Review: Class Design, Inheritance, and Polymorphism

CODING CONVENTIONS

Conventions- Why do they matter?

- Makes code more readable
- Makes code easier to maintain (by you or, more likely, by others)
- Provides a quick signifier to the reader of the type of information (e.g., variable, constant, class)
- Offers an indicator of the quality of your code
  - Oracle: If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
- More in-depth discussion: http://hilton.org.uk/presentations/naming-guidelines

Java Conventions

- Oracle: https://www.oracle.com/technetwork/java/javase/overview/codeconvtoc-136057.html
- Google: https://google.github.io/styleguide/javaguide.html
- GitHub: https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md

Variables

- lowerCamelCase
  - first letter lowercase, first letter of each internal word capitalized
- descriptive and meaningful
  - avoid one-character names except for temporary variables (like looping variables)

- This applies to parameters, too!

Constants

- ALL_CAPS_WITH_UNDERSCORES
- From Oracle: Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Methods

- lowerCamelCase
- descriptive
- often contains a verb

Classes and Interfaces

- UpperCamelCase
  - first letter and each internal word capitalized
- Simple and Descriptive
- Often a singular noun
- Use whole words
- Avoid acronyms and abbreviations (unless abbreviation is much more widely used, such as URL or HTML)

Others

- Oracle: It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others- you shouldn't assume that other programmers know precedence as well as you do.
- Use indentation and whitespace to improve readability

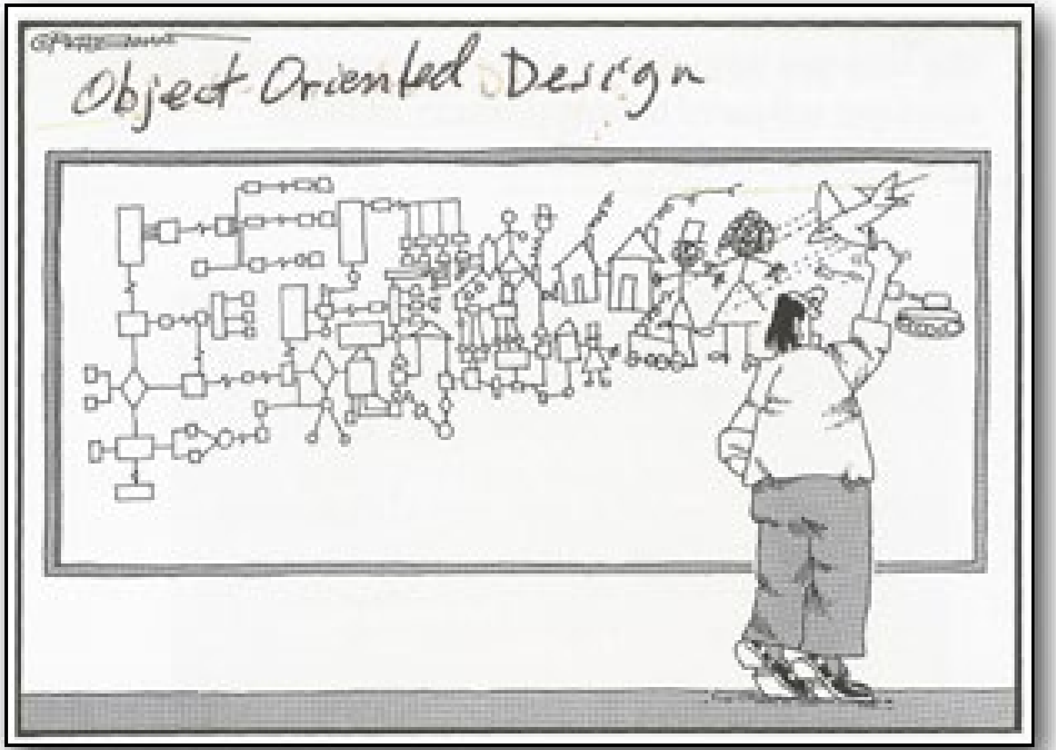
A Final Thought

“Coding conventions should be used as a guide, not as a sacred, immutable law of nature.

