Inheritance in Java

Inheritance Overview

- Pillar of OOP - another layer of organization
- Defines relationships between Classes
- Contributes to code reuse.
  - DRY - don't repeat yourself

Inheritance Example

```java
public abstract class GameOfThronesHouse {

    private final String name;

    private final GameOfThronesCharacter[] characters;

    private final Banner banner;

    protected GameOfThronesHouse(String name, GameOfThronesCharacter[] characters, Banner banner) {
        this.name = name;
        this.characters = characters;
        this.banner = banner;
    }

    public String getName() {
        return name;
    }

    public String getFormalName() {
        return String.format("House %s", getName());
    }

    public GameOfThronesCharacter[] getCharacters() {
        return characters;
    }

    public Banner getBanner() {
        return banner;
    }

}
```

```java
public abstract class GameOfThronesCharacter {

    private final String name;

    private final GameOfThronesHouse house;

    public GameOfThronesCharacter(String name, GameOfThronesHouse house) {
        this.name = name;
        this.house = house;
    }

    public String getName() {
        return name;
    }

    public String getFormalName() {
        return String.format("%s %s", getName(), getHouse().getName());
    }

    public GameOfThronesHouse getHouse() {
        return house;
    }

}
```

Game of Thrones - Starks

```
1   public class Stark extends GameOfThronesCharacter {
2
3       public Stark(String name) {
4           super(name, new StarkHouse());
5       }
6
7   }
```

```
1   public class StarkHouse extends GameOfThronesHouse {
2
3       public StarkHouse() {
4           super("Stark",
5                   new Stark[] {
6                       new Stark("Bran"),
7                       new Stark("Sansa"),
8                       new Stark("Arya"),
9                       new Stark("Robb")
10                      // TODO - add others
11                  },
12              new Banner("Gray"));
13      }
14  }
```

Game of Thrones - Lannisters

```java
public class Lannister extends GameOfThronesCharacter {

    public Lannister(String name) {
        super(name, new LannisterHouse());
    }

}
```

```java
public class LannisterHouse extends GameOfThronesHouse {

    public LannisterHouse() {
        super("Lannister",
                Lannister[] {
                        new Lannister("Tyrion"),
                        new Lannister("Jaime"),
                        new Lannister("Cersei"),
                        new Lannister("Tywin")
                        // TODO - add others
                },
                new Banner("Red")
        );
    }
}
```

Inheritance Hierarchy

- Classes can extend from others forming a hierarchy.

```
1   public abstract class GameOfThronesHouse {
2   ...
3   }
4   public abstract class GameOfThronesNorthernHouse extends GameOfThronesHouse {
5   ...
6   }
7   public class StarkHouse extends GameOfThronesNorthernHouse {
8   ...
9   }
```

No Multiple Inheritance

- Classes extend from one and only one class
    - Cannot extend from two classes
    - All classes, even if not explicitly specified, extend from at least one class. If not specified the class is Object
- Problems with multiple inheritance
    - Biggest challenge is "the diamond problem"
    - If class C extends from both class A and class B and class A and class B implement a method called foo and C doesn't override it, when invoked at runtime which method should be called, A's or B's?
    - Java avoids multiple inheritance altogether.

Polymorphism

- Many distinct types referenced by their shared supertype
  - Take an example of an Animal class with subclasses Dog and Cat (i.e., `Dog extends Animal`). Then both Dog and Cat can be identified as Animal.

```
1  public class Animal {
2      public String makeNoise() {
3          return "";
4      }
5  }

1  public class Dog extends Animal {
2      @Override public String makeNoise() {
3          return "woof";
4      }
5  }

1  public class Cat extends Animal {
2      @Override public String makeNoise() {
3          return "meow";
4      }
5  }
```

```
1  public class AnimalListener {
2
3      public static void main(String[] args) {
4          Animal[] animals = new Animal[] { new Cat(), new Dog() };
5          AnimalListener listener = new AnimalListener();
6          listener.listen(animals);
7      }
8
9      public void listen(Animal[] animals) {
10         for (Animal animal : animals) {
11             System.out.printf("%s%n", animal.makeNoise());
12         }
13     }
14
15 }
```

