2023-02-27

9.3 Overriding Methods

Overriding Methods

Method Overriding - To implement a new version of a method to replace code that would otherwise have been inherited from a superclass

To override a method from a superclass, implement the method in the subclass.

Overriding example

```
public class Turtle {
  private int x, y;

  public void forward(int z) {
    x += z;
  }

  public int getX() {
    return x;
  }
  public void setX(int x) {
    this.x = x;
  }
}
```

```
public class TurboTurtle extends Turtle {
  public void forward(int z) {
    setX(getX() + 100*z);
  }
}

// QUESTION: Why did we have to use the
// accessor methods in TurboTurtle instead
// of just saying x += 100*z?
```

Confuseth them not: Overloading and Overriding

We learned previously about Method Overloading: defining methods with the same name but different method signatures.

When you overload a method, you implement a new method with the same name but different parameters.

Overriding a method is very different than overloading a method. **Overriding replaces an inherited method, overloading creates a complementary method with the same name.**

```
public class Turtle {
  private int x;
  private int y;

  public void forward() {
    x += 5;
  }

  public void forward(int z) {
    x += z;
  }
}
```

Failure To Override

To override a method, you must match the method signature exactly. (Parameter names can be different... but the method name, return type, and parameter types must all match.)

If you don't match it up, you may accidentally create an overloaded method! And the compiler will not warn you.

```
class Base {
 public void method() {
   System.out.println("Base.method");
 }
}
class OverrideTest extends Base {
 public void methud() {
   System.out.println("OverrideTest.method");
 }
 public static void main(String args[]) {
   new OverrideTest().method();
 }
}
```

Accidental typo

@Override

Annotations are additional information that can be specified on Java methods, variables and classes. They start with @

@Override is an annotation which tells the compiler that you're trying to override a method... and if your declaration doesn't actually override a method, you get a helpful compile error.

(Why an annotation, not a keyword like C#? Adding keywords to an existing language is hard.)

```
class Base {
 public void method() {}
}
class OverrideTest extends Base {
 @Override
 public void methud() {}
 public static void main(String args[]) {}
}
```

Accidental typo

```
OverrideTest.java:5: error: method does
not override or implement a method from a
supertype
 @Override
 ^
1 error
```

Why?

What are uses of overriding?

The `Object` Superclass

All classes are subclasses of the `Object` class.

(Not necessarily direct subclasses... there are often some superclasses in between.)

When you code, you want to use as specific a type as you can, but this is legal:

```
Object myObject = "Hello, world!";
```

because every String is also an Object.

You also can assign a String to an Object variable, or otherwise use a String in any Object context. It is always legal and requires no special syntax to convert a reference to a superclass reference:

```
String myString = "Hello, world!";
Object myObject = myString;
```