Those with less experience will initially benefit from following conventions, but as you gain experience, you should rely more on your own sense of taste, as opposed to blindly following the rules. The problem with rules is that there are many cases in which they should be broken.

In general, the question in your mind should always be one of **compassion for the reader**: *"Will this help or hinder a maintainer who is unfamiliar with my code?"*. That is the guiding principle.”

Source: http://www.javapractices.com/topic/TopicAction.do?Id=115



CLASS DESIGN

Writing Classes

- Classes define a blueprint of what all objects of that class will look like.
- Classes define:
  - Characteristics (state)
    - Instance data
  - Functionality (behaviors)
    - Methods
- An object is an *instance of* a class.

Classes Contain:

- Instance data
- Constructor(s)
- Getters and Setters
- toString and equals (overridden from Object)

- Class-specific methods

Instance Data

- Represents the characteristics of the object
- Each object has its **own copy** of instance data variables
- Make instance data variables private!

Practice

- Write a class to represent an employee.

Constructors

- Sets up the object
  - Initializes instance data variables using either parameters or default values (declared as constants)
- Often overloaded
  - Invoke overloaded constructors with the `this` keyword

Constructors

- The constructor is what is called when the object is created with the `new` keyword
  - `new` does three things:
    - allocates the necessary memory for the object
    - executes the constructor
    - returns the address of (reference to) the object so it can be stored in the variable

Constructors

- If you don’t define any constructor for a class, the compiler will define a default, no-argument constructor

Getters and Setters

- Provides access to the instance data
- Also called *accessors* and *mutators*
- Getter
  - Returns the current value of a variable

```
public TYPE getVARNAME() {
  return VARNAME;
}
```

- Setter
  - Updates the value of a variable
  - Can define parameters for valid values

```
public void setVARNAME(TYPE NEWVALUE) {
  VARNAME = NEWVALUE;
}
```

`toString` Method

- Returns a text representation of the object
- Overrides the method inherited from the `Object` class

```
public String toString() {
  return STRINGVAL;
}
```

The `equals` Method

- Returns true if the current object is *the same as* (or *logically equivalent to*) another object
- Overrides the method inherited from the `Object` class
  - The `Object` version tests whether two references are *aliases*. This is the same functionality as ==.

```
public boolean equals(Object obj) {
  return BOOLVAL;
}
```

The `equals` Contract

- Reflexive
  - For non-null x, x.equals(x) is true
- Symmetric
  - x.equals(y) is true if and only if y.equals(x) is true
- Transitive
  - If x.equals(y) is true and y.equals(z) is true, then x.equals(z) is true
- Consistent
  - If x and y haven’t changed, repeated calls to x.equals(y) return the same value
- For any non-null x, x.equals(null) is false

The equals Method Header

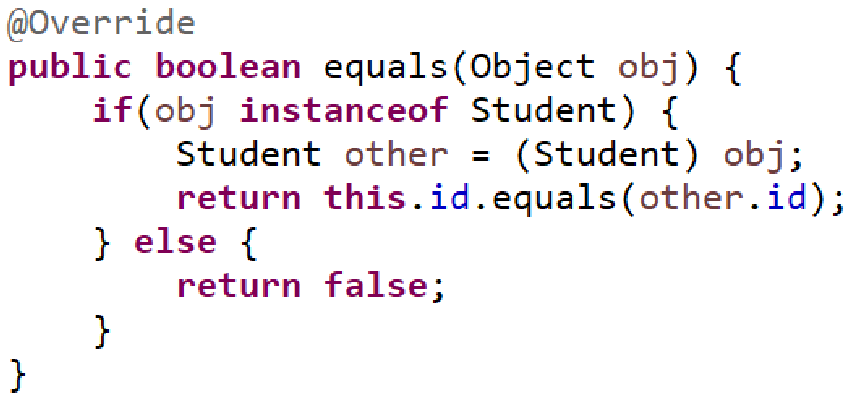
- The method is overriding the one from Object- so the parameter **must** by type Object.
- For an example of why, see this video.

