Object Oriented Programming Using Java - I

Professor Sudip Misra

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

Programming Paradigms

- In general, two major schools:

- Procedural:
  - Top-down approach
  - Splits the logic into a set of subroutines
  - Process oriented
  - Cannot hide data

- Object oriented:
  - Modeled on real-world objects
  - Focus on data
  - Four cornerstones:
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism

Note: This lecture is based on Java 6. You are encouraged to learn the features of Java 8 and 9 by yourself.

OOP Basics

- Class: A blueprint of all objects belonging to the same type
  - More like an architecture/specification, e.g., Student
- Object: A particular instance of a class
  - Consists of states (aka data, attributes, and members) and behaviors (aka methods)
  - More like actual occurrences
  - Memory allocated for objects
  - Example: “Harry Potter” and “Draco Malfoy” are two different students belonging to the same class Student
    - States: name, roll, house
    - Behaviors: startQuest(), playQuidditch()

Note: This lecture is based on Java 6. You are encouraged to learn the features of Java 8 and 9 by yourself.

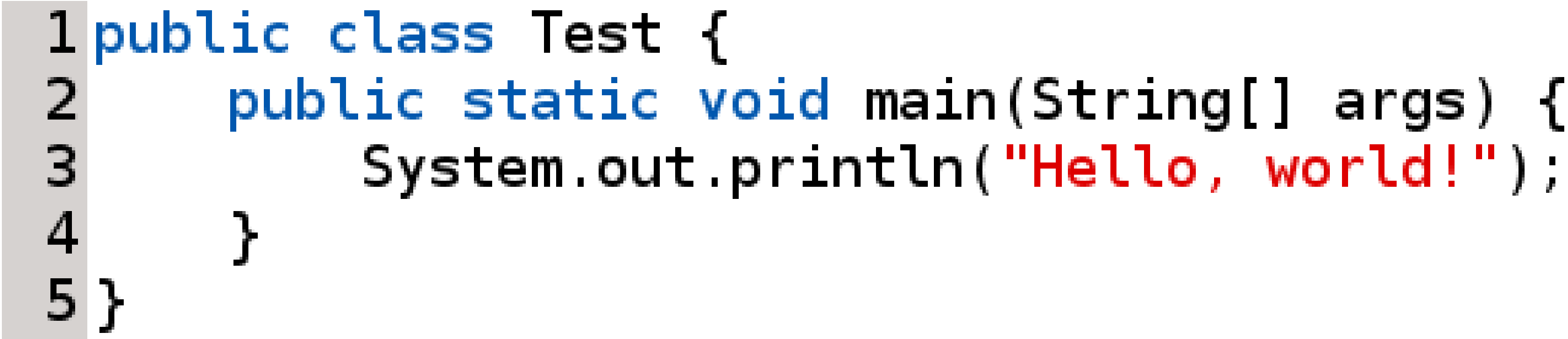
Compilers & Interpreters (Cont'd)

- Two types of translators: compilers & interpreters
- Compiler:
  - Converts entire source code into a binary executable file (0s and 1s)
  - Compilation fails in the presence of even a single error
  - Binary executables => fast execution
  - But not portable to machines of different architectures!
  - Security concerns when executables are downloaded from the Internet

Compilers & Interpreters (Cont'd)

- Interpreter:
  - Reads a line of code, and executes it
  - Stops when an error is encountered, if any
  - Relatively slow speed of execution
  - Since no binaries involved, can be executed anywhere!
- Examples:
  - Compiled languages: C, C++, and others
  - Interpreted: Python, Shell scripting, and so on

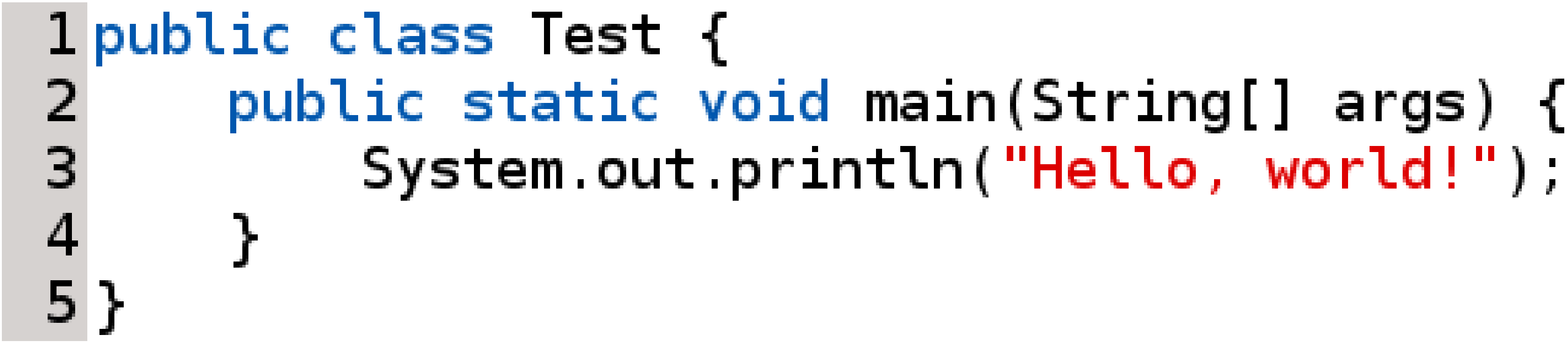
Hello, world!



```java
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- Like many other languages, main() is the entry point to a Java program
- The return type is void because main() does not return anything

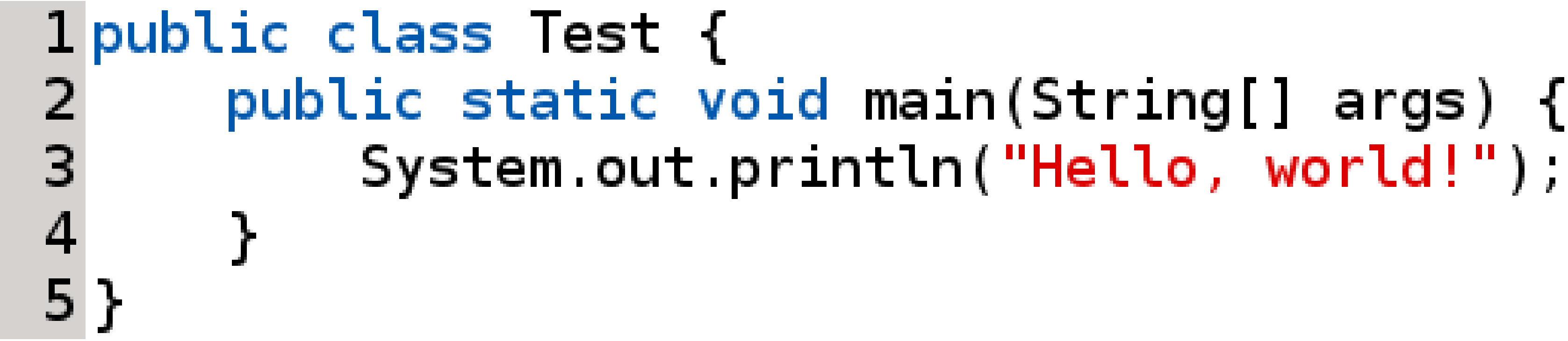
HELLO, WORLD! (CONT'D)



```java
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- main() takes an array of strings as (command line) arguments
- Unlike C, String is a built-in data type (object to be correct)
- The keyword static indicates that one should be able to invoke main() without instantiating any class

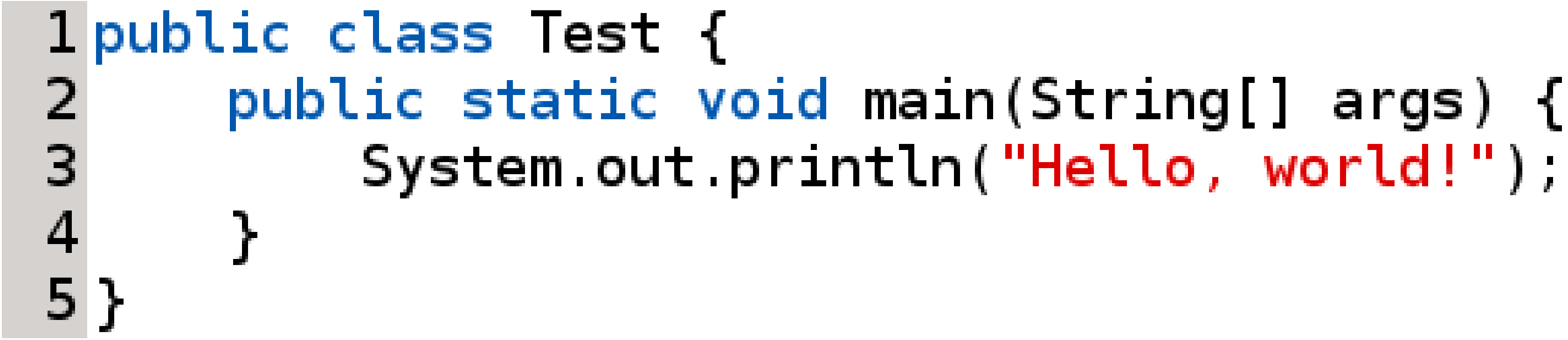
HELLO, WORLD! (CONT'D)



```
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- The keyword public is an access specifier; indicates that the OS should be able to invoke the method main()
- System.out.println() displays a String object in the terminal; appened with a new line
- Curly braces are used to denote a block of code like C

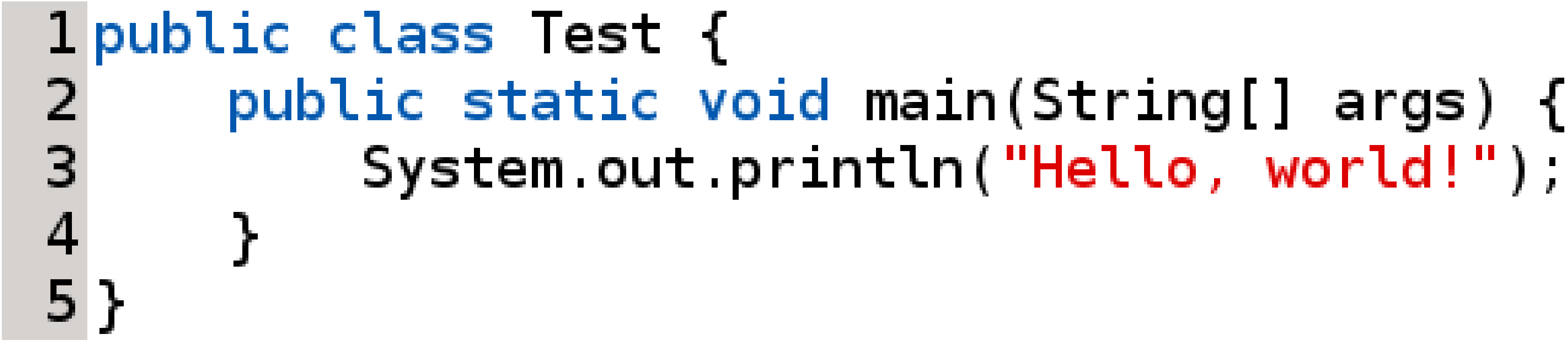
Hello, world! (Cont'd)



```java
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- The main() method resides inside a class named Test
- The class is public; anybody can access it

Hello, world! (Cont'd)



