CSE 1325 - Object-Oriented Programming

Classes and Objects

Alex Dillhoff

University of Texas at Arlington

Object-Oriented Programming

Object-Oriented Programming involves abstracting the properties and behaviors of some concept or entity into objects.

The program is determined by how these objects interact with themselves and the rest of the code.

Objects are defined by putting the data first before defining the actions that manipulate that data.

Object-Oriented Programming

Throughout this course, we will cover the three main pillars of OOP:

- Encapsulation
- Inheritance
- Polymorphism

Classes

Encapsulation is the idea of combining data and behavior into a single entity.

Through this abstraction, the data and logic are hidden away from outside interference.

Classes

This prevents misuse of the data and ensures that only the intended behavior is executed.

In Java, this is done through the use of **classes**.

Classes

In OOP languages, an object is an **instance** of a particular class.

A **class** defines

- how the object is made,
- what data properties the object has,
- and the behaviors (methods) of the object.

Classes

The data, or properties, of an object are defined as **instance fields**.

Each object may have specific values set for its instance fields.

Classes

The concept of instance fields are intuitive when thinking about user accounts online.

A user has a unique name, some password, and maybe some personal information.

These would all be examples of instance fields.

Classes

The functions and behaviors that act upon the properties of an object are called **methods**.

An example of a method might be a `changeEmail()` method.

Classes

Generally, classes should be defined such that the instance fields are never directly accessible by other classes.

Any changes to those properties should be defined within class methods.

Classes

A general layout for a class is grouped in the following order:

```
public class MyClass {
    /* Instance Fields */
    /* Constructors */
    /* Mutators and Accessors */
    /* Methods */
}
```

Objects

A specific instance of a class is called an **object**.

Objects are characterised by their **behavior**, their **state**, and their **identity**.

Objects - Behavior

- What can this object do?
- What methods can we execute on a particular object?
- How can we change the underlying state or properties of an object?

Objects - Behavior

Examples

- A user can **log in**.
- An item can be **added to** a cart.
- A player can **join** a group.

Objects - State

- What defines the state of the object?
- How does the data change when interacting with this object?
- What is the current state of the data?

Objects - Behavior

Examples

- A user is **logged in**.
- An item is **in** a cart.
- A player is **in** a group.

Objects - Identity

- What makes this particular object unique?
- How can we compare two objects of the same type?

Objects - Identity

If the object needs to be truly unique, assign it a unique identifier (e.g. user IDs).

If the object is not unique, then it should be comparable to others based on one of its instance fields.

Object and Memory

In Java, all objects are created on the **heap** as opposed to the **stack**.

This is indicated by the fact that all objects are created with the `new` operator.

Unlike heap storage in C/C++, there is no operator to free memory for objects. All memory allocations are handled by the **garbage collector**.

Working with Objects

Java has many useful classes that can be used as part of the API.

So far, we have primarily used static methods from classes such as `Math`.

Let's create and use an object of the `Date` class.

The Date Class

Example: `DateExample.java`

Mutators and Accessors

Also referred to as **setters** and **getters**, these are methods that allow users to safely interface with the underlying instance fields.

Mutators and Accessors