Classes and Objects

Recall that an object has state and behavior.

Consider a six-sided die (singular of dice):
- It’s state can be defined as which face is showing.
- It’s primary behavior is that it can be rolled.

We can represent a die in software by designing a class called `Die` that models this state and behavior:
- The class serves as the blueprint for a die object.
- We can then instantiate as many die objects as we need for any particular program.

Classes

The values of the data define the state of an object created from the class.

The functionality of the methods define the behaviors of the object.

For our `Die` class, we might declare an integer that represents the current value showing on the face.

One of the methods would “roll” the die by setting that value to a random number between one and six.

EXAMPLES: DICE & PALINDROMES
The Die Class

The `Die` class contains two data values:
- a constant `MAX` that represents the maximum face value
- an integer `faceValue` that represents the current face value

The `roll` method uses the `random` method of the `Math` class to determine a new face value.

There are also methods to explicitly set and retrieve the current face value at any time.

See `Rephactor Example: Dice`.

Code Walk-Thru

Walk-thru of `Rephactor Example: Dice`

Walk-thru of `Rephactor Example: Palindromes`

Encapsulation

We can take one of two views of an object:
- internal - the details of the variables and methods of the class that defines it
- external - the services that an object provides and how the object interacts with the rest of the system

From the external view, an object is an encapsulated entity, providing a set of specific services. These services define the `interface` to the object.

Encapsulation

One object (called the client) may use another object for the services it provides.

The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished.

Any changes to the object’s state (its variables) should be made by that object’s methods.

We should make it difficult, if not impossible, for a client to access an object’s variables directly.

That is, an object should be self-governing.

Encapsulation

An encapsulated object can be thought of as a black box -- its inner workings are hidden from the client.

The client invokes the interface methods of the object, which manages the instance data.
Inheritance

Inheritance allows a software developer to derive a new class from an existing one. The existing class is called the parent class, or superclass, or base class. The derived class is called the child class or subclass. As the name implies, the child inherits characteristics of the parent. That is, the child class inherits the methods and data defined by the parent class.

Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class:

- Proper inheritance creates an is-a relationship, meaning the child is a more specific version of the parent.

Deriving Subclasses

In Java, we use the reserved word `extends` to establish an inheritance relationship:

```java
class Car extends Vehicle {
    // class contents
}
```

The super Reference

Constructors are not inherited, even though they have public visibility. Yet we often want to use the parent's constructor to set up the "parent's part" of the object. The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor.

A child's constructor is responsible for calling the parent's constructor. The first line of a child's constructor should use the `super` reference to call the parent's constructor. The `super` reference can also be used to reference other variables and methods defined in the parent's class.
Multiple Inheritance

Java supports single inheritance, meaning that a derived class can have only one parent class.

Multiple inheritance allows a class to be derived from two or more classes, inheriting the members of all parents.

Collisions, such as the same variable name in two parents, have to be resolved.

Java does not support multiple inheritance.

In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead.