Week 3: Diving into OOP

CS 151

Computer Science Study Lab + Peer Connections

https://www.sjsu.edu/cs/students/study-lab.php

226 MacQuarrie Hall

Monday - Thursday starting February 3rd

12 pm - 3 pm

See Canvas Announcements for Peer Connections Info

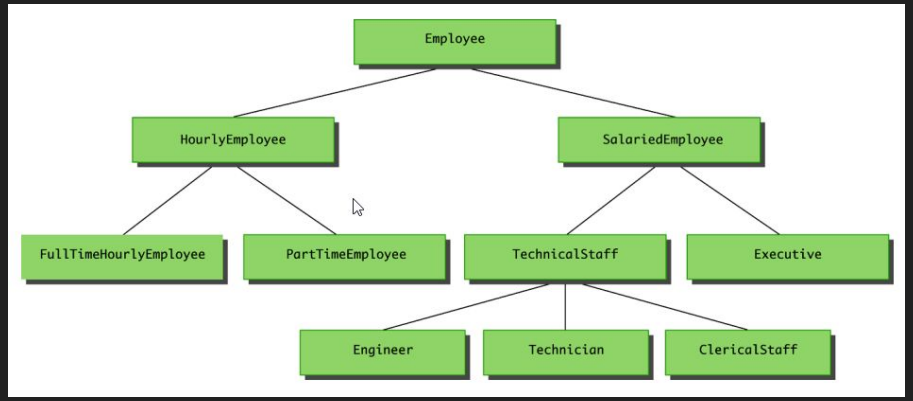
Recap

```java
public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

```java
public class Job {
    private String title;
    private Employee employee;

    public Job(String title, Employee employee) {
        this.title = title;
        this.employee = employee;
    }

    public String getJobDetails() {
        return employee.getName() + " works as a " + title;
       // return this.employee.getName() …
       // which version is better?
    }

    public void promote(String newTitle) {
        this.title = newTitle;
    }

    public static void main(String[] args) {
        Person alice = new Employee("Alice", 30);
        Job job = new Job("Engineer", alice);

        System.out.println(job.getJobDetails());
        // Output: Alice works as a Engineer

        job.promote("Senior Engineer");
        System.out.println(job.getJobDetails());
        // Output: Alice works as a Senior Engineer
    }
}
```

Inheritance

- When designing classes, there is often a natural hierarchy for grouping them
- Example: Hourly and Salaried Employees
  - Hourly: Full Time and Part Time
  - Salaried: Technical and Executive
- All Employees share some traits:
  - name, hireDate, payEmployee()
- But some methods can differ:
  - calculatePaycheck(hours) vs calculatePaycheck(salary)
- In Java, we can define an Employee class to store certain shared traits for all Employees, but also have more specific subcategories for specific cases



Inheritance

- *Inheritance* is one of the core concepts of Object-Oriented Programming
- Allows a new class to build from an existing class using `extends`
  - Parent class/superclass
  - Child class/subclass
- This is advantageous because it allows code to be reused effectively
- What does a child class mean?
  - Child class can automatically access methods and properties of the parent class
  - And can also have additional methods/properties of its own

```
class StringInstrument {
  protected String type = "String Instrument";
  public void play() {
    System.out.println("Playing the instrument!");
  }
}

class Guitar extends StringInstrument {
  private int numberOfStrings = 6;

  @Override
  public void play() {
    System.out.println("Strumming the guitar!");
  }

  public static void main(String[] args) {
    Guitar myGuitar = new Guitar();
    myGuitar.play();
    // Output: Strumming the guitar!
    System.out.println("Type: " + myGuitar.type);
    // Output: Type: String Instrument
  }
}
```

Inheritance

- Definitions for inherited variables/methods don’t explicitly appear, but they are there
- This is advantageous because it allows code to be reused effectively
- If you’re coding a class and don’t want other developers to inherit from it, use the keyword “final”
  - a final class cannot be inherited from
  - a final method cannot be redefined in a subclass, but the subclass can exist
- `final class Animal`
- `class Cat extends Animal`
  - This will throw compilation error

```
class StringInstrument {
  protected String type = "String Instrument";
  public void play() {
    System.out.println("Playing the instrument!");
  }
}

class Guitar extends StringInstrument {
  private int numberOfStrings = 6;

  @Override
  public void play() {
    System.out.println("Strumming the guitar!");
  }

  public static void main(String[] args) {
    Guitar myGuitar = new Guitar();
    myGuitar.play();
    // Output: Strumming the guitar!
    System.out.println("Type: " + myGuitar.type);
    // Output: Type: String Instrument
  }
}
```

Inheritance - Override

- `@Override`: indicates a method is intended to override a method in the superclass
  - ignore parent method and use child method
  - `@override` is not strictly necessary but
    - clarity to readers of the code
    - compiler will confirm that it does override a superclass method

```
class StringInstrument {
  protected String type = "String Instrument";

  public void play() {
    System.out.println("Playing the instrument!");
  }
}

class Guitar extends StringInstrument {
  @Override
  public void play() {
    super.play(); // Calls the base class's play method
    System.out.println("Strumming the guitar!");
  }

  public static void main(String[] args) {
    Guitar myGuitar = new Guitar();
    myGuitar.play();
    // Output:
    // Playing the instrument!
    // Strumming the guitar!
  }
}
```

Inheritance - Override

- When a child overrides a parent method, the method **can be made more public**
  - One exception: `private` methods
  - This is because the subclass won't even be aware these methods exist
- An overridden method **cannot be changed to a more restrictive permission**
  - If a superclass method is `public`, the overriding method cannot be `protected` or `private`
  - This violates Liskov Substitution Principle
- but a subclass can contain new private fields that were not present in the parent class

```java
class StringInstrument {
  protected String type = "String Instrument";

  public void play() {
    System.out.println("Playing the instrument!");
  }
}

class Guitar extends StringInstrument {
  @Override
  public void play() {
    super.play(); // Calls the base class's play method
    System.out.println("Strumming the guitar!");
  }

  public static void main(String[] args) {
    Guitar myGuitar = new Guitar();
    myGuitar.play();
    // Output:
    // Playing the instrument!
    // Strumming the guitar!
  }
}
```

Inheritance - super

- From a subclass, we can use `super` to access the parent class constructor
- Because subclasses are usually a more specific version of a superclass, it often makes sense to use the parent constructor
  - Dog class wouldn't make an Animal object using `super`, but can use parent constructor to help make the Dog object

```
class Animal {
    protected String name;
    // Parent class constructor
    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
```

```
class Dog extends Animal {
    // Child class constructor calling the parent constructor using
super
// What will happen if we just have public Dog() with no args?
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        super.makeSound(); // Calling the parent class method
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");
        System.out.println("Dog's name: " + myDog.name);
        myDog.makeSound(); // Output: Animal makes a sound followed
by Dog barks
    }
}
```

Inheritance - super

- A call to `super` must be the first action taken in a constructor definition
- If no `super` constructor is present (subclass just has a normal constructor) that's okay!
  - subclass will still access public/protected attributes and methods
  - but won't have any fields that were defined in the parent constructor

```
class BaseClass {
    protected int var1;
    protected int var2;

    // Base class constructor
    public BaseClass(int p1, int p2) {
        this.var1 = p1;
        this.var2 = p2;
    }
}
```

```
class DerivedClass extends BaseClass {
    private double instanceVariable;

    // Derived class constructor
    public DerivedClass(int p1, int p2, double p3) {
        super(p1, p2); // Call to the base class constructor
        this.instanceVariable = p3;
    }

    public void display() {
        System.out.println("var1: " + var1 + ", var2: " + var2
+ ", instanceVariable: " + instanceVariable);
    }

    public static void main(String[] args) {
        DerivedClass obj = new DerivedClass(5, 10, 3.14);
        obj.display(); // Output: var1: 5, var2: 10,
instanceVariable: 3.14
    }
}
```

Inheritance - Override

- `super` can be used in any method, not just the constructor
- Calling `super` from any overridden method will invoke the parent method

```
class StringInstrument {
  protected String type = "String Instrument";

  public void play() {
    System.out.println("Playing the instrument!");
  }
}

class Guitar extends StringInstrument {
  @Override
  public void play() {
    super.play(); // Calls the base class's play method
    System.out.println("Strumming the guitar!");
  }

  public static void main(String[] args) {
    Guitar myGuitar = new Guitar();
    myGuitar.play();
    // Output:
    // Playing the instrument!
    // Strumming the guitar!
  }
}
```

Multiple Constructors

- Classes can have more than one constructor
- No argument constructor
  - default is provided by Java, but we should define it ourselves
  - used when we don’t have any information about the object yet
  - you can set default values here

```java
class Vehicle {
    private String brand;
    private int year;

    public Vehicle() {
        this.brand = "Unknown";
        this.year = 0;
    }

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public void displayInfo() {
        System.out.println(brand + " " + year);
    }

    public static void main(String[] args) {
        Vehicle v1 = new Vehicle();
        Vehicle v2 = new Vehicle("Toyota", 2022);
        v1.displayInfo(); // Output: Unknown 0
        v2.displayInfo(); // Output: Toyota 2022
    }
}
```

Multiple Constructors

```
class Book {
    private String title;
    private String author;
    private int year;

    // Constructor with one argument
    public Book(String title) {
        this.title = title;
        this.author = "Unknown";
        this.year = -1; // Indicates year is unknown
    }

