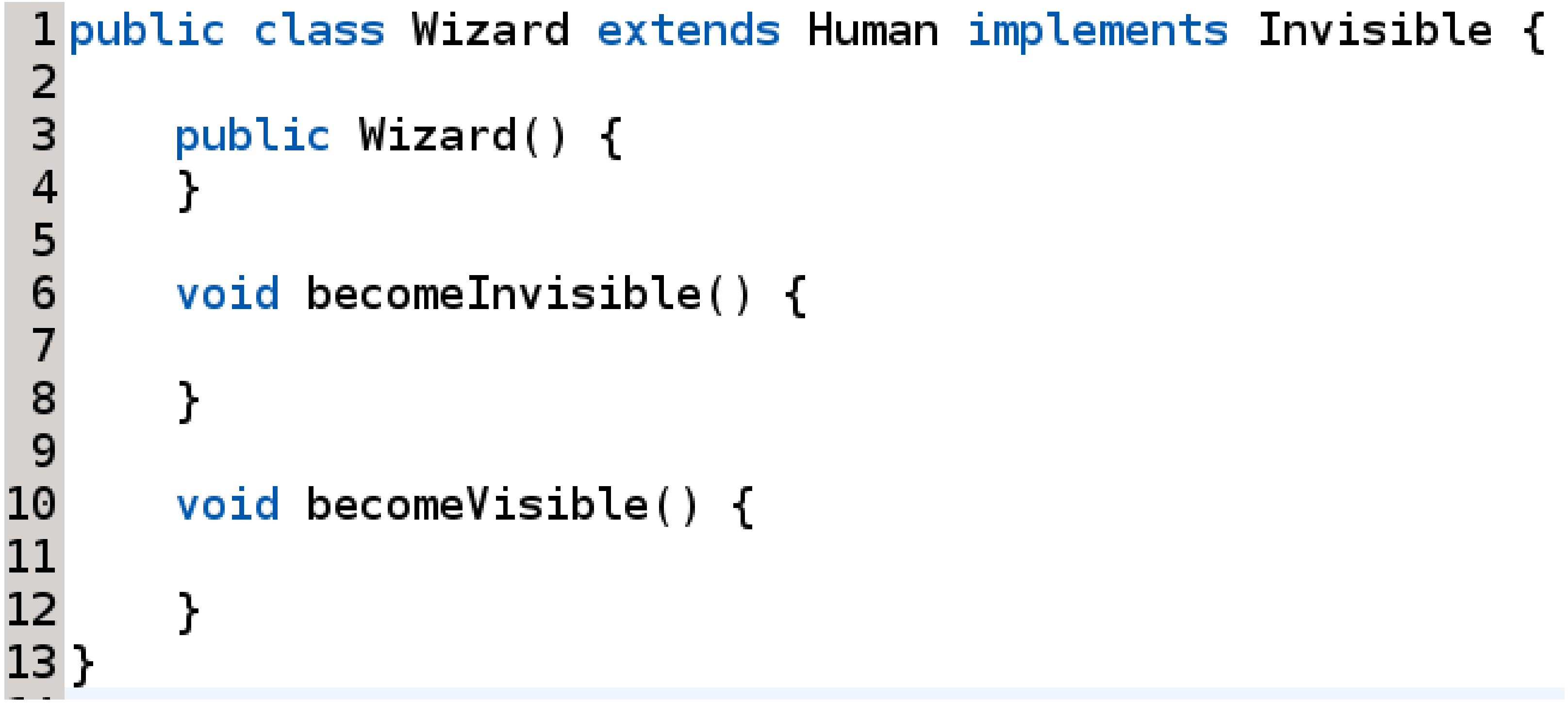
```
interface Invisible {

    /** Implement this method to indice invisibility */
    void becomeInvisible();
    /** Implement this method to become visible again */
    void becomeVisible();
}
```

The Wizard Class (1)

```java
public class Wizard extends Human implements Invisible {

    public Wizard() {

    }
}
```

```
Compilation error:
Wizard.java:1: error: Wizard is not abstract
  and does not override abstract method
  becomeVisible() in Invisible
public class Wizard extends Human implements
  Invisible {
        ^
1 error
```

The Wizard Class (2)



```
public class Wizard extends Human implements Invisible {

    public Wizard() {
    }

    void becomeInvisible() {

    }

    void becomeVisible() {

    }
}
```

Is it OK now?