Polymorphism Caution

- Dog is an Animal but not all Animals are Dogs
- Seems simple enough but can lead to issues with array objects.
  - Java has covariant arrays which lead to issues with polymorphic types.
  - If Java had invariant arrays then there wouldn't be an issue but the language would be less expressive.
  - Invariant arrays mean: given type X and type Y which extends from X (i.e., Y is an instance of X) then the array type, Y[] is NOT an instance of X[].
  - Covariant arrays means: given type X and type Y which extends from X (i.e., Y is an instance of X) then the array type, Y[] is an instance of X[].

Polymorphism Caution (cont)

- So what could go wrong?

```
1   Dog[] dogs = new Dog[] { new Dog(), new Dog() };
2   animals = dogs;
3   animals[0] = new Cat();
4   Dog dog = dogs[0]; // Rut roh...
```

- So what happens? Compile error?

The `final` keyword on classes/methods

- Marking classes as final makes them immutable; i.e., they cannot be subclassed
- Marking methods as final makes them immutable; i.e., they cannot be overridden
- Although making class fields immutable is desired, marking classes and methods final is not a common practice. Some do this for "performance" but it makes unit testing hard and prevents any future extensibility.

```java
1  public final class FinalClass {
2
3      // Cannot be extended!
4
5  }
6
7  public class FinalMethod {
8
9      public final String cannotBeOverridden() {
10         return "Will always return this String!";
11     }
12
13 }
```

Casts / `instanceof`

- **Cast converts from one type to another**
- **Instanceof checks if an object is an instance of a Class**
- Use sparingly. Most often found in reflection / tooling code
  - Casts were often present prior to Java 5 as there were no generics

```java
1   public class TypeCheck {
2
3       public boolean isString(Object value) {
4           return (value instanceof String);
5       }
6
7       public String coerce(Object value) throws ClassCastException {
8           return (String) value;
9       }
10
11  }
```

Abstract Classes

- Marking a class as `abstract` allows you to define functionality that can be leveraged by subclasses without allowing code to directly instantiate the type.
- When designing think of these as logical groupings but which cannot themselves exist (or in practice there'd always be a more specific type of these).
  - Animal and Employee are great examples. You don't *just* have an Animal you always have something that is an Animal but is something more specific like a Dog.
- When creating abstract classes need to keep encapsulation in mind. What fields/methods do you want to expose to everyone, just subclasses or no one.
  - If everyone (and method as fields should rarely if ever be public) mark `public`
  - If just subclasses mark `protected`
  - If just used by abstract class mark `private`

Abstract Class Example

```java
public abstract class Employee {
    private final String name;

    private final double salary;

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

    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }
}
```

Concrete Class Example

```java
public class Programmer extends Employee {

    private final String languagePreference;

    public Programmer(String name, double salary, String languagePreference) {
        super(name, salary);
        this.languagePreference = languagePreference;
    }

    public String getLanguagePreference() {
        return languagePreference;
    }

    @Override public String toString() {
        return String.format("Name - %s %nSalary - %.2f %nLanguage - %s",
                getName(), getSalary(), getLanguagePreference());
    }
}
```

Abstract methods

- Abstract classes can also define methods to be implemented.
- These methods it can invoke in other methods.
- You know a functionality should exist at the abstract level you just don't know what it is concretely; delegate this decision to the concrete class.

```java
public abstract class Animal {

    public abstract String makeNoise();

    public abstract int getNumberOfLimbs();

    public String describe() {
        return String.format("%s! I have %d limbs.", makeNoise(), getNumberOfLimbs());
    }

}
```

The Object Class

- Every class (not primitives) have the Object class as the root of their inheritance hierarchy.
- A class either extends from another or from Object
  - Can only explicitly extend from non-Object classes (by marking 'extends X')
  - Not extending from another class implicitly means extend from Object class

Object Class Methods

- getClass()
  - returns the Class class instance
- hashCode()
  - a hash code value for this object. More on this later.
- equals(Object obj)
  - returns true if this object equals `obj`. More on this later.
- clone()
  - returns a clone of this object. Prefer constructor methods. Rarely used in newer Java libraries.
- toString()
  - return a String representation of this object.