The `Object` Superclass

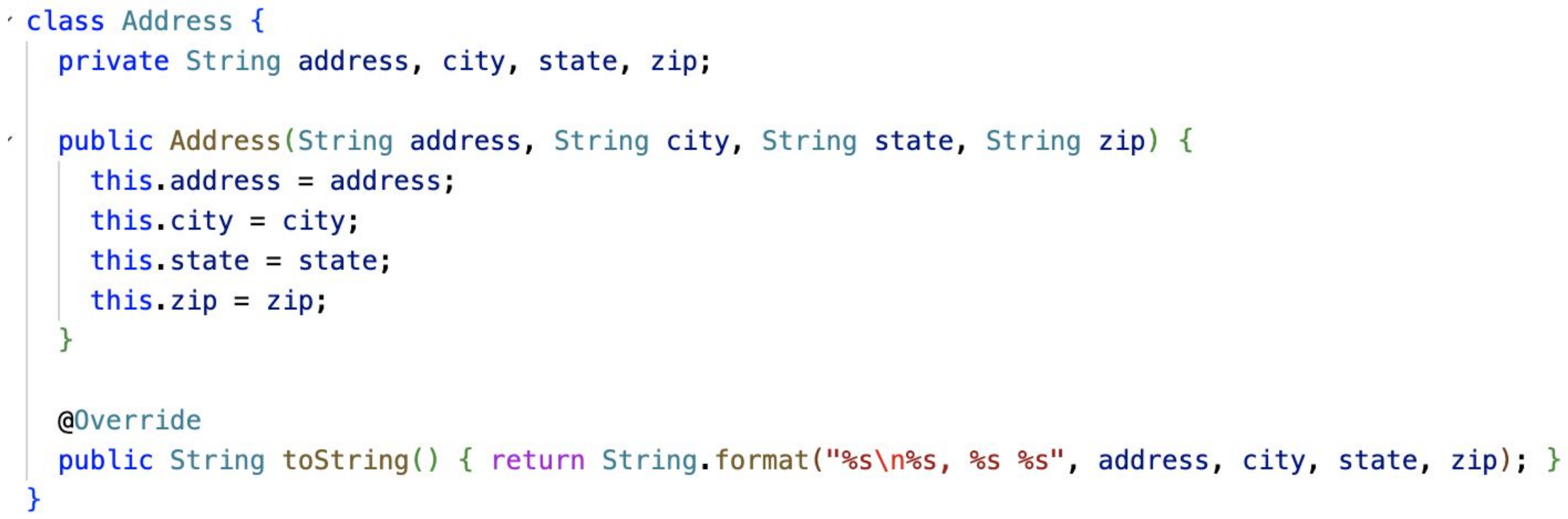
The `Object` class lives in the java.lang package, which is where classes fundamental to the Java language live, like java.lang.String.

`Object` has several public methods. This means that all Java objects inherit these methods. Many of them can be overridden.

Some of the most common ones to override are:

- `public String toString();`
- `public boolean equals(Object o);`

Overriding the `toString` method



```java
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}
```

Overriding the `equals` method

Note that the equals method compares the object with type Object.

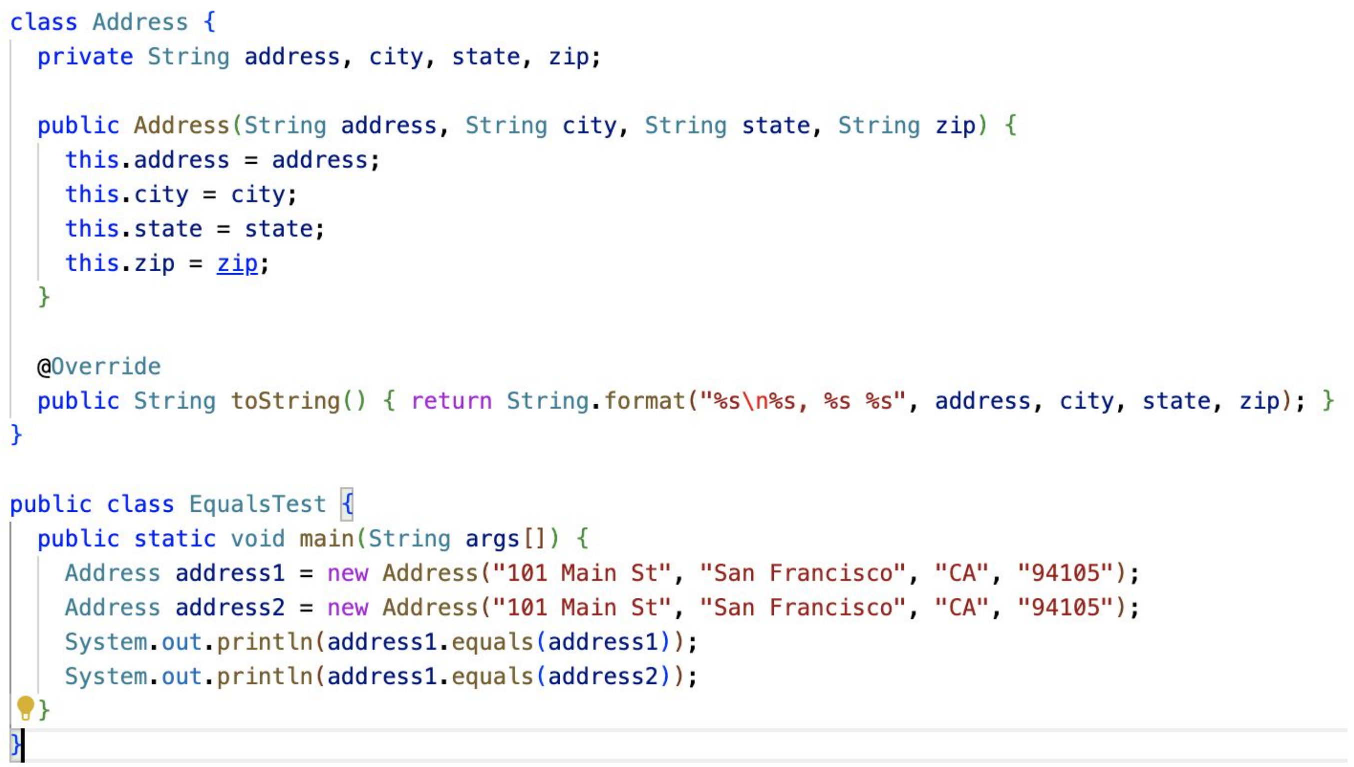
So any object can be compared with any other object, of any class, using the equals method!

However, you often want your object to only be equal to objects of the same class, and you may need to compare member variables specific to your class.

What happens if you don't override Object.equals? The default implementation is essentially the same as == on reference types: It returns true if the other object is the exact same object instance.

```
@Override
public boolean equals(Object o)
{
  // returns true or false
}
```

Object.equals – What does this code do?



```java
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}

public class EqualsTest {
  public static void main(String args[]) {
    Address address1 = new Address("101 Main St", "San Francisco", "CA", "94105");
    Address address2 = new Address("101 Main St", "San Francisco", "CA", "94105");
    System.out.println(address1.equals(address1));
    System.out.println(address1.equals(address2));
  }
}
```

Casting

The type cast operator () makes it possible to convert a reference of superclass type (such as Object) to a subclass type (such as Student).

This only works if the superclass reference really IS pointing to an instance of the subclass!

If it isn't, a ClassCastException will be thrown.

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

Bug: This could throw ClassCastException, if one passes, say, a Date or String to this method.

instanceof operator

The instanceof operator lets you check the "is-a" relationship of an object with a class.

x instanceof T evaluates to true if the object reference x is of type T.

Using instanceof, we can check that the cast to Student is safe before doing it.

An alternative is to try/catch ClassCastException, but that's more expensive.

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

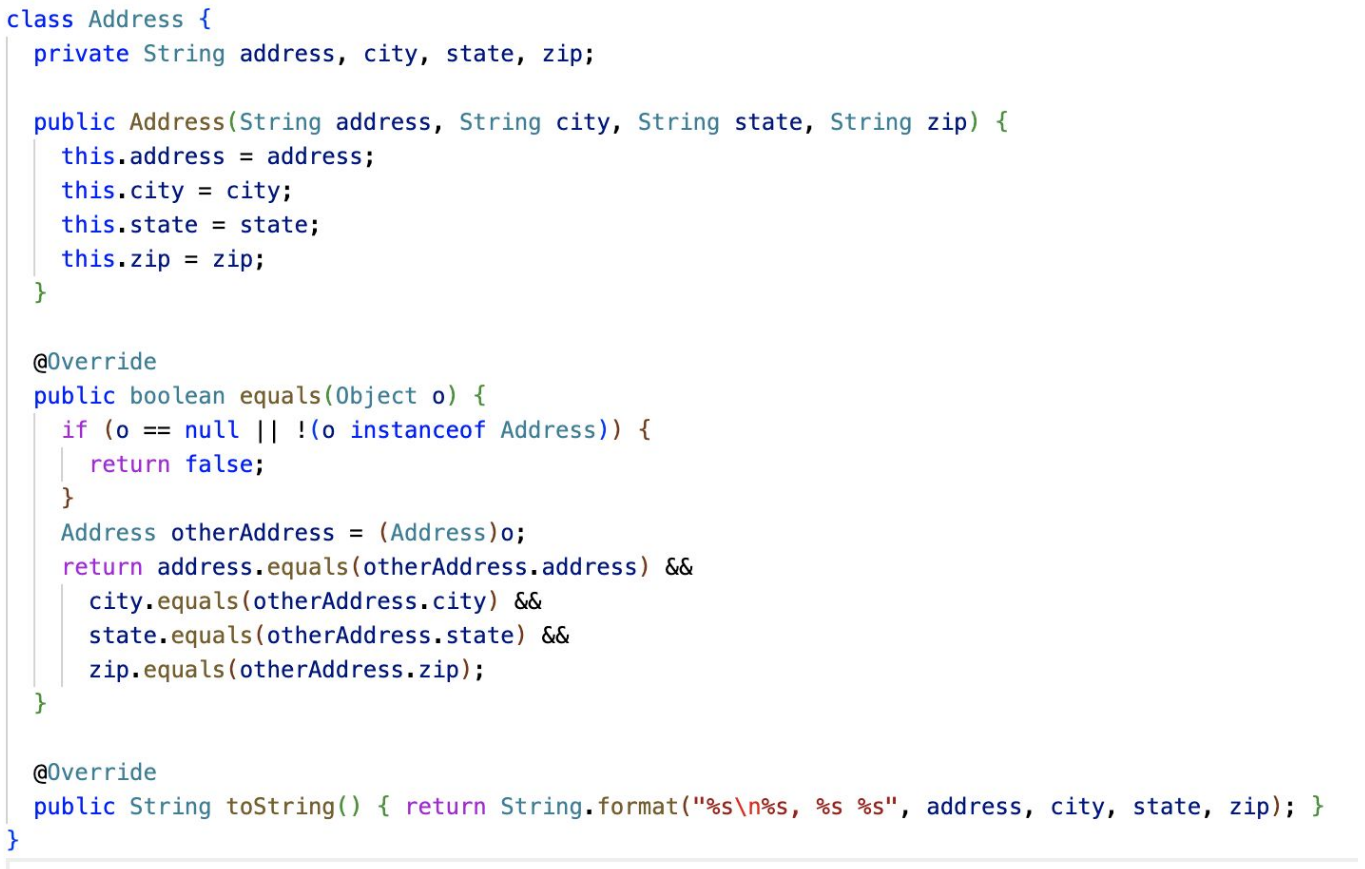
Object.equals contract

Java defines a contract that Object.equals implementations must follow:

Reflexive: x.equals(x) is true
Symmetric: if x.equal(y) is true, y.equals(x) is true
Transient: if x.equals(y) and y.equals(z) are true, x.equals(z) is true
Consistent: x.equals(y) should return the same thing if you call it again, if nothing about them changed
Handles null: x.equals(null) should return false. (And it shouldn't crash with a NullPointerException!)

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

Overriding Object.equals



```java
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Address)) {
      return false;
    }
    Address otherAddress = (Address)o;
    return address.equals(otherAddress.address) &&
      city.equals(otherAddress.city) &&
      state.equals(otherAddress.state) &&
      zip.equals(otherAddress.zip);
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}
```