The Wizard Class (2)

```
Wizard.java:10: error: becomeVisible() in
  Wizard cannot implement becomeVisible() in
  Invisible
    void becomeVisible() {
         ^
  attempting to assign weaker access
  privileges; was public
Wizard.java:6: error: becomeInvisible() in
  Wizard cannot implement becomeInvisible() in
  Invisible
    void becomeInvisible() {
         ^
  attempting to assign weaker access
  privileges; was public
2 errors
```

The Wizard Class (Complete)

```java
public class Wizard extends Human implements Invisible {

    public Wizard() {
    }

    public void becomeInvisible() {

    }

    public void becomeVisible() {

    }
}
```

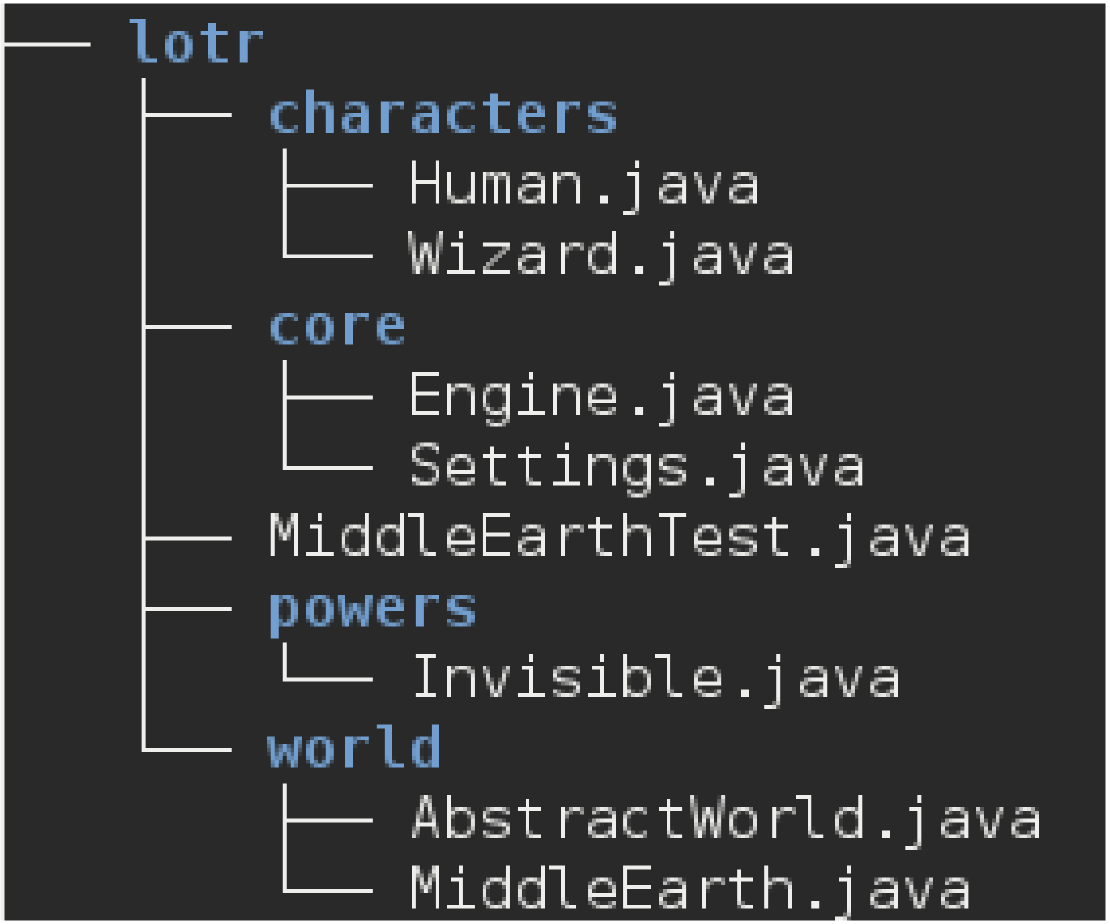
Why Package?

- Consider you are developing a game based on LoTR
  - Lots and lots of Java class & interface files!
- Difficult to manage when kept inside a single directory
- Collision of classes:
  - Suppose you declare a Collection class to represent collection of different artifacts
  - Java library itself has a Collection class!
- Segregate namespace to avoid conflict of types

How to ...

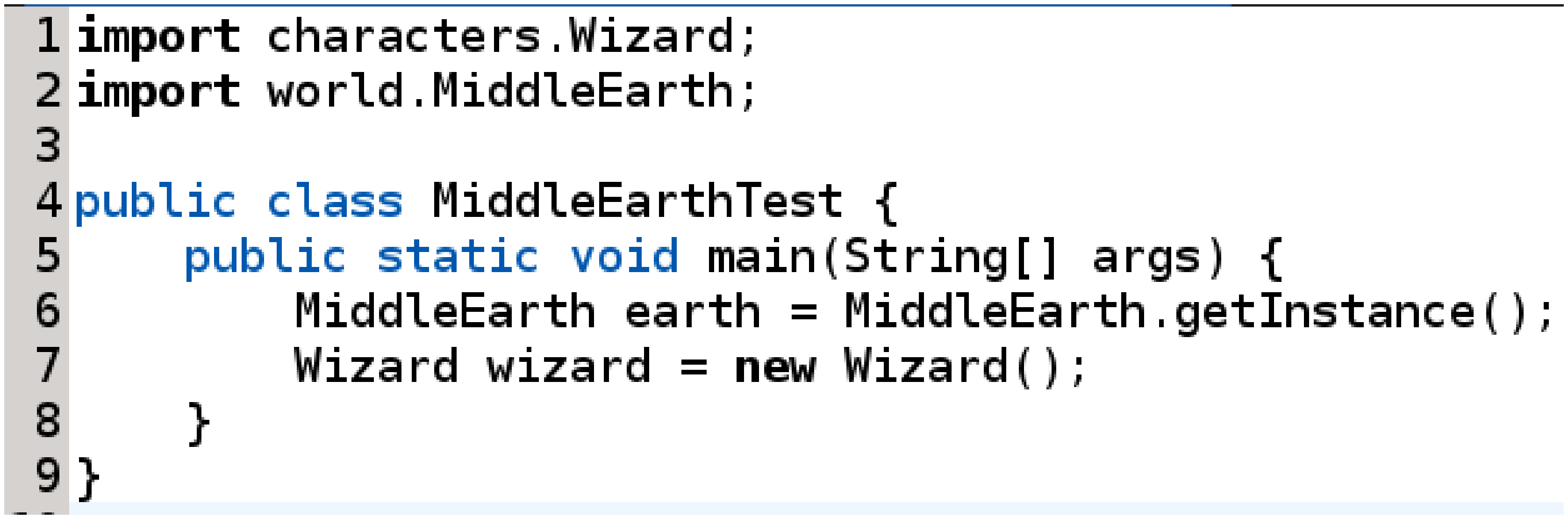
- Declare a package: Add as the first line of class/interface
  - package pkgname;
- Store a package: Simply use directories!
  - Hierarchical package structure
  - Package pkg1.sub is stored in the directory pkg1/sub/
  - Example: java.awt.image
- Use a package: Import them into your code
  - import java.awt.image;