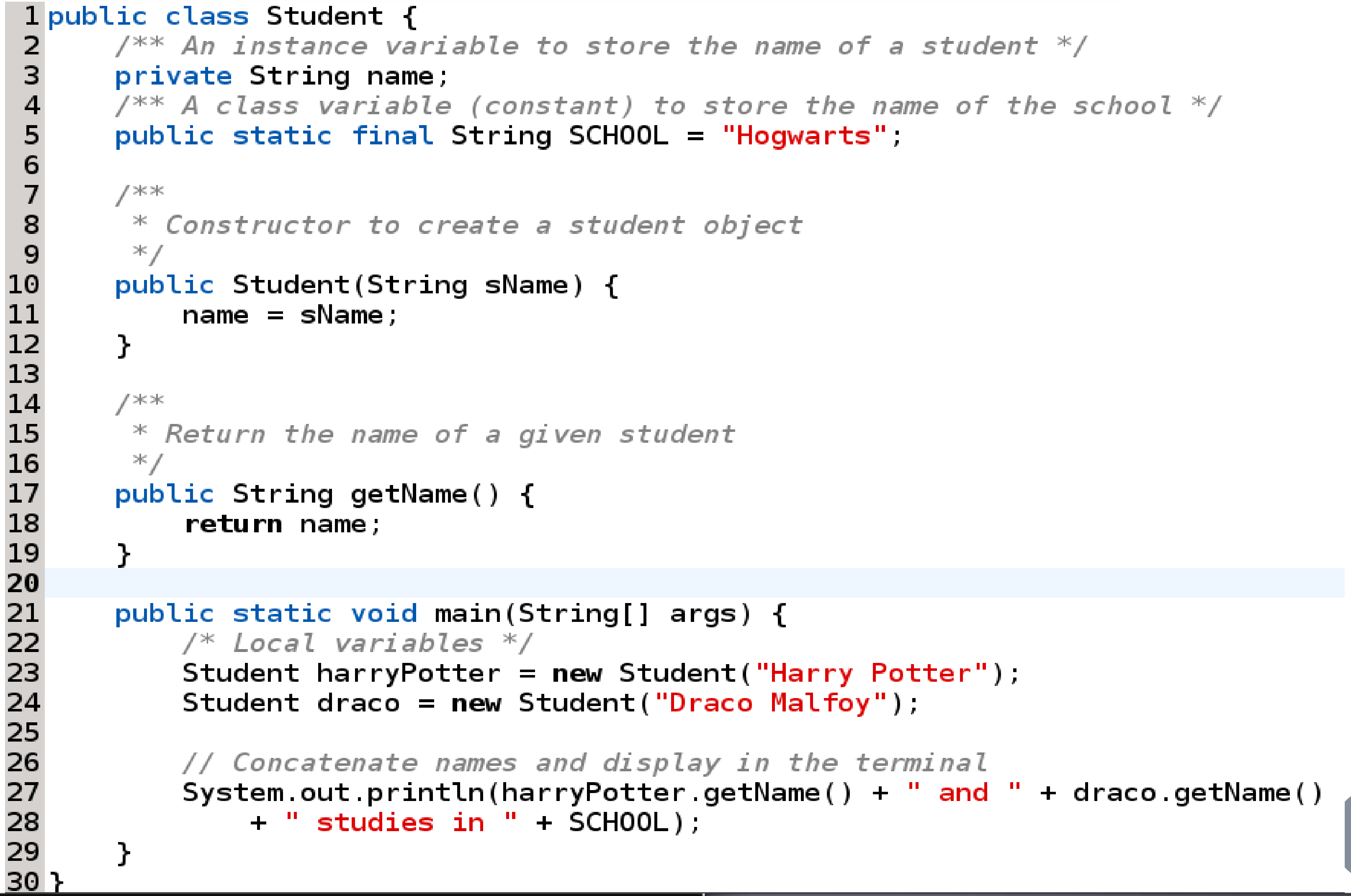
```java
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- When a class is declared as public, it must be written in a file with file name exactly as the class name
  - The above code should be written in a file called Test.java

Running a Java Program

- Step #1: compile the source code:
  - `javac Test.java`
  - Shows errors & warnings, if any
  - Produces a file called Test.class otherwise
- Step #2: interpret the bytecode:
  - `java Test`
  - No .class extension is given
  - Here, the output is: Hello, world!
- Repeat step #s 1 & 2 on any code change

Running a Java Program



```java
public class Student {
    /** An instance variable to store the name of a student */
    private String name;
    /** A class variable (constant) to store the name of the school */
    public static final String SCHOOL = "Hogwarts";

    /**
     * Constructor to create a student object
     */
    public Student(String sName) {
        name = sName;
    }

    /**
     * Return the name of a given student
     */
    public String getName() {
        return name;
    }

    public static void main(String[] args) {
        /* Local variables */
        Student harryPotter = new Student("Harry Potter");
        Student draco = new Student("Draco Malfoy");

        // Concatenate names and display in the terminal
        System.out.println(harryPotter.getName() + " and " + draco.getName()
            + " studies in " + SCHOOL);
    }
}
```

Statics

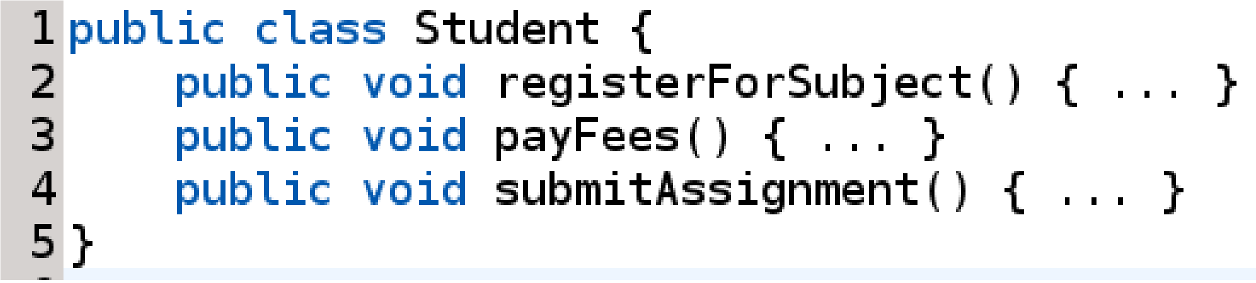
- What can be declared as static?
  - Members
  - Methods
  - Block of code
- Static member/methods are properties of the class, NOT of the objects
- Static entities are usually accessed via class name, e.g., Student.SCHOOL
- Static blocks are executed only once when a class is loaded
- Constants are usually declared as static because their values remain the same for all the objects of a given class

ACCESS MODIFIERS (CONT'D)

Access Modifier	Class	Classes in the same package	Subclass outside the package	Any class
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
<none>	Yes	Yes	No	No
private	Yes	No	No	No

Abstraction

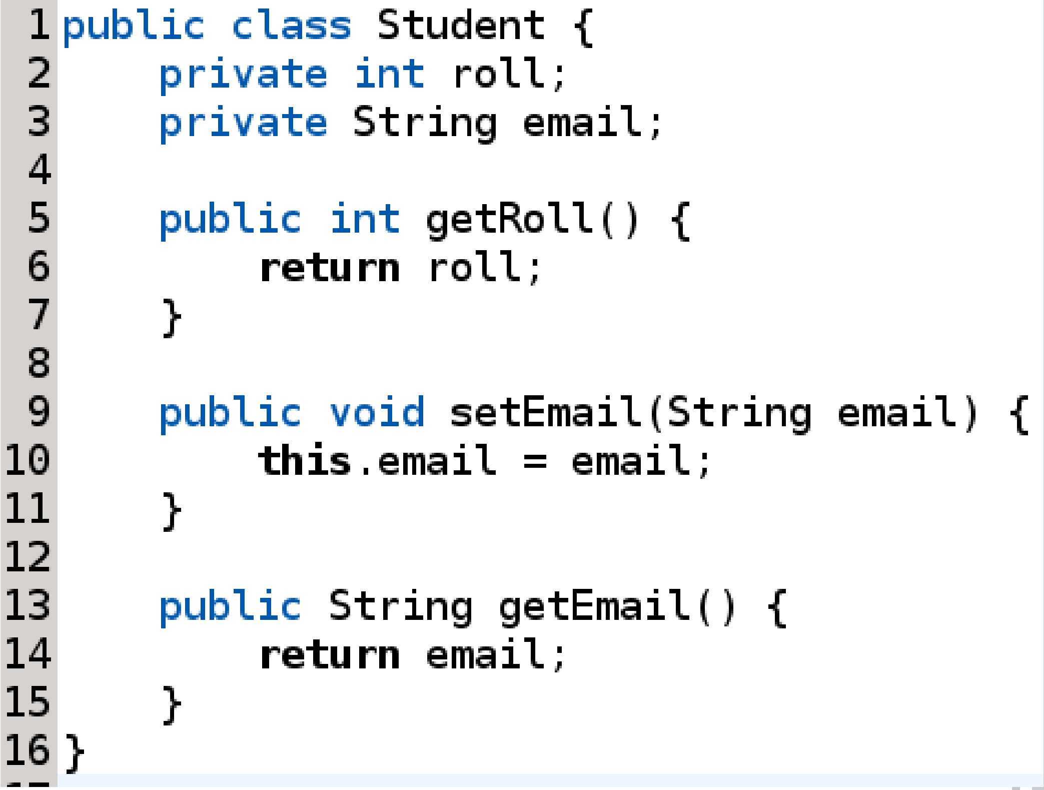
- Objects interact with the external world via its methods/behaviors
  - Only observable methods (i.e., not private)
- Examples:
  - We don't need to understand every tiny details about a phone to make a call – just touch the relevant buttons
  - A student can register for a subject; we don't need to know exactly how