String overrides Object.equals

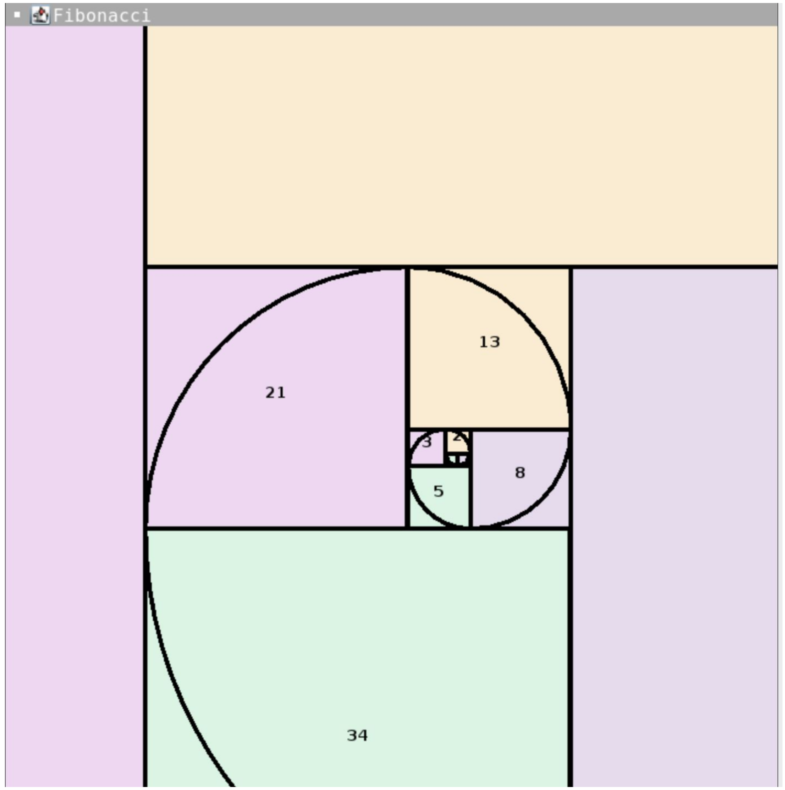
```
String z = “z”;
String a = z + z;
String b = “zz”;
String c = b;

a == b; // false because a and b refer to different strings
b == c; // true because c and b refer to the same strings

a.equals(b); // true because the values of a and b are the same
c.equals(b); // true because c and b refer to the same string
```

More examples of overriding...

Overriding in Java AWT (Abstract Window Toolkit)

```java
public class Spiral extends JPanel implements ActionListener {
  private Timer timer;
  private double scale = 100.0;

  public Spiral() {
    timer = new Timer(40, this);
    timer.start();
  }

  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    RenderingHints rh = new RenderingHints(
             RenderingHints.KEY_TEXT_ANTIALIASING,
             RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2.setRenderingHints(rh);
    g2.setStroke(new BasicStroke(4));
    g2.clearRect(0, 0, getWidth(), getHeight());
    g2.setFont(new Font("Helvetica", Font.BOLD, 14));

    Fibonacci fibonacci = new Fibonacci();
```