Object Class Methods (cont)

- notify()
  - wakes up a thread listening for this object's monitor. We'll discuss this in the Concurrency lecture. In general, better ways of doing this with concurrency libraries introduced in Java 5.
- notifyAll()
  - just like `notify` but wakes all threads listening for this object's monitor.
- wait(long) & wait(long, int) & wait()
  - waits the current thread until another calls one of the notify methods. Overloaded methods with varying timeout.
- finalize()
  - * Called when no more references exist *
  - Never override this. Never rely upon it being invoked.

Object Class - 3 methods you'll actually override

- toString()
  - Mostly used for logging.
  - Make this terse and descriptive. At least print things which will help you debug issues and identify the issue.
- equals(Object)
  - Used to identify instantiated objects with like properties
    - Default implementation from Object returns referential equality
  - In conjunction with hashCode, used often by the Collection interfaces
  - Strict rules about implementation.
- hashCode()
  - Used in conjunction with equals for hash-table implementations.
  - Strict rules about implementation.

equals Method Implementation Rules

- Reflexive
  - for any non-null x, `x.equals(x) == true`
- Symmetric
  - for any non-null x and y, `x.equals(y) == true` if and only if `y.equals(x) == true`
- Transitive
  - for any non-null x, y and z, if `x.equals(y) == true` and `y.equals(z) == true` then `x.equals(z) == true`
- Consistent
  - for any non-null x and y, x.equals(y) should, for repeated invocations, consistently return either true or false provided nothing under comparison has changed
- Null is false
  - for any non-null x, `x.equals(null) == false`

Counter Examples - equals

- What rule does this violate?
  - Is it reflexive?
  - Is it symmetric?
  - Is it transitive?
  - Is it consistent?
  - Is it "null == false"?

```java
public class WrongEquals {

    private final int variable;

    public WrongEquals(int variable) {
        this.variable = variable;
    }

    @Override public boolean equals(Object o) {
        return (variable < ((WrongEquals) o).variable);
    }

    @Override
    public int hashCode() {
        return variable;
    }
}
```

Counter Examples - equals

- What rule does this violate?
  - Is it reflexive?
  - Is it symmetric?
  - Is it transitive?
  - Is it consistent?
  - Is it "null == false"?

```java
public class WrongEquals {

    private final String variable;

    public WrongEquals(String variable) {
        this.variable = variable;
    }

    @Override public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o instanceof String) {
            return variable.equals((String) o);
        }
        if (o instanceof WrongEquals) {
            return variable.equals(((WrongEquals) o).variable);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (variable == null ? 0 : variable.hashCode());
    }
}
```

Counter Examples - equals

```java
public class WrongEquals {

    protected final String variable;

    public WrongEquals(String variable) {
        this.variable = variable;
    }

    @Override public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof WrongEquals)) {
            return false;
        }
        return variable.equals(((WrongEquals) o).variable);
    }

    @Override
    public int hashCode() {
        return (variable == null ? 0 : variable.hashCode());
    }

}
```

```java
public class WrongEqualsExt extends WrongEquals {

    private final String variable2;

    public WrongEqualsExt(String variable, String variable2) {
        super(variable);
        this.variable2 = variable2;
    }

    @Override public boolean equals(Object obj) {
        if (!(obj instanceof WrongEqualsExt)) {
            return super.equals(obj);
        }
        return (super.equals(obj)
            && variable2.equals(((WrongEqualsExt) obj).variable2));
    }

    @Override public int hashCode() {
        return super.hashCode();
    }

}
```

Counter Examples - equals (cont)