Pattern Matching for instanceof

- This is new to Java 16.
- This feature allows us to streamline our equals methods.

instanceof

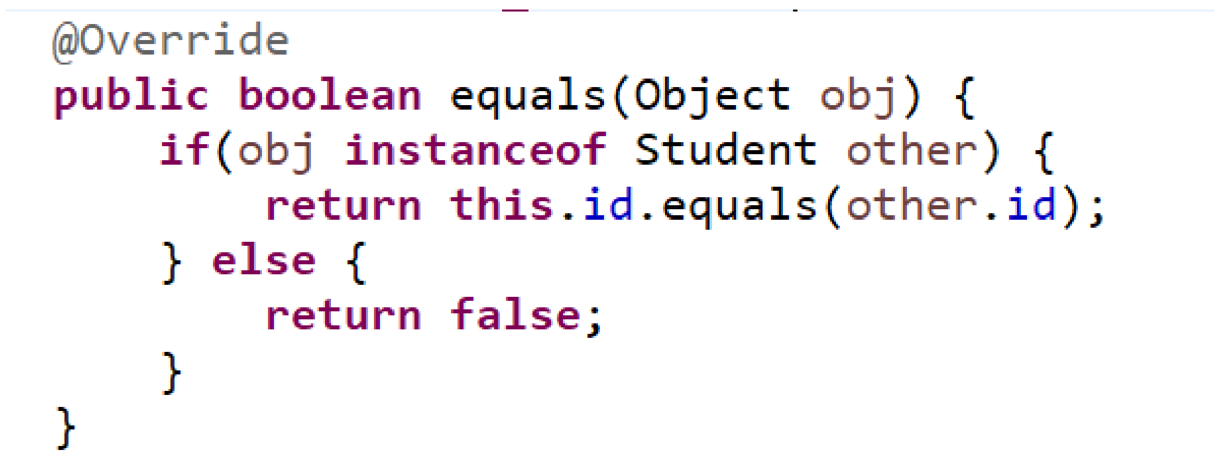
- Using instanceof is an important check to use with casting to avoid runtime ClassCastException errors.
- Example:



```java
@Override
public boolean equals(Object obj) {
    if(obj instanceof Student) {
        Student other = (Student) obj;
        return this.id.equals(other.id);
    } else {
        return false;
    }
}
```

Pattern Matching for instanceof

- We can now rewrite this as:



```java
@Override
public boolean equals(Object obj) {
    if(obj instanceof Student other) {
        return this.id.equals(other.id);
    } else {
        return false;
    }
}
```

Pattern Matching for instanceof

- This uses a *type test pattern* that specifies a type and a binding variable.
- If obj is an instance of the specified class, it is cast to that class and assigned to the binding variable.
- The binding variable is only in scope for that block of the conditional.

```
  if(obj instanceof SomeClass thing) {
      // can use thing here
      // the type of thing is SomeClass
} else {
      // cannot use something here- out of scope
}
```

Pattern Matching for instanceof

- We can even go further to simplify!

```
@Override
public boolean equals(Object obj) {
   if(obj instanceof Employee) {
      Employee otherEmployee = (Employee) obj;

      if(this.id==otherEmployee.id && this.name.equalsIgnoreCase(otherEmployee.name)) {
         return true;
      } else {
         return false;
      }
   } else {
      return false;
   }
}
```

Pattern Matching for instanceof

```
@Override
public boolean equals(Object obj) {
   return (obj instanceof Employee otherEmployee)
      && this.id == otherEmployee.id
      && this.name.equalsIgnoreCase(otherEmployee.name);
}
```

Encapsulation

- Data hiding
- Objects protect their instance data
- Provide methods to appropriately access data
- Changes to an object’s state (instance data) made only through methods
  - Clients cannot access instance data directly
- Supported through private instance data variables and public getters/setters (with validity checks as appropriate)

INHERITANCE

Inheritance

- *Inheritance* allows you to design a new class from an existing class
- Inheritance creates an *is-a* relationship.
  - The child *is a* more specific version of the parent.
- The child inherits characteristics of the parent
  - Methods
  - Data
- You tailor the child class by:
  - Adding new methods and variables
  - Modifying inherited methods

Practice

- Write child classes to represent a full time and part time employee.