    // Constructor with two arguments
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.year = -1; // Indicates year is unknown
    }

    // Constructor with three arguments
    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public void printDetails() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Year: " + (year == -1 ? "Unknown" :
year));
    }

    public static void main(String[] args) {
        // Using different constructors
        Book book1 = new Book("The Great Gatsby");
        Book book2 = new Book("1984", "George Orwell");
        Book book3 = new Book("To Kill a Mockingbird", "Harper
Lee", 1960);

        book1.printDetails(); // Output: Title: The Great Gatsby,
Author: Unknown, Year: Unknown
        book2.printDetails(); // Output: Title: 1984, Author:
George Orwell, Year: Unknown
        book3.printDetails(); // Output: Title: To Kill a
Mockingbird, Author: Harper Lee, Year: 1960
    }
}
```

Inheritance - Overloading

- Overloading (not to be confused with Overriding) is same method name, different parameters
  - different number of parameters
  - or different types
  - or both
- Based on what is being passed in, the correct method is chosen at compile time
- Overloading improves code readability and provides flexibility in method usage
- Constructors can also be overloaded, as demonstrated in the previous slide

```
class Calculator {
    // Overloaded method for adding two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method for adding two doubles
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));
        // Output: 15 (int version)
        System.out.println(calc.add(5.5, 2.3));
        // Output: 7.8 (double version)
    }
}
```

Inheritance - Constructors

- When `super` is not used in the subclass constructor, Java will automatically call the no-argument constructor of the Parent
- Child classes should be considered extensions of the Parent class
  - Everything that needs to be initialized for the Parent class should also be initialized for the Child class

```
class Parent {
    protected int value;

    public Parent() {
        this.value = 42; // Initializes value
    }
}

class Child extends Parent {
    // No explicit super() call, but the no-argument
constructor of Parent is automatically called
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        System.out.println("Value: " + child.value);
       // Output: Value: 42
    }
}
```

Inheritance - Constructors

- When `super` is not used in the subclass constructor, Java will automatically call the no-argument constructor of the Parent
- Recall: Java will automatically create a no-argument constructor ONLY when no other constructors are explicitly defined
- Solutions:
  - Preferred: Define a no argument constructor in the Parent class
  - If appropriate: Call the existing constructor in the Parent class from the Child class with `super`

```
class Parent {
    protected int value;

    public Parent(int value) {
        this.value = value;
    }
}

class Child extends Parent {
    // There is no call to super(int) here
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child(); // Compilation error
    }
}
```

Inheritance

```
// Base class
class Employee {
    protected String name;
    protected double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public void work() {
        System.out.println(name + " is working.");
    }

    public void displayDetails() {
        System.out.println("Employee: " + name + ",
Salary: $" + salary);
    }
}
```

```
// Derived class
class Manager extends Employee {
    private int teamSize;

    public Manager(String name, double salary, int
teamSize) {
        super(name, salary);   //superclass constructor
        this.teamSize = teamSize;
    }

    // Manager specific method
    public void manage() {
        System.out.println(name + " manages a team of "  +
teamSize + " employees.");
    }

    // Overriding the displayDetails method
    @Override
    public void displayDetails() {
        super.displayDetails(); // superclass method
        System.out.println("Team Size: "  + teamSize);
    }
}
```

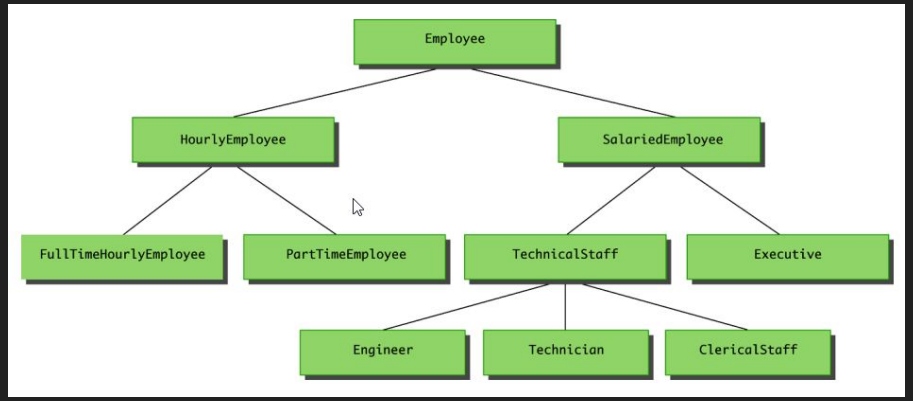
Inheritance - Methods

- Subclasses can access public and protected methods/properties, regardless of whether the subclass is in the same package as the superclass
- Private methods/properties are not accessible by subclasses, only by the superclass itself
- One small caveat:
  - Private methods can be called indirectly if a public method happens to invoke a private method
  - This isn’t problematic - private methods are often helping methods

```
class BaseClass {
    public void publicMethod() {
        System.out.println("Public method called.");
        privateHelperMethod(); // Indirectly calling
the private method
    }

    private void privateHelperMethod() {
        System.out.println("Private helper method
called.");
    }
}

class DerivedClass extends BaseClass {
    // No access to privateHelperMethod directly
}

public class Main {
    public static void main(String[] args) {
        DerivedClass obj = new DerivedClass();
        obj.publicMethod(); // Calls publicMethod,
which in turn calls privateHelperMethod
    }
}
```

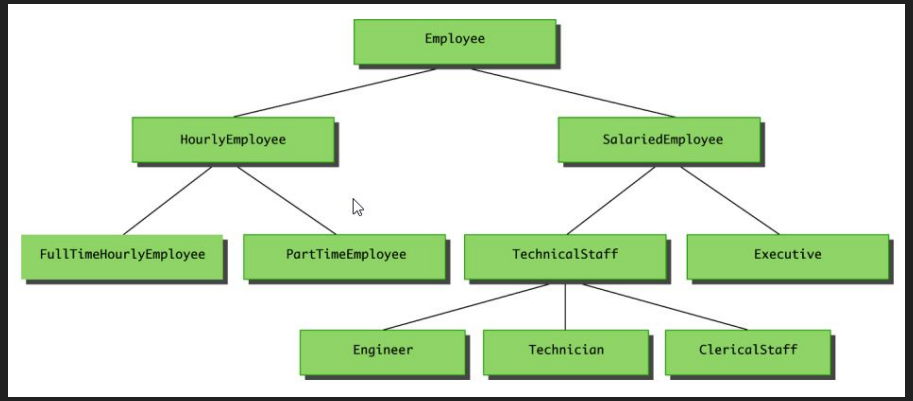
Inheritance

- A derived class demonstrates an “is a” relationship between it and its base class
  - Forming an “is a” relationship is one way to make a more complex class out of a simpler class
  - For example, HourlyEmployee “is an” Employee
  - HourlyEmployee is a more complex class compared to more general Employee class



Inheritance

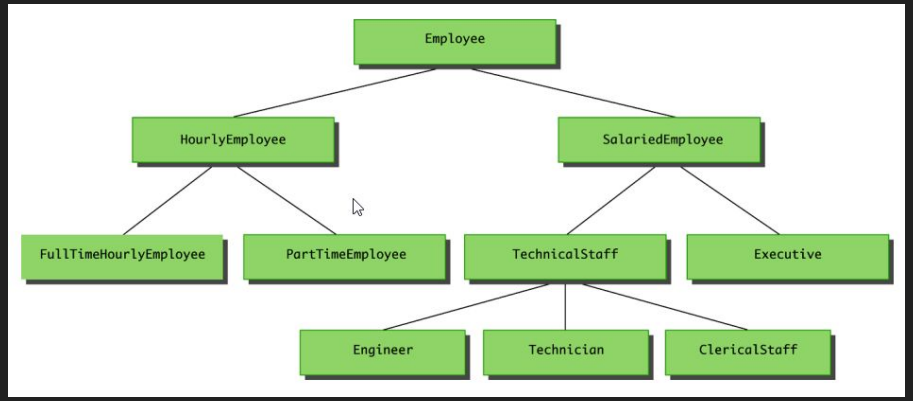
- Another way to make a more complex class is through “has a”
  - This type of relationship is called composition
  - Occurs when a class contains an instance variable of a class Type
- `Employee` class contains an instance variable hireDate of class `Date`, therefore, an `Employee` “has a” `Date`



Inheritance

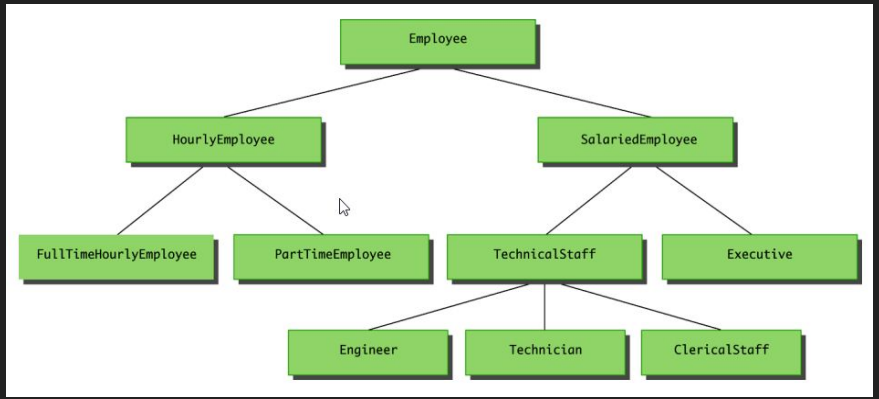
- You cannot use multiple `super`
  - Repeating `super` will not invoke a method from a further ancestor class
  - super can only be used to invoke methods from a direct parent
- Example:
  - An HourlyEmployee object cannot call

```
super.super.toString()
```



Inheritance - Types

- An Object of a derived class has the type of the derived class, but also the type of the base class
- More generally, an object of derived class has the type of all of its ancestor classes
    - But not vice versa




```
public class Main {
    public static void main(String[] args) {
        PartTimeEmployee partTimeEmp = new
PartTimeEmployee("John Doe", 20.0, 25);

        // Casting PartTimeEmployee object to
        HourlyEmployee
        HourlyEmployee hourlyEmp = partTimeEmp;

        // Casting PartTimeEmployee object to Employee
        Employee emp = partTimeEmp;

        partTimeEmp.displayInfo(); // PartTimeEmployee
        hourlyEmp.displayInfo();   // HourlyEmployee
        emp.displayInfo();         // Employee
    }
}
```

Java - Casting

- You can cast objects into different Classes
  - converting one data type into another
  - If casting to an ancestor class, no explicit casting is needed
  - Because Employee is a superclass of everything else, the examples here are implicit casting
  - no parentheses are needed
- Every Class in Java can therefore be implicitly cast as Object

```
public class Main {
    public static void main(String[] args) {
        // Create a PartTimeEmployee object
        PartTimeEmployee partTimeEmp = new PartTimeEmployee();

        // Cast PartTimeEmployee to HourlyEmployee
        HourlyEmployee hourlyEmp = partTimeEmp;

        // Cast HourlyEmployee to Employee
        Employee emp = partTimeEmp;

        // Use getClass() to verify the class type
        System.out.println("partTimeEmp is of type: " +
partTimeEmp.getClass());
        // partTimeEmp is of type: class PartTimeEmployee

        System.out.println("hourlyEmp is of type: " +
hourlyEmp.getClass());
        // hourlyEmp is of type: class PartTimeEmployee

        System.out.println("emp is of type: " +
emp.getClass());
        // emp is of type: class PartTimeEmployee
    }
}
```

Java - Casting

- What does this mean?

```
Employee emp = new HourlyEmployee();
```

- The object itself is a runtime instance of HourlyEmployee. In memory, the object has all the properties and methods of HourlyEmployee object.
- The *reference* is `Employee` class
- Why does this matter?
  - Polymorphism (more on this later)
  - Method accessibility

```
public class Main {
    public static void main(String[] args) {
        // Create a PartTimeEmployee object
        PartTimeEmployee partTimeEmp = new PartTimeEmployee();

        // Cast PartTimeEmployee to HourlyEmployee
        HourlyEmployee hourlyEmp = partTimeEmp;

        // Cast HourlyEmployee to Employee
        Employee emp = partTimeEmp;

        // Use getClass() to verify the class type
        System.out.println("partTimeEmp is of type: " +
partTimeEmp.getClass());
        // partTimeEmp is of type: class PartTimeEmployee

        System.out.println("hourlyEmp is of type: " +
hourlyEmp.getClass());
        // hourlyEmp is of type: class PartTimeEmployee

        System.out.println("emp is of type: " +
emp.getClass());
        // emp is of type: class PartTimeEmployee
    }
}
```

Java - Casting

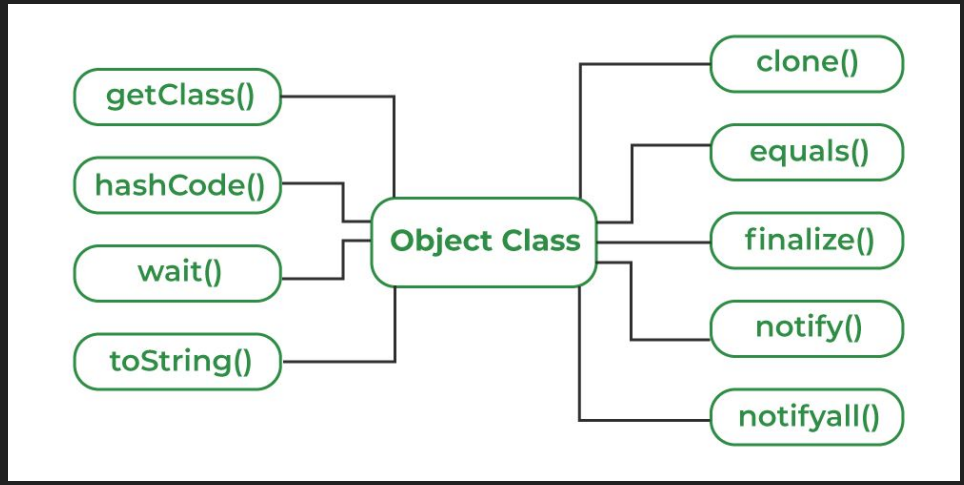
- This example is explicit casting
- Parentheses are needed to cast emp into an HourlyEmployee otherwise it won't compile (`ClassCastException`)
- When does this work?
  - If the object is an instance of the target class (or one of its subclasses) the cast will succeed
  - If casting to HourlyEmployee, the object must either be HourlyEmployee or a subclass of it
- This is also possible without assigning the cast to a new variable