```java
public class Student {
    public void registerForSubject() { ... }
    public void payFees() { ... }
    public void submitAssignment() { ... }
}
```

Encapsulation

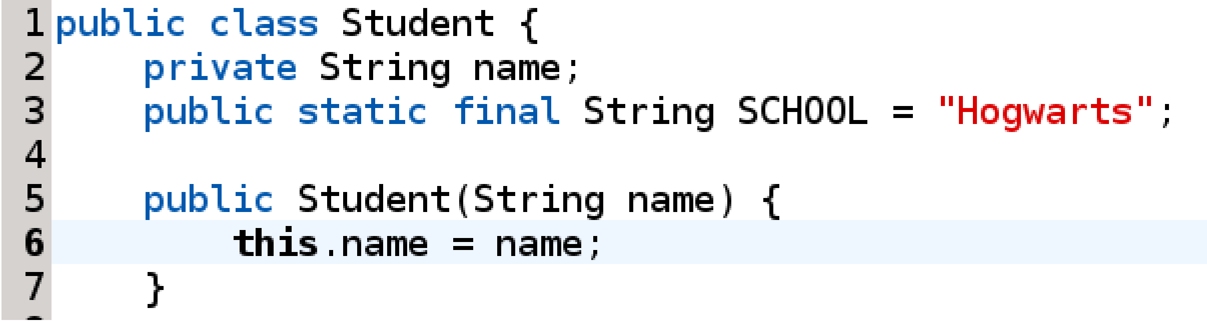
- Wrap data with behaviors to prevent exposing actual data to the real world (information hiding)
- Abstraction & encapsulation are complementary concepts
- Example: We can only read the roll of a student, but can't modify it
- Achieved via getters and setters



```java
public class Student {
    private int roll;
    private String email;

    public int getRoll() {
        return roll;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }
}
```

What is this? (Cont'd)

- this:
  - Java keyword
  - Reference to the current object
  - Used to avoid ambiguity in parameters name
  - Used to invoke the constructor as this()
  - Pass current object as argument to a method
  - Many other use cases



```java
public class Student {
    private String name;
    public static final String SCHOOL = "Hogwarts";

    public Student(String name) {
        this.name = name;
    }
```

DEFAULT CONSTRUCTOR (CONT'D)

- The output is:
  - null and null studies in Hogwarts
- In case no constructor is defined explicitly, a default constructor with no parameters is created
  - Initializes all instance variables with their default values
  - Since String is an object, its default value is null
- Tip: Remember to define your constructor!

No-arg Constructor

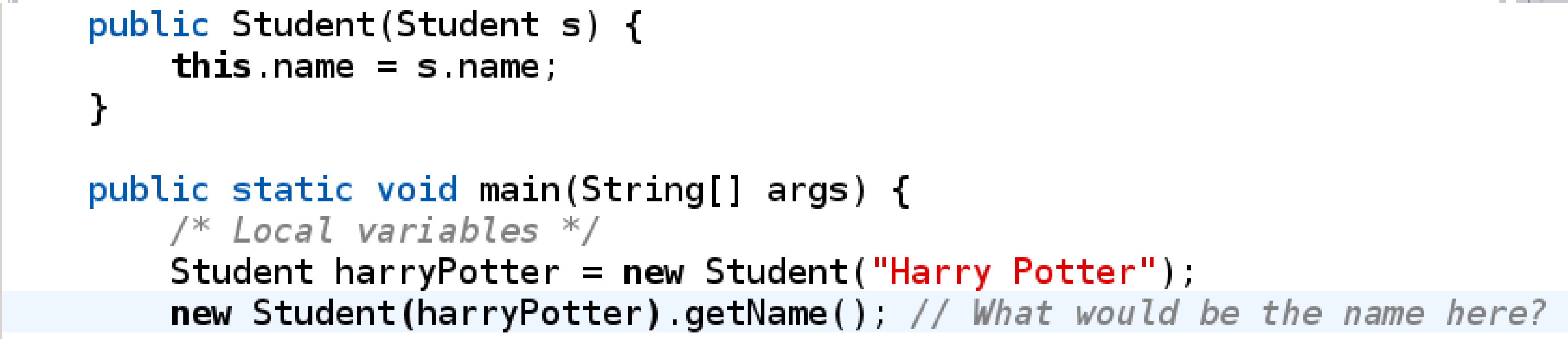
- A no-arg constructor takes no formal parameters
- Example:

```
 public Student() {
     System.out.println(“No-arg constructor invoked”);
 }
```

- The above is different from default constructor because we are explicitly defining it

Copy Constructor

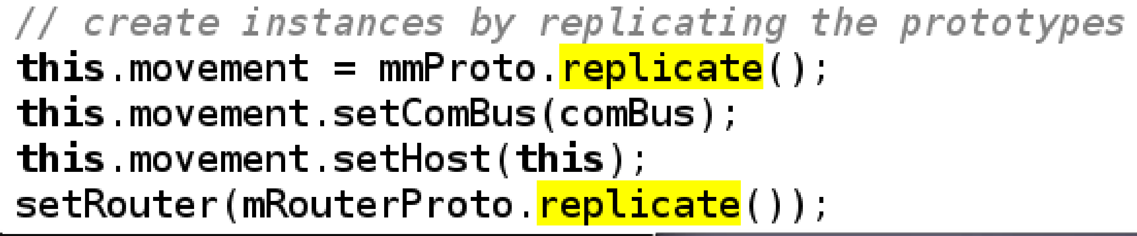
- Used to create an object by copying the values of instance variables from another existing object
- Example:



```
public Student(Student s) {
    this.name = s.name;
}

public static void main(String[] args) {
    /* Local variables */
    Student harryPotter = new Student("Harry Potter");
    new Student(harryPotter).getName(); // What would be the name here?
```

Copy Constructor (Cont'd)

- Real-life example:
  - The Opportunistic Network Environment (ONE) simulator is used to simulate Opportunistic Mobile Networks
  - At the beginning of the simulation, nodes are created, and router objects are attached to them
  - After the first router is instantiated, all others are replicated
  - The replicate() method in the routing class calls the copy constructor
  - Advantage: No need to know the type of the actual router class used here



```
// create instances by replicating the prototypes
this.movement = mmProto.replicate();
this.movement.setComBus(comBus);
this.movement.setHost(this);
setRouter(mRouterProto.replicate());
```

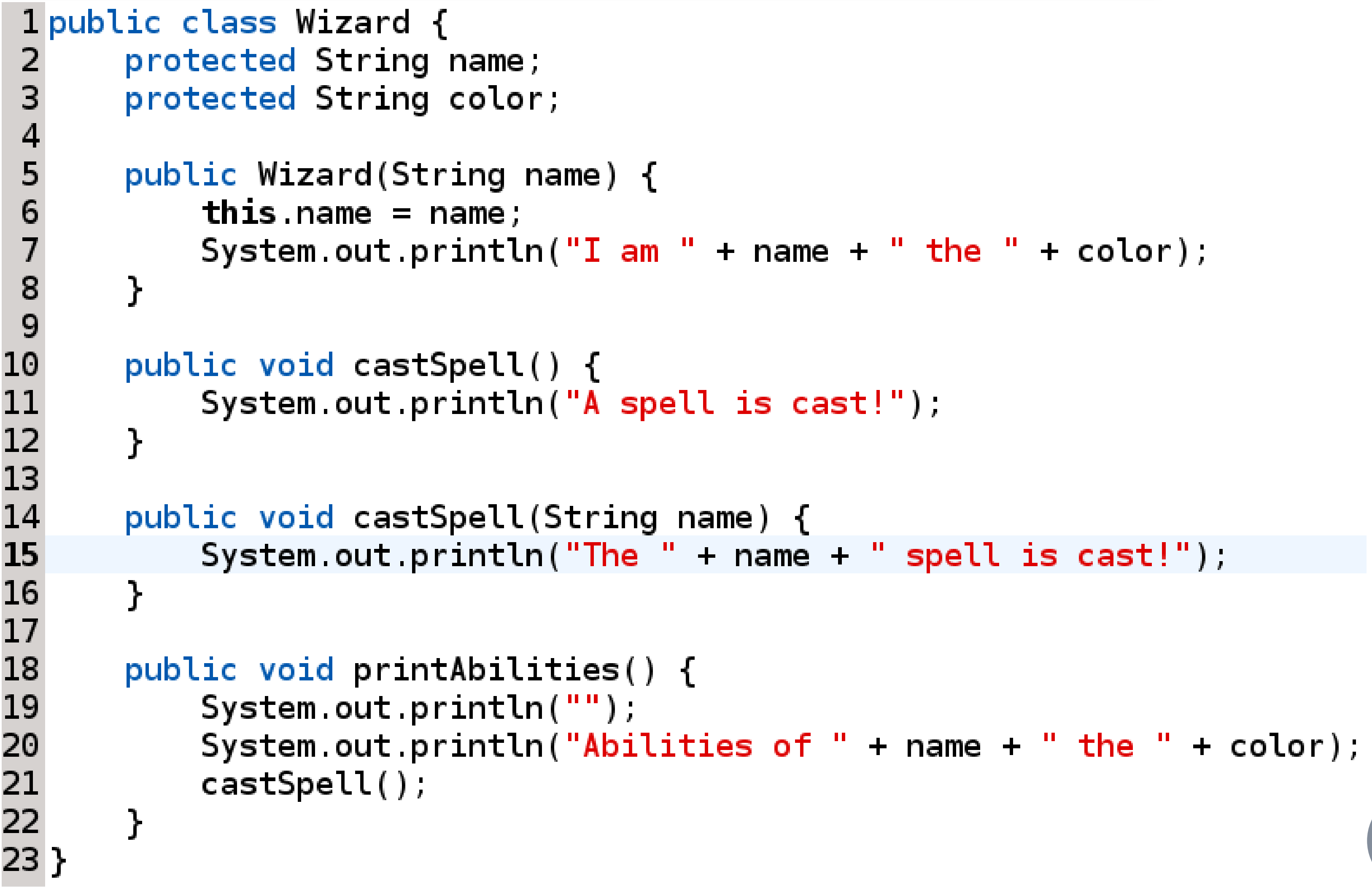
INHERITANCE

- In real world, objects aren't always distinct; they often share state and behavior
- Consider a bike (bicycle)
  - Fit with a motor => motorbike
    - Fit 4 wheels to motorbike => motorcar
      - Replace motor of cars with horse => chariot
- What is inheritance?
  - Child (sub or derived) classes automatically receive the states & behaviors of their parent (super or base) classes
  - Subclasses "specialize" the behavior so inherited
- In theory, a subclass can inherit from any no. of superclasses
  - Java allows inheritance only from a single superclass

The Wizards of Middle Earth

- Let us consider the different categories of wizards (with due apologies to J. R. R. Tolkien)
- A wizard can cast a spell and has other abilities
  - A brown wizard can read mind of others
  - Grey and white wizards can not only read minds, but also influence them in the process
  - Both grey and white wizards can fight Balrogs
  - A white wizard is more wise
- Let's look at the corresponding Java classes

The Wizard Class

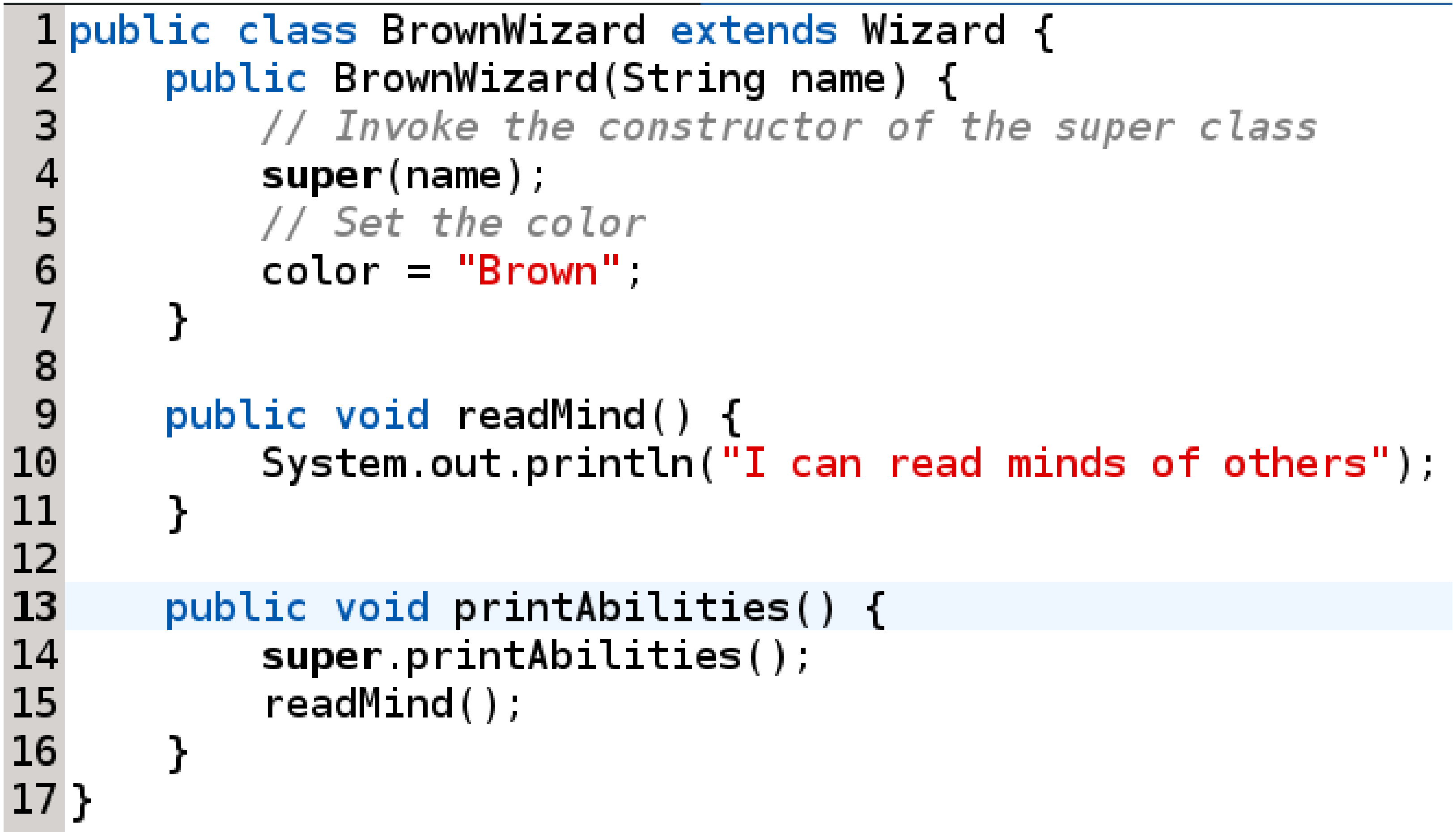


```java
public class Wizard {
    protected String name;
    protected String color;

    public Wizard(String name) {
        this.name = name;
        System.out.println("I am " + name + " the " + color);
    }

    public void castSpell() {
        System.out.println("A spell is cast!");
    }

    public void castSpell(String name) {
        System.out.println("The " + name + " spell is cast!");
    }

    public void printAbilities() {
        System.out.println("");
        System.out.println("Abilities of " + name + " the " + color);
        castSpell();
    }
}
```

THE WIZARD CLASS (CONT'D)

- There are two instance variables – name and color
  - color is protected because subclasses will set the value
- The castSpell() method is defined in this class
  - All subclasses would inherit it
- The printAbilities() method prints all the abilities of a Wizard object
  - Must be public to invoke it
- There are two methods with the name castSpell() in this class
  - Their number & type of parameters are different
  - This is called as method **overloading**

The BrownWizard Class

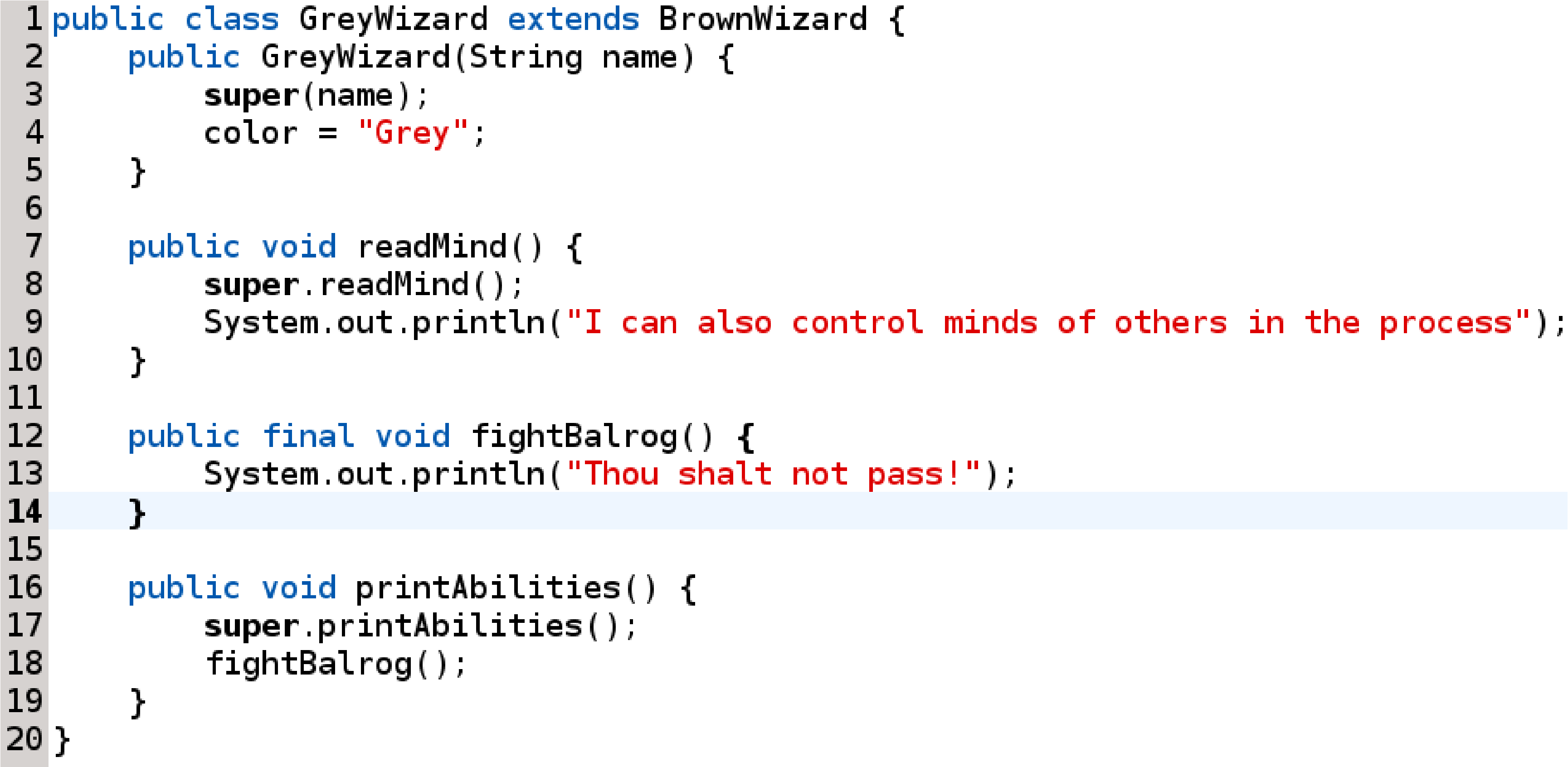


```java
public class BrownWizard extends Wizard {
    public BrownWizard(String name) {
        // Invoke the constructor of the super class
        super(name);
        // Set the color
        color = "Brown";
    }

    public void readMind() {
        System.out.println("I can read minds of others");
    }

    public void printAbilities() {
        super.printAbilities();
        readMind();
    }
}
```

THE BROWNWIZARD CLASS (CONT'D)

- The super(name) call invokes the constructor of the parent class
  - Must be called explicitly with correct number & type of arguments when there is no no-arg constructor
  - Must be the first statement of the constructor of a child class
- The color property of the wizard is set here
- The printAbilities() method is again defined here
  - This is known as method **overriding**
  - To override a method, its name, return type, and number & type of parameters must be **exactly** same as in the parent class
- printAbilities() refers to the method in the current class
  - super.printAbilities() refers to the method in the parent class

The GreyWizard Class

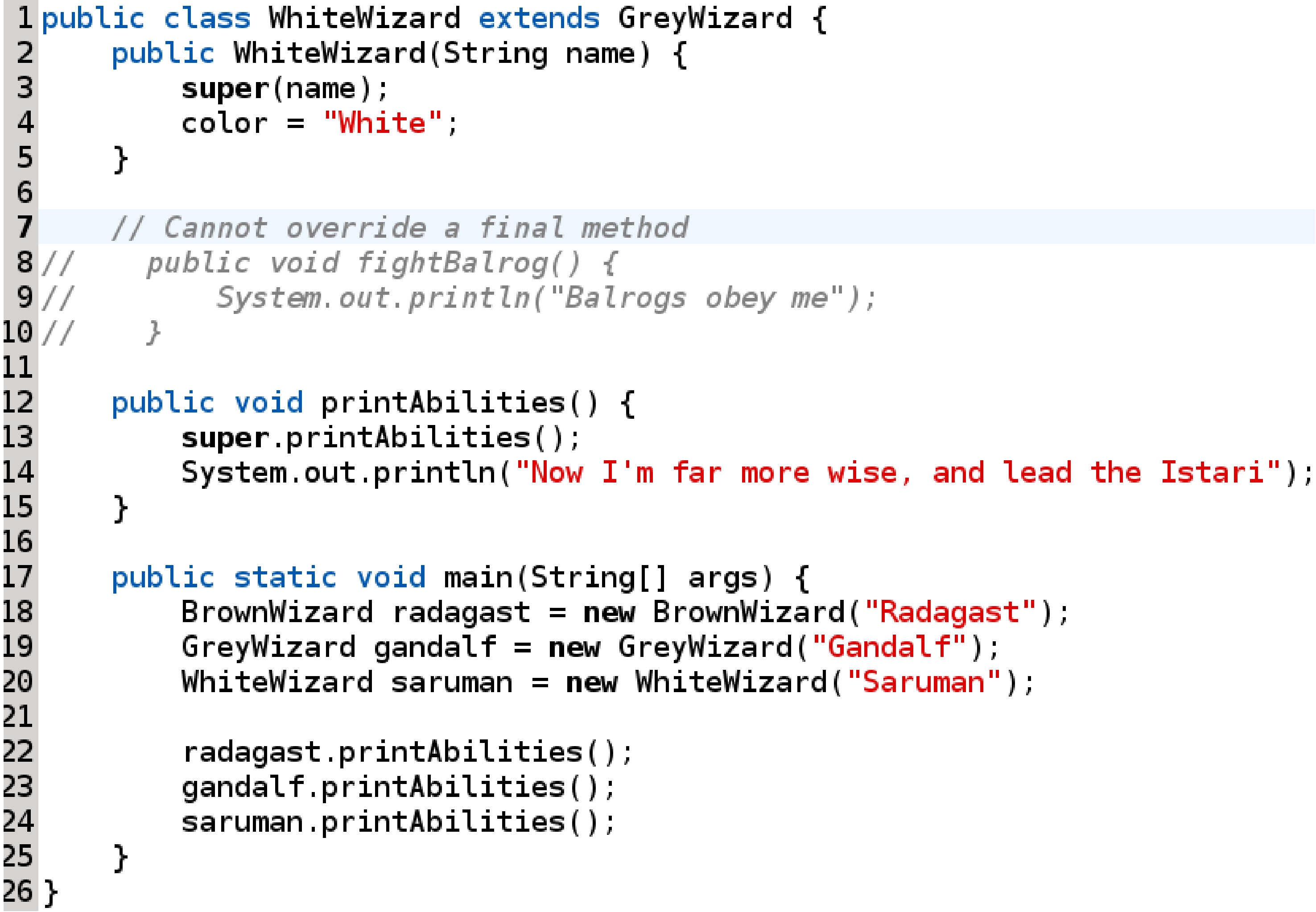