Parent Constructors

- Constructors are *not* inherited, even though they have public visibility.
  - The child class does not inherit the constructor. It needs its own constructor.
  - Often, we still need to set up the “parent’s part” of the object but then we need to set up some additional things for the “child’s part” of the object.
- It’s good practice to call the parent constructor and then add any additional set up needed in the child class.

The `super` Reference

- Use `super` to invoke the parent constructor.
- Use `super` to reference other variables and methods in the parent class.
- Use `super` to reference public getters/setters of private variables.
- Use `super` to invoke the parent's version of a method you are overriding.

Overriding Methods

- A child class can *override* the definition of an inherited method and provide its own implementation.
  - The new method must have the same signature (name and parameters) as the parent’s method, but will have a different implementation.
- If a child object executes the method, the child’s version will be executed.
- If a parent object executes the method, the parent’s version will be executed.

Overriding Methods (cont.)

- A method in the parent class can be invoked explicitly with the `super` reference.
  - Typically, you invoke the parent's version with `super` and then add additional code needed for the child version.

- If a method is declared with the `final` modifier, it cannot be overridden.

@Override Annotation

- Annotations contain metadata about programs.
- You can use annotations to create new compiler checks.
- @Override will ensure that the method header is correct at compile time.
- Always better to find/fix at compile time rather than runtime!

Class Hierarchies

- Common features should be put up as high in the hierarchy as possible to increase the amount of reuse…
  - Write once, use often!
- An inherited variable or method is passed continually down the line
  - A child class inherits from **all** of its ancestor classes

The `Object` Class

- The Object class is at the top of the Java class hierarchy.
- All classes inherit from Object (either directly or indirectly).

The `Object` Class Methods

- There are a few useful methods in `Object` that are inherited by all classes:

```
public String toString()
public boolean equals(Object obj)
public int hashcode()
```

Designing for Inheritance

1. Inheritance should always represent an *is-a* relationship.
2. Find common characteristics of classes and push them as high in the class hierarchy as possible.
3. Override methods to tailor or change the functionality of a child.

Designing for Inheritance

4. Allow each class to manage its own data.
   - Use the `super` reference to invoke the parent's constructor to set up its data.
   - Use the `super` reference to invoke the parent's existing methods.
5. Even if there are no current uses for them, override general `Object` methods like `toString` and `equals`.
6. Use visibility modifiers carefully to provide needed access without violating encapsulation.

Restricting Inheritance

- The `final` modifier can be used to prevent inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any child classes
- If the `final` modifier is applied to a class, then that class cannot have any child classes

New to Java 17: Sealed Classes

- A *sealed* class restricts which other classes can extend it.
- A *sealed* interface restricts which classes can implement it.

- Example:

```
public sealed class Account permits CheckingAccount, SavingsAccount{}

public final class CheckingAccount extends Account
public sealed class SavingsAccount extends Account permits HighInterestAccount
```

Sealed Classes

- Sealed classes allow you to fully specify your domain.

```
public static void process(Account account) {
    if(account instanceof CheckingAccount) {

    } else if(account instanceof SavingsAccount) {

    }
}
```

Permitted Subclasses

- Must be accessible at compile time.
- Must directly extend the sealed class
- Must be final, sealed, or non-sealed

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept.
- Abstract classes allow you to establish common elements in a hierarchy that are too generic to instantiate.
- An abstract class *cannot* be instantiated.