```
((HourlyEmployee) emp).printHourlyRate();
// Output: Hourly rate is $20/hour
```

```
public class Main {
    public static void main(String[] args) {
        // Create an HourlyEmployee object
        // but assign it to an Employee reference
        Employee emp = new HourlyEmployee();

        // Attempt to call printHourlyRate directly on
Employee reference (won't compile)
        // emp.printHourlyRate(); // Error: Cannot
resolve method 'printHourlyRate' in 'Employee'

        // Explicit casting to access
HourlyEmployee-specific method
        HourlyEmployee hourlyEmp = (HourlyEmployee) emp;
        hourlyEmp.printHourlyRate(); // Output: Hourly
rate is $20/hour

        System.out.println("hourlyEmp is of type: " +
hourlyEmp.getClass().getSimpleName());
        // Expected Output: hourlyEmp is of type:
HourlyEmployee
    }
}
```

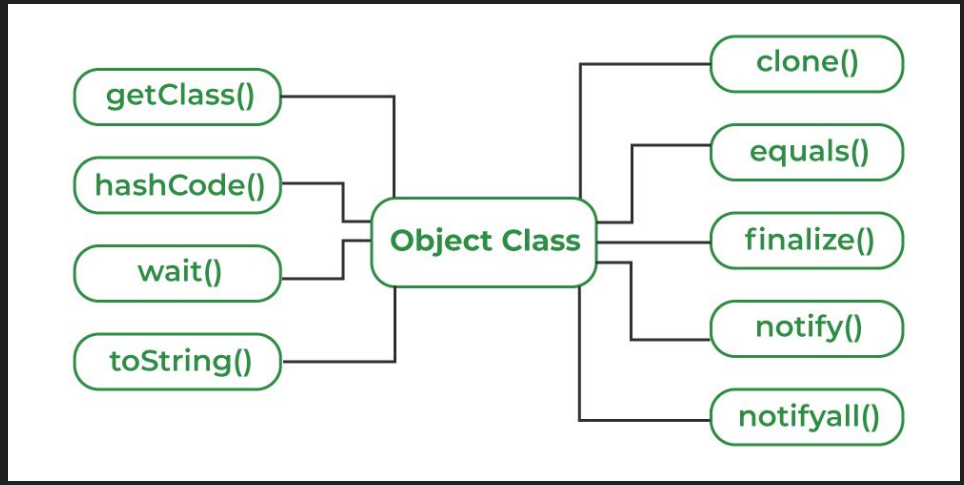
Java - Object

- In Java, every class is a descendant of the class `Object`
  - Every class has `Object` as its ancestor
  - Every object of every class is of type `Object`, as well as being the type of its own class
- The class `Object` is in the package java.lang which is always imported automatically
- What does this mean?
  - A parameter of type Object can be replaced by an object of any class whatsoever
  - Many library methods accept an argument of type Object so that they can be used with any class



Java - Object

- The class Object has some methods every Java class inherits
  - such as equals() and toString()
- However, these methods should usually be overridden with definitions more appropriate for your specific class
- equals() by default checks if two objects are the same object in memory/reference
  - but for a class like `Vehicle`, you might choose to override this and say equal if model/brand/color matches
  - logical equality vs computer equality
- toString() by default returns the class name and a hash code



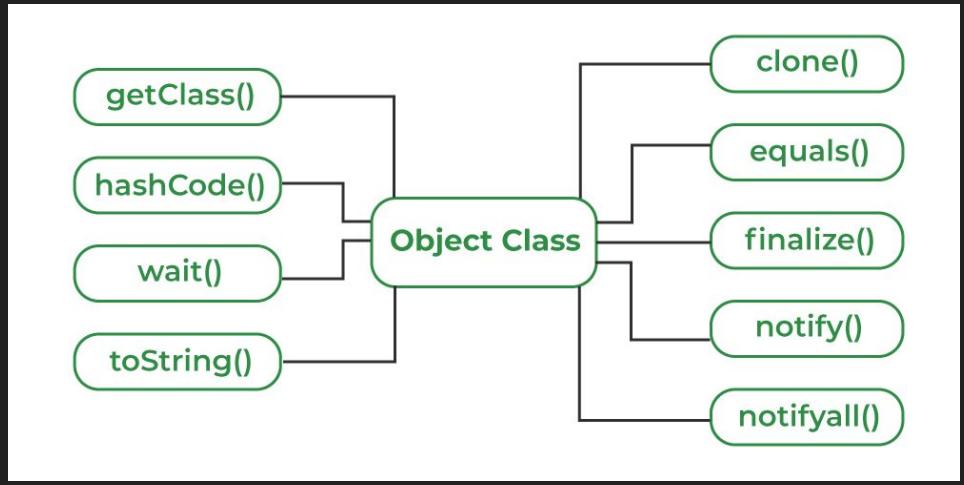
Java - Object

- public final `getClass()`
  - Returns a representation of the class that was used with `new` to create the object
  - The result can be compared with normal equality operator to determine whether two objects are actually the same class

```
(object1.getClass() == object2.getClass())
```

```
System.out.println("partTimeEmp is of type: " +
partTimeEmp.getClass());
        // partTimeEmp is of type: class
PartTimeEmployee
```

```
System.out.println("hourlyEmp is of type: " +
hourlyEmp.getClass());
        // hourlyEmp is of type: class
PartTimeEmployee
```



Java - Object

```
class Animal {
    private String name;
    public Animal(String name) { this.name = name; }
    @Override
    public String toString() { return "Animal: " +
name; }
}

class Vehicle {
    private String brand;
    public Vehicle(String brand) { this.brand = brand;
}
    @Override
    public String toString() { return "Vehicle: " +
brand; }
}
```

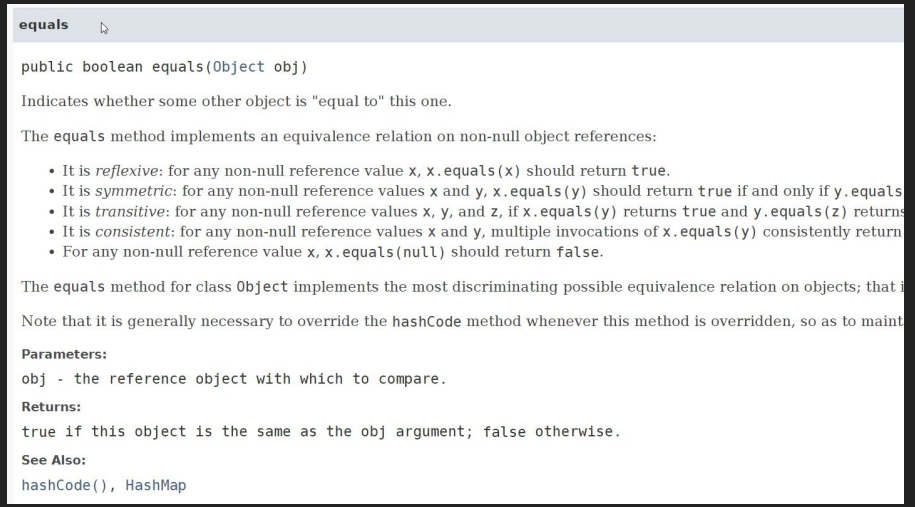
```
public class Main {
    public static void printObjectDetails(Object
obj) {
        System.out.println(obj); // Calls
toString() method
    }

    public static void main(String[] args) {
        printObjectDetails(new Animal("Dog"));
        // Output: Animal: Dog
        printObjectDetails(new Vehicle("Toyota"));
        // Output: Vehicle: Toyota
    }
}
```

Inheritance - Overriding

- How should the `equals` method behave?

```
public boolean equals(Employee otherEmployee)
{ . . . }
```

```
public boolean equals(Object otherObject)
{ . . . }
```



equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals`
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that i

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maint

Parameters:

`obj - the reference object with which to compare.`

Returns:

`true if this object is the same as the obj argument; false otherwise.`

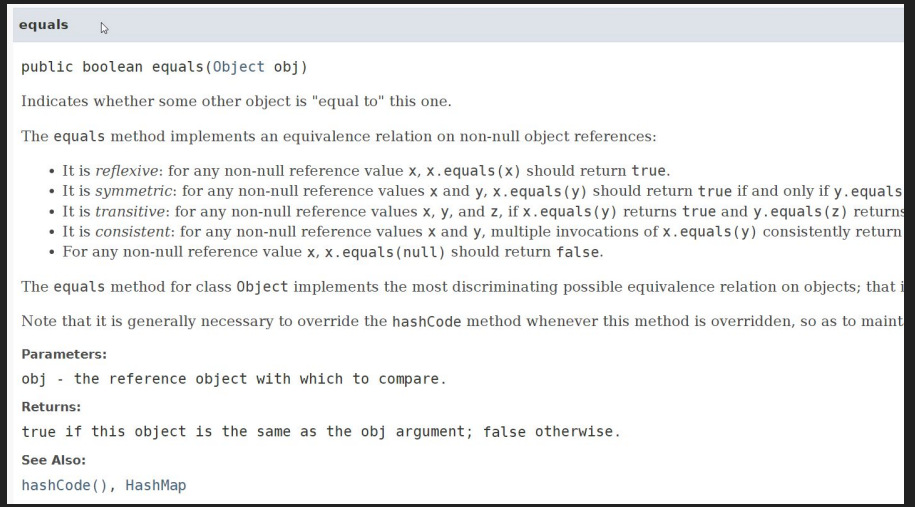
See Also:

`hashCode(), HashMap`

Inheritance - Overriding

- The first is overloaded. It has the same name as `equals` from the `Object` class but a different parameter (`Employee` instead of `Object`)
  - When using this, you can only compare `Employee` objects to this object
- The second option is overriding the equals provided by the superclass `Object`
  - More versatile because it will work with any Object in Java
  - Within the equals method, you can cast the Object into an Employee class for the actual logical comparison
  - Is `@Override` appropriate here?