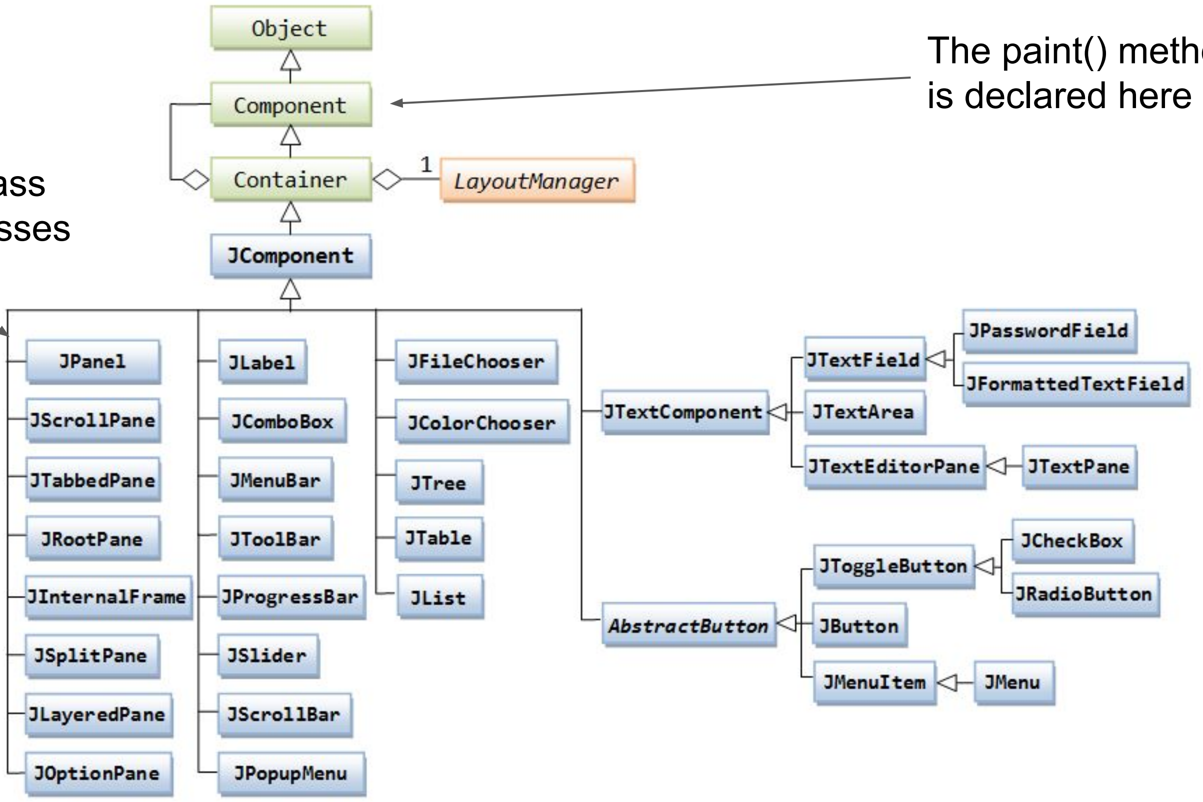
Override



Java AWT/Swing Class Hierarchy

The paint() method we are overriding is declared here

This is the class Spiral subclasses



Servlets - Handling HTTP requests

Client (browser) sends HTTP request...

```
GET / HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/110.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;
q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br..Accept-Language: en-US,en;q=0.9
```

Server replies with HTTP response...

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.10.8
Date: Mon, 27 Feb 2023 06:07:31 GMT
Content-type: text/html; charset=utf-8
Content-Length: 5595
```
(HTML follows)

Servlets - Handling HTTP requests

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class DemoServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  {
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    pw.println("<html><body>");
    pw.println("Hello, world!");
    pw.println("</body></html>");
    pw.close();
  }
}
```

Override

9.4 super.method()

super.method()

We saw super() already in Section 9.2. In that case, super() is used for what’s called “constructor chaining,” where a subclass constructor calls a superclass constructor.

The super keyword can also be used to invoke a superclass's version of a method, even if the subclass overrides the method.

Often, an overridden method wants to do everything the original method did, but add on some additional behavior.

super.method()

```
class Person {
  ...
  public void dump(PrintWriter pw) {
    pw.println("Name: " + name);
  }
}
class Teacher extends Person {
  private String classroom;
  ...
  public void dump(PrintWriter pw) {
    super.dump(pw);
    System.out.println("Classroom: " + classroom);
  }
}
```

super.method() and super() differences

If you don't use super() to do constructor chaining in a subclass, Java essentially does it for you. It will add an implicit super() call to the no-param constructor of the superclass, if one exists.

This is done because Java regards constructors as really important to getting a properly initialized object. You can't skip around a superclass constructor.

Method overrides are different. Java lets you completely replace the definition of a method. The new method code has a choice: It can use super() to call the superclass version of the method at some point, or not.

super.method()

```
class NPC {
  ...
  public void tick() {
    // Implements random movement of the NPC around the map
  }
}
class Teacher extends NPC {
  ...
  public void tick() {
    super.tick(); // call super.tick() to get the basic NPC behavior like random movement
    // Additional code here for special NPC behavior specific to this character
  }
}
```

You don't need super.method() all the time!

In a subclass, you can invoke any public or protected method of that class's superclasses.

You don't need to say super.method() to get at these methods. You can just say method().

You only need super.method() when your subclass has overridden method(), but you still need to invoke the original version of the method.

super.method() is usually called from the body of the subclass's implementation of method()!

Replit: OverrideMusic

In today's replit, there will be some **abstract classes**.