```java
public class GreyWizard extends BrownWizard {
    public GreyWizard(String name) {
        super(name);
        color = "Grey";
    }

    public void readMind() {
        super.readMind();
        System.out.println("I can also control minds of others in the process");
    }

    public final void fightBalrog() {
        System.out.println("Thou shalt not pass!");
    }

    public void printAbilities() {
        super.printAbilities();
        fightBalrog();
    }
}
```

THE GREYWIZARD CLASS (CONT'D)

- Once again we override the printAbilities() method in this class
- The fightBalrog() method is declared as final
  - No child class can override this method
- How to prevent a class from being inherited?
  - Declare the class as final

The WhiteWizard Class



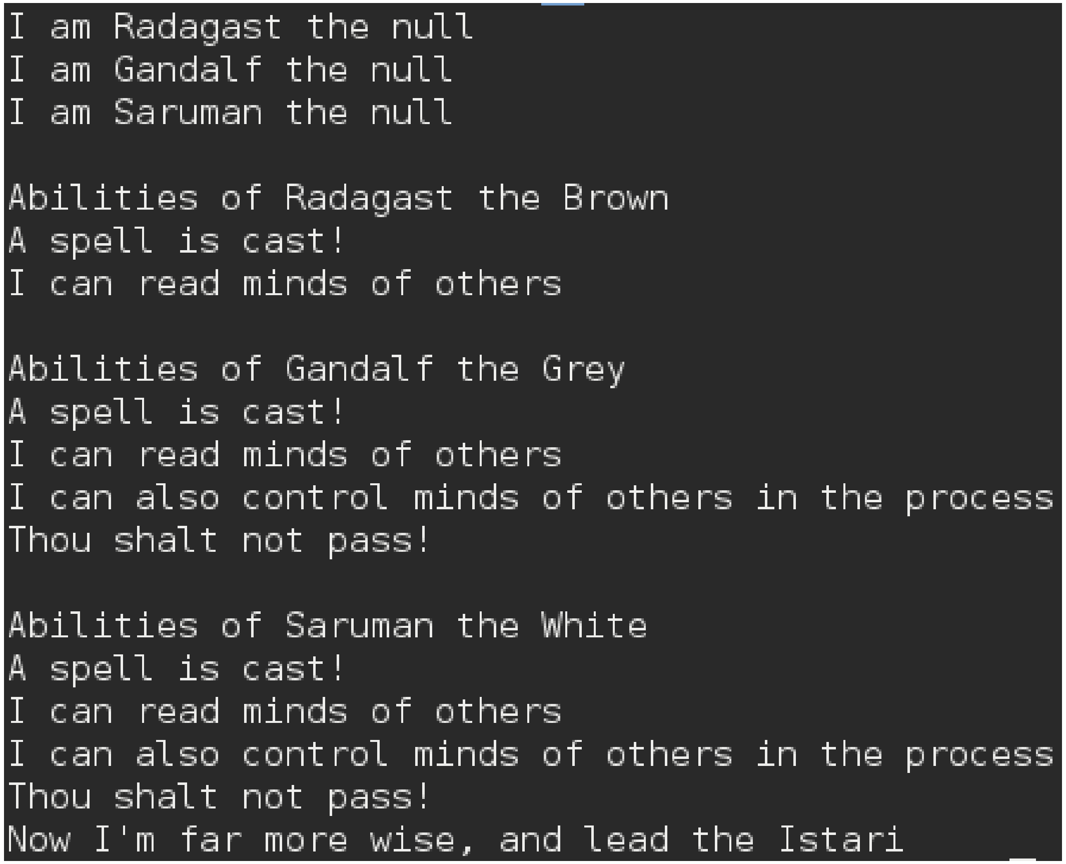
```java
public class WhiteWizard extends GreyWizard {
    public WhiteWizard(String name) {
        super(name);
        color = "White";
    }

    // Cannot override a final method
//    public void fightBalrog() {
//        System.out.println("Balrogs obey me");
//    }

    public void printAbilities() {
        super.printAbilities();
        System.out.println("Now I'm far more wise, and lead the Istari");
    }

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

        radagast.printAbilities();
        gandalf.printAbilities();
        saruman.printAbilities();
    }
}
```

The WhiteWizard Class (Cont'd)

- The main() method in this class creates three wizards of different types
  - Abilities of the corresponding wizards are printed

OUTPUT

```
javac
  *Wizard.java
java
  WhiteWizard
```



```
I am Radagast the null
I am Gandalf the null
I am Saruman the null

Abilities of Radagast the Brown
A spell is cast!
I can read minds of others

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!

Abilities of Saruman the White
A spell is cast!
I can read minds of others
I can also control minds of others in the process
Thou shalt not pass!
Now I'm far more wise, and lead the Istari
```

OUTPUT EXPLAINED

- Names of the wizards are printed correctly, but not color
  - The members name & color are not initialized in Wizard
    - Null values by default
  - Name and color are printed inside the constructor of the Wizard class
  - Subclasses (e.g., BrownWizard) passes name as an argument to the constructor => parent class' constructor prints the name
  - The color is assigned **after** the call to the constructor of the superclass
    - The print statement has already executed by then => color is printed as null
  - Colors take effect once the execution of the constructors of the child classes are over
    - Prints color in the subsequent statements correctly

Output Explained (Cont'd)

- How to print the color correctly inside the constructor?
  - Pass it as another parameter to the constructor
- Each child class inherits one or more behavior from its parent class
  - WhiteWizard prints more abilities than GreyWizard, which prints more than BrownWizard
- Is Wizard a sub class of any other class?
  - Yes! All classes in Java are subclasses of the Object class

Thank you!