```
public boolean equals(Employee otherEmployee)
{ . . . }

public boolean equals(Object otherObject)
{ . . . }
```



equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals`
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that i

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maint

Parameters:

`obj - the reference object with which to compare.`

Returns:

`true if this object is the same as the obj argument; false otherwise.`

See Also:

`hashCode(), HashMap`

Inheritance - Overriding

- What should an equals() method do?
  - Make sure the other Object isn't null
  - Check if the other Object is an Employee object
    - if so, cast into Employee reference
    - only then can you to access properties like `.name` for the comparison logic
    - Object class doesn't know about those fields
  - Then compare all the properties
- Many other data structures, like Hashset or ArrayList, have methods that depend on equals
  - Overriding equals ensures that these methods work as intended

```
class Employee {
    private String name;
    private String hireDate;

    public Employee(String name, String hireDate) {
        this.name = name;
        this.hireDate = hireDate;
    }

    // Properly overridden equals method
    @Override
    public boolean equals(Object otherObject) {
        if (otherObject == null) {
            return false;
        } else if (getClass() != otherObject.getClass()) {
            return false;
        } else {
            Employee otherEmployee = (Employee) otherObject;
            return (name.equals(otherEmployee.name) &&
                    hireDate.equals(otherEmployee.hireDate));
        }
    }
}
```

Method Overriding vs Method Overloading

- Method Overriding:
  - Redefining a method in a subclass that already exists in the superclass with the same signature
  - Purpose: Allows a subclass to provide a specific implementation of a method already defined
  - Inheritance: Only possible with inheritance
  - Annotation: @Override is not strictly necessary but highly recommended
- Method Overloading:
  - Defining multiple methods with the same name but different parameter lists (different number or types of parameters) in the same class/subclass
  - Purpose: Flexibility by allowing the same method to handle different inputs
  - Inheritance: No inheritance needed
  - Annotation: Nothing special, just define the method multiple times with different parameters

HW 3:

- Create a class named Shape:
  - Define a method public void draw() that prints "Drawing a shape".
  - Define a method public double area() that returns 0.0.
- Create a subclass named Circle that extends Shape:
  - Add a private attribute radius of type double.
  - Override the draw() method to print "Drawing a circle".
  - Override the area() method to calculate and return the area of the circle (π * radius * radius).
  - Create an equals() function that works on all Objects
- Create a subclass named Rectangle that extends Shape:
  - Add private attributes length and width of type double.
  - Override the draw() method to print "Drawing a rectangle".
  - Override the area() method to calculate and return the area of the rectangle (length * width).
  - Create an equals() function that works on all Objects
- In the main method:
  - Create instances of Circle and Rectangle.
  - Call the draw() and area() methods on each instance.
  - Output the results to verify that method overriding is functioning correctly.

HW 3:

- Create a class named Calculator:
  - Define a method public int add(int a, int b) that returns the sum of two integers.
  - Overload the add method with a version that takes three integers (public int add(int a, int b, int c)) and returns the sum of three integers.
  - Overload the add method with a version that takes two doubles (public double add(double a, double b)) and returns the sum of two double values.
- Create a subclass named ScientificCalculator that extends Calculator:
  - Override the add(int a, int b) method to print "Using ScientificCalculator to add two integers" before returning the sum.
  - Add a new method public double power(double base, double exponent) that calculates and returns the value of base raised to the power of exponent.
- In the main method:
  - Create an instance of ScientificCalculator.
  - Call each version of the add method using different argument types and numbers.
  - Call the power() method to demonstrate the additional functionality of ScientificCalculator.
  - Output the results to verify that method overloading and overriding are functioning correctly.