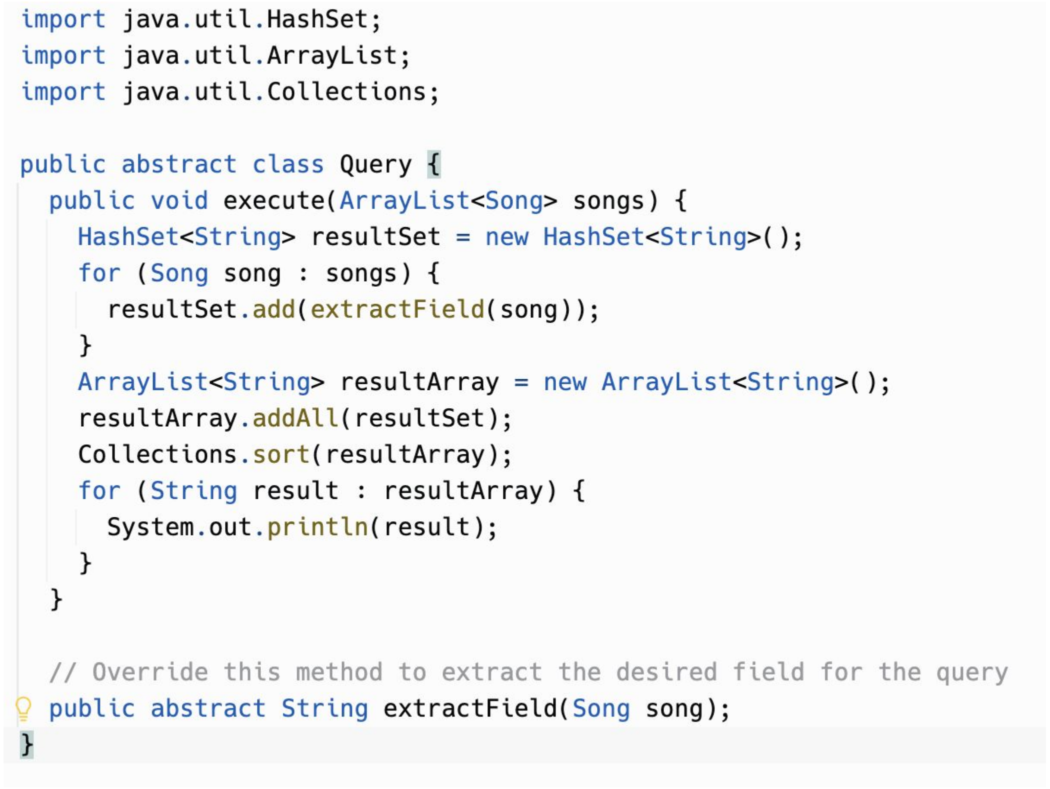
An abstract class cannot be new-ed... you have to subclass it and define the abstract methods, in this case processRecord.

```java
public abstract class FileScanner {
  public abstract void processRecord(String[] fields);

  public boolean processFile(String path) {
    Scanner scanner;
    try {
      scanner = new Scanner(new FileInputStream(path));
    } catch (IOException e) {
      System.out.println("Error! Could not open music file " + path);
      e.printStackTrace();
      return false;
    }

    while (scanner.hasNextLine()) {
      String line = scanner.nextLine();
      line = line.trim();
      if (line.length() == 0) {
        continue;
      }
      String[] fields = line.split(",");
      processRecord(fields);
    }

    scanner.close();
    return true;
  }
}
```

Reuse through subclasses

Subclassing is another means of reusing code. This Query class has the code for taking a set of values from the song list, unique-ifying them, and sorting them and printing them.

To use it, you have to subclass it and override extractField to say which field to extract.



```java
import java.util.HashSet;
import java.util.ArrayList;
import java.util.Collections;

public abstract class Query {
  public void execute(ArrayList<Song> songs) {
    HashSet<String> resultSet = new HashSet<String>();
    for (Song song : songs) {
      resultSet.add(extractField(song));
    }
    ArrayList<String> resultArray = new ArrayList<String>();
    resultArray.addAll(resultSet);
    Collections.sort(resultArray);
    for (String result : resultArray) {
      System.out.println(result);
    }
  }

  // Override this method to extract the desired field for the query
  public abstract String extractField(Song song);
}
```

Practice on your own

- CSAwesome 9.3 - Overriding methods
- CSAwesome 9.4 - super.method()
- Replit - OverrideMusic