

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

```
1  public abstract class GameOfThronesHouse {
2
3      private final String name;
4
5      private final GameOfThronesCharacter[] characters;
6
7      private final Banner banner;
8
9      protected GameOfThronesHouse(String name, GameOfThronesCharacter[] characters, Banner banner) {
10         this.name = name;
11         this.characters = characters;
12         this.banner = banner;
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public String getFormalName() {
20         return String.format("House %s", getName());
21     }
22
23     public GameOfThronesCharacter[] getCharacters() {
24         return characters;
25     }
26
27     public Banner getBanner() {
28         return banner;
29     }
30
31 }
```

```
1 public abstract class GameOfThronesCharacter {
2
3     private final String name;
4
5     private final GameOfThronesHouse house;
6
7     public GameOfThronesCharacter(String name, GameOfThronesHouse house) {
8         this.name = name;
9         this.house = house;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public String getFormalName() {
17        return String.format("%s %s", getName(), getHouse().getName());
18    }
19
20    public GameOfThronesHouse getHouse() {
21        return house;
22    }
23
24 }
```

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

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

```
1 public class LannisterHouse extends GameOfThronesHouse {  
2  
3     public LannisterHouse() {  
4         super("Lannister",  
5             Lannister[] {  
6                 new Lannister("Tyrion"),  
7                 new Lannister("Jaime"),  
8                 new Lannister("Cersei"),  
9                 new Lannister("Tywin")  
10                // TODO - add others  
11            },  
12            new Banner("Red")  
13        );  
14    }  
15 }
```

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.

```
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

```
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

```
1 public abstract class Employee {  
2     private final String name;  
3  
4     private final double salary;  
5  
6     protected Employee(String name, double salary) {  
7         this.name = name;  
8         this.salary = salary;  
9     }  
10  
11     public String getName() {  
12         return name;  
13     }  
14  
15     public double getSalary() {  
16         return salary;  
17     }  
18 }
```

Concrete Class Example

```
1  public class Programmer extends Employee {
2
3      private final String languagePreference;
4
5      public Programmer(String name, double salary, String languagePreference) {
6          super(name, salary);
7          this.languagePreference = languagePreference;
8      }
9
10     public String getLanguagePreference() {
11         return languagePreference;
12     }
13
14     @Override public String toString() {
15         return String.format("Name - %s %nSalary - %.2f %nLanguage - %s",
16                               getName(), getSalary(), getLanguagePreference());
17     }
18 }
```


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.

```
1 public abstract class Animal {  
2  
3     public abstract String makeNoise();  
4  
5     public abstract int getNumberOfLimbs();  
6  
7     public String describe() {  
8         return String.format("%s! I have %d limbs.", makeNoise(), getNumberOfLimbs());  
9     }  
10  
11 }
```

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”?

```
1 public class WrongEquals {  
2  
3     private final int variable;  
4  
5     public WrongEquals(int variable) {  
6         this.variable = variable;  
7     }  
8  
9     @Override public boolean equals(Object o) {  
10         return (variable < ((WrongEquals) o).variable);  
11     }  
12  
13     @Override  
14     public int hashCode() {  
15         return variable;  
16     }  
17 }
```

Counter Examples - equals

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

```
1 public class WrongEquals {
2
3     private final String variable;
4
5     public WrongEquals(String variable) {
6         this.variable = variable;
7     }
8
9     @Override public boolean equals(Object o) {
10         if (this == o) {
11             return true;
12         }
13         if (o instanceof String) {
14             return variable.equals((String) o);
15         }
16         if (o instanceof WrongEquals) {
17             return variable.equals(((WrongEquals) o).variable);
18         }
19         return false;
20     }
21
22     @Override
23     public int hashCode() {
24         return (variable == null ? 0 : variable.hashCode());
25     }
26 }
```