- What rule does this violate?
  - Is it reflexive?
  - Is it symmetric?
  - Is it transitive?
  - Is it consistent?
  - Is it "null == false"?

```java
public class WrongEquals {
    ...
    @Override public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof WrongEquals)) {
            return false;
        }
        return variable.equals(((WrongEquals) o).variable);
    }
}
public class WrongEqualsExt extends WrongEquals {
    ...
    @Override public boolean equals(Object obj) {
        if (!(obj instanceof WrongEqualsExt)) {
            return super.equals(obj);
        }
        return (super.equals(obj)
            && variable2.equals(((WrongEqualsExt) obj).variable2));
    }
}
```

Counter Examples - equals (cont)

```java
49  @Test
50  public void equals() {
51      String foo = "foo";
52      WrongEquals wrongEquals = new WrongEqualsExt(foo, "NOT EQUALS");
53      WrongEquals wrongEquals1 = new WrongEquals(foo);
54      WrongEquals wrongEquals2 = new WrongEqualsExt(foo, foo);
55
56      System.out.printf("%s%n", wrongEquals.equals(wrongEquals1));
57      System.out.printf("%s%n", wrongEquals1.equals(wrongEquals2));
58      System.out.printf("%s%n", wrongEquals.equals(wrongEquals2));
59  }
```

Counter Examples - equals

- What rule does this violate?
  - Is it reflexive?
  - Is it symmetric?
  - Is it transitive?
  - Is it consistent?
  - Is it "null == false"?

```java
1   public class WrongEquals {
2
3       private final AtomicInteger counter;
4
5       public WrongEquals(int start) {
6           this.counter = new AtomicInteger(start);
7       }
8
9       public int getCount() {
10          return counter.getAndDecrement();
11      }
12
13      @Override public boolean equals(Object o) {
14          if (this == o) {
15              return true;
16          }
17          if (o == null || getClass() != o.getClass()) {
18              return false;
19          }
20          WrongEquals that = (WrongEquals) o;
21          return counter.get() == that.getCount();
22      }
23
24      @Override public int hashCode() {
25          return counter.hashCode();
26      }
27  }
```

Counter Examples - equals

- What rule does this violate?
  - Is it reflexive?
  - Is it symmetric?
  - Is it transitive?
  - Is it consistent?
  - Is it "null == false"?

```java
public class WrongEquals {

    private final AtomicInteger counter;

    public WrongEquals(int start) {
        this.counter = new AtomicInteger(start);
    }

    public int getCount() {
        return counter.getAndDecrement();
    }

    @Override public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        WrongEquals that = (WrongEquals) o;
        return getCount() == that.getCount();
    }

    @Override public int hashCode() {
        return counter.hashCode();
    }
}
```

Implementing equals - instanceof or getClass?

- Always (*) use `getClass()` checks

```
17        if (o == null || getClass() != o.getClass()) {
18            return false;
19        }
```

* you can use `instanceof` if the class is `final` or you mark the equals implementation as `final`

hashCode Method Implementation

- Consistent
  - if invoked on the same object more than once in same JVM then it must return the same integer provided nothing under comparison has changed
- Equality
  - If x and y are equals (`x.equals(y) == true`) then their hash-code values must be the same (`x.hashCode() == y.hashCode() == true`)
  - This does not imply that unequal objects must have different hash-codes, in fact they often do not (called collisions). Collisions should be minimized for fast hash-table implementations.
    - I.e., if x and y are unequal (`x.equals(y) == false`) then their hash-code values may be the same (`x.hashCode() == y.hashCode() == true`)

Counter Examples - hashCode

- Why is this an incorrect hashCode implementation?

```java
1   public class WrongHashCode {
2
3       private final String id;
4
5       private final Number amount;
6
7       private final String name;
8
9       public WrongHashCode(String id, Number amount, String name) {
10          this.id = id;
11          this.amount = amount;
12          this.name = name;
13      }
14
15      @Override public boolean equals(Object o) {
16          if (this == o) {
17              return true;
18          }
19          if (o == null || getClass() != o.getClass()) {
20              return false;
21          }
22
23          WrongHashCode that = (WrongHashCode) o;
24
25          if (amount != null ? !amount.equals(that.amount) : that.amount != null) {
26              return false;
27          }
28          if (id != null ? !id.equals(that.id) : that.id != null) {
29              return false;
30          }
31          if (name != null ? !name.equals(that.name) : that.name != null) {
32              return false;
33          }
34          return true;
35      }
36
37      @Override public int hashCode() {
38          int result = id != null ? id.hashCode() : 0;
39          result = 31 * result + (amount != null ? amount.hashCode() : 0);
40          return result;
41      }
42  }
```

hashCode Gotcha!