Example: Directory Structure



```
── lotr
   ├── characters
   │   ├── Human.java
   │   └── Wizard.java
   ├── core
   │   ├── Engine.java
   │   └── Settings.java
   ├── MiddleEarthTest.java
   ├── powers
   │   └── Invisible.java
   └── world
       ├── AbstractWorld.java
       └── MiddleEarth.java
```

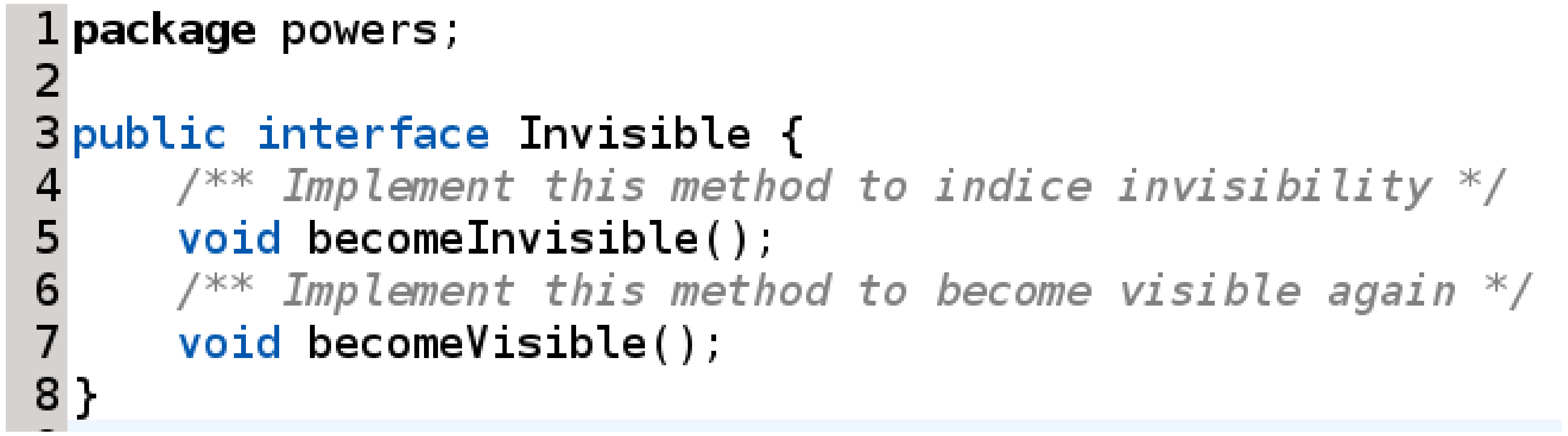
MiddleEarthTest Class



```
import characters.Wizard;
import world.MiddleEarth;

public class MiddleEarthTest {
    public static void main(String[] args) {
        MiddleEarth earth = MiddleEarth.getInstance();
        Wizard wizard = new Wizard();
    }
}
```

Invisible Interface

- We now explicitly declare Invisible to be public
  - Earlier it was package-private
  - Would have resulted in compilation error



```java
package powers;

public interface Invisible {
    /** Implement this method to indice invisibility */
    void becomeInvisible();
    /** Implement this method to become visible again */
    void becomeVisible();
}
```

Wizard Class

```java
package characters;

import powers.Invisible;

public class Wizard extends Human implements Invisible {

    public Wizard() {
    }

    public void becomeInvisible() {
    }

    public void becomeVisible() {
    }
}
```

WIZARD CLASS (CONT'D)

- Wizard and Human belong to the same package
- No need to import the classes from the same package
  - They are automatically available in the namespace
- Therefore, two classes with the same name cannot exist in the same package
- How to compile?
  - Go **inside** the lotr/ directory
  - javac MiddleEarthTest.java
  - java MiddleEarthTest

Importing Several Classes

- To import all classes from a package:
```
import java.util.*;
```
- Generally not recommended
- If only a few classed from the package are used, import them individually
```
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
```
- Generally listed alphabetically

Thank you!