Counter Examples - equals

```
1 public class WrongEquals {
2
3     protected final String variable;
4
5     public WrongEquals(String variable) {
6         this.variable = variable;
7     }
8
9     @Override public boolean equals(Object o) {
10         if (this == o) {
11             return true;
12         }
13         if (!(o instanceof WrongEquals)) {
14             return false;
15         }
16         return variable.equals(((WrongEquals) o).variable);
17     }
18
19     @Override
20     public int hashCode() {
21         return (variable == null ? 0 : variable.hashCode());
22     }
23
24 }
```

```
26 public class WrongEqualsExt extends WrongEquals {
27
28     private final String variable2;
29
30     public WrongEqualsExt(String variable, String variable2) {
31         super(variable);
32         this.variable2 = variable2;
33     }
34
35     @Override public boolean equals(Object obj) {
36         if (!(obj instanceof WrongEqualsExt)) {
37             return super.equals(obj);
38         }
39         return (super.equals(obj)
40             && variable2.equals(((WrongEqualsExt) obj).variable2));
41     }
42
43     @Override public int hashCode() {
44         return super.hashCode();
45     }
46
47 }
```

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”?

```
1 public class WrongEquals {  
2     ...  
3     @Override public boolean equals(Object o) {  
4         if (this == o) {  
5             return true;  
6         }  
7         if (!(o instanceof WrongEquals)) {  
8             return false;  
9         }  
10        return variable.equals(((WrongEquals) o).variable);  
11    }  
12 }  
13 public class WrongEqualsExt extends WrongEquals {  
14     ...  
15     @Override public boolean equals(Object obj) {  
16         if (!(obj instanceof WrongEqualsExt)) {  
17             return super.equals(obj);  
18         }  
19         return (super.equals(obj)  
20             && variable2.equals(((WrongEqualsExt) obj).variable2));  
21     }  
22 }
```

Counter Examples - equals (cont)

```
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”?

```
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”?

```
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 getCount() == that.getCount();
22    }
23
24    @Override public int hashCode() {
25        return counter.hashCode();
26    }
27 }
```

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?

```
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
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
@Override public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }

    WrongHashCode that = (WrongHashCode) o;

    if (amount != null ? !amount.equals(that.amount) : that.amount != null) {
        return false;
    }
    if (id != null ? !id.equals(that.id) : that.id != null) {
        return false;
    }
    if (name != null ? !name.equals(that.name) : that.name != null) {
        return false;
    }
    return true;
}

@Override public int hashCode() {
    int result = id != null ? id.hashCode() : 0;
    result = 31 * result + (amount != null ? amount.hashCode() : 0);
    return result;
}
```


hashCode Gotcha!

- Although hashCode values may change for the same object, this is unadvisable 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)

```
1 public class GotchaTwo {  
2  
3     @Override public Integer hashCode() {  
4         return super.hashCode();  
5     }  
6 }
```

Autoboxing Gotcha! (cont)

```
1 public class GotchaThree {  
2  
3     public static void main(String[] args) {  
4  
5         int first;  
6         Integer second;  
7         Integer third;  
8  
9         first = 127;  
10        second = 127;  
11        third = 127;  
12  
13        System.out.printf("%s%n", (first == second));  
14        System.out.printf("%s%n", (second == third));  
15  
16        first = 128;  
17        second = 128;  
18        third = 128;  
19  
20        System.out.printf("%s%n", (first == second));  
21        System.out.printf("%s%n", (second == third));  
22  
23    }  
24  
25 }
```

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

```
1  public class Varargs {  
2  
3      public static void main(String[] args) {  
4          Varargs varargs = new Varargs();  
5          varargs.print("foo", "and", "bar", "and", "more");  
6      }  
7  
8      // arguments type is String[]  
9      public void print(String ... arguments) {  
10         for (String argument : arguments) {  
11             System.out.printf("%s%n", argument);  
12         }  
13     }  
14  
15 }
```

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.

```
1 public enum GasolineGrade {
2
3     Premium(97),
4
5     Plus(93),
6
7     Regular(87),
8
9     Diesel(20);
10
11     private final Integer octane;
12
13     GasolineGrade(Integer octane) {
14         this.octane = octane;
15     }
16
17     public Integer getOctane() {
18         return octane;
19     }
20 }
```


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/Spring-2017/homework/week4>

Autoboxing Gotcha!

```
1  public class GotchaOne {
2      public static void main(String[] args) {
3          GotchaOne gotchaOne = new GotchaOne();
4          gotchaOne.print(gotchaOne.load());
5      }
6      public Integer load() {
7          Integer value = null;
8          // TODO - Load from DB
9          return value;
10     }
11     public void print(int value) {
12         System.out.printf("Value is %d%n", value);
13     }
14 }
```