- Although hashCode values may change for the same object, this is un-advisable and can lead to hard to diagnose bugs.
  - Another win for immutability! Immutable objects by definition will not have changing hashCode values as their variables are not changing.
- What trouble can arise if the hashCode value changes?

Auto boxing & unboxing

- Allows primitive values to be converted to and from their corresponding Object representations automatically by the compiler/JVM.
  - Auto-boxing -> convert primitive to corresponding Object
  - Auto-unboxing -> convert Object to corresponding primitive
- Introduced in Java 1.5 as a programming convenience.

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Autoboxing Gotcha! (cont)

```java
public class GotchaTwo {

    @Override public Integer hashCode() {
        return super.hashCode();
    }
}
```

Autoboxing Gotcha! (cont)

```java
public class GotchaThree {

    public static void main(String[] args) {

        int first;
        Integer second;
        Integer third;

        first = 127;
        second = 127;
        third = 127;

        System.out.printf("%s%n", (first == second));
        System.out.printf("%s%n", (second == third));

        first = 128;
        second = 128;
        third = 128;

        System.out.printf("%s%n", (first == second));
        System.out.printf("%s%n", (second == third));

    }

}
```

Variable Arguments - varargs

- Allows methods to be defined without knowing the exact number of arguments passed.
- Arguments must be of the same type and must occur as the last argument of the method.
- Introduced in Java 1.5 as a programming convenience.
  - Simply syntactic sugar around wrapping the methods into an array of the same type.
  - Super convenient when invoking methods though; use this whenever possible

Varargs - example

```java
public class Varargs {

    public static void main(String[] args) {
        Varargs varargs = new Varargs();
        varargs.print("foo", "and", "bar", "and", "more");
    }

    // arguments type is String[]
    public void print(String ... arguments) {
        for (String argument : arguments) {
            System.out.printf("%s%n", argument);
        }
    }

}
```

Enum Types

- Enumeration of all possible values of a type at compile time
- Use when all values are known upfront
- Useful as easy for programmers to reason about logic
  - I.e., can use in switch statements

```
1   public enum Day {
2
3       Sunday,
4       Monday,
5       Tuesday,
6       Wednesday,
7       Thursday,
8       Friday,
9       Saturday
10
11  }
```

Enum Types (cont)

- Enum types are a special type of Class
  - Can be extended and have instance fields and methods just like any other Object of a Class
- Enum value is simply an Object instance of the Enum class.

```java
public enum GasolineGrade {

    Premium(97),

    Plus(93),

    Regular(87),

    Diesel(20);

    private final Integer octane;

    private GasolineGrade(Integer octane) {
        this.octane = octane;
    }

    public Integer getOctane() {
        return octane;
    }
}
```

Read Chapter 6

All sections except 6.3 and 6.5 will be covered in next lecture

- You can skip sections 6.3 (default methods) 6.5 (Proxies)
    - 6.3 will be covered in lecture 12 (Functional Java)

Homework 4

https://github.com/NYU-CS9053/Fall-2019-II/homework/week4

Autoboxing Gotcha!

```java
public class GotchaOne {
    public static void main(String[] args) {
        GotchaOne gotchaOne = new GotchaOne();
        gotchaOne.print(gotchaOne.load());
    }
    public Integer load() {
        Integer value = null;
        // TODO - Load from DB
        return value;
    }
    public void print(int value) {
        System.out.printf("Value is %d%n", value);
    }
}
```