- The modifier `abstract` is used to declare an abstract class

Abstract Classes

- An abstract class often contains abstract methods
  - An *abstract method* is a method header without a method body (no implementation)
  - The abstract modifier must be applied to *each* abstract method
  - An abstract method cannot be `final` or `static`
- An abstract class can (and typically does) contains non-abstract methods with full definitions.

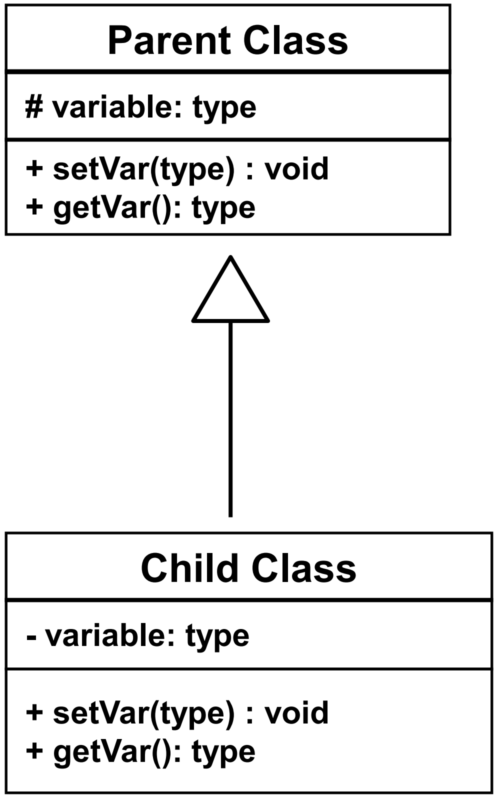
Design Questions

- Class should be abstract if…
  - It doesn’t make sense to instantiate objects of that type.
- Method goes in the parent class if…
  - All child classes should inherit this functionality.
- Method should be abstract if…
  - All child classes should implement it and
  - It doesn’t make sense to implement in the parent.

Practice

- Should any of our classes/methods be abstract?

UML Basics for Inheritance



Tutorial: https://www.tutorialspoint.com/uml/uml_class_diagram.htm

POLYMORPHIC REFERENCES

Binding

- Typically, we invoke a method through an object:

```
    myObject.doSomething();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes.
- If this binding occurred at compile time, then that line of code would call the same method every time.

Binding (cont.)

- However, Java defers method binding until run time- this is called *dynamic binding* or *late binding*.
- Late binding provides flexibility in program design.
  - Different methods can be called by the same line of code.

Polymorphism

- *Polymorphism* means “having many forms”
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- In Java, all object references are potentially polymorphic

Polymorphism (cont.)

- Let’s say we create the following variable:

```
    Person p;
```

- Java allows the variable `p` to point to an object of type `Person` or to an object of *any compatible type*
  - A compatible type can be established through inheritance or through interfaces
- We won’t know until run-time whether job is an `Person` object or another compatible type

Object References

- When you declare an object reference, you specify the type.
  - The compiler only knows about this type.
  - The compiler only allows you to invoke methods associated with the declared type.
- At runtime, the JVM knows the *actual* type of the object.
  - It could be the declared type or any subclass of the declared type.

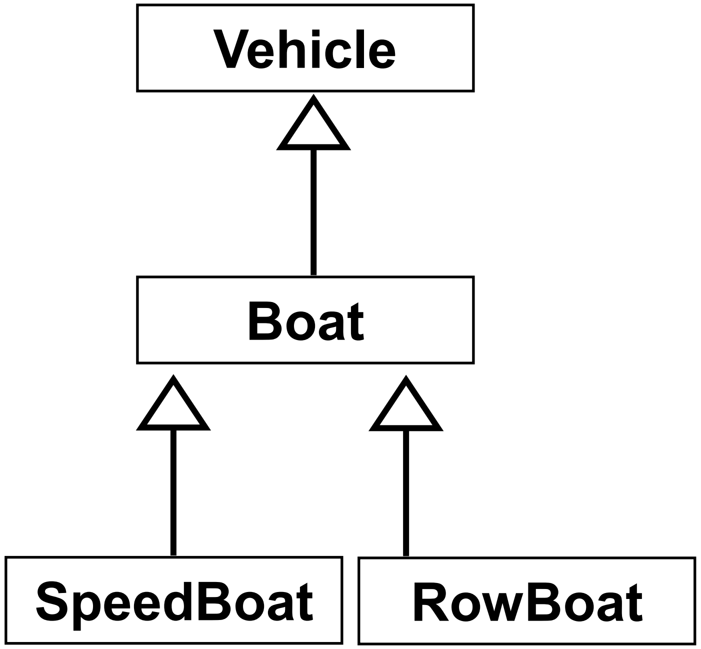
Object References

- You can declare an object to be of a type high up on the inheritance hierarchy.
- You can then instantiate that object to be of any type lower on the hierarchy.

- Why??
  - ParentClass[]
  - ArrayList<ParentClass>

References and Inheritance

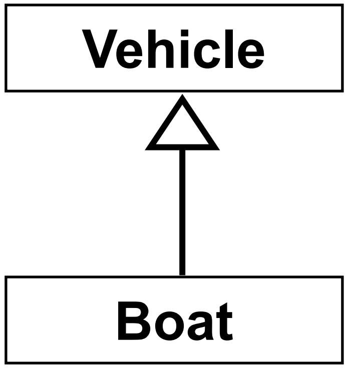
- The compiler only knows that v1, v2, v3, and v4 are of type Vehicle.
- But at runtime, the JVM knows that v1 is a Vehicle, v2 is a Boat, v3 is a SpeedBoat, and v4 is a RowBoat.



```
Vehicle v1, v2, v3, v4;
v1 = new Vehicle();
v2 = new Boat();
v3 = new SpeedBoat();
v4 = new RowBoat();
```

Widening Conversions

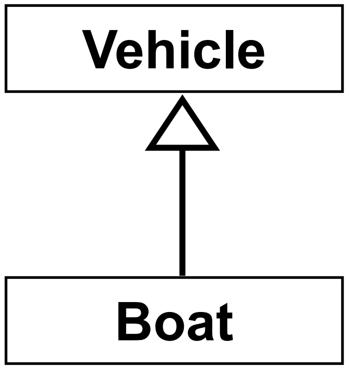
- Assigning a child object to a parent reference is a *widening conversion*
  - This can be performed with simple assignment
  - This is the most useful type of conversion



```
Vehicle vehicle1 = new Boat();
Vehicle[] vehicles = new Vehicle[5];
vehicles[0] = new Boat();
vehicles[1] = new RowBoat();
```

Narrowing Conversions

- Assigning a parent object to a child reference is a narrowing conversion
  - This must be done explicitly with a cast (downcast)
- Downcasting should always be paired with the instanceof operator.



```
Vehicle v = new Boat();
Boat boat1 = (Boat) v;
```

The `instanceof` Operator

- You can use the `instanceof` operator to return a boolean that represents whether an object is an *instance of* a class.
  - Will return true for all classes in the inheritance tree that match.
  - boatVehicle instanceof Boat   // returns true
  - boatVehicle instanceof Vehicle   // returns true

Declared Type vs Actual Type

- Declared type is determined at compile time
  - Controls which methods can be invoked: a method must exist in the declared class
- Actual type is determined at run time
  - Controls which version of the method is invoked

Class Practice

- Write a driver program to create and test your classes using polymorphic references.

On Your Own Practice

- Design a set of classes to represent a product sold at a store.
- Design at least one class and two subclasses (example: perishable, electronic, digital, etc.).
- Write a tester program to demonstrate your classes and methods.
- Consider:
  - Class and inheritance design
  - Instance data variables
  - Methods
- Optional: sketch your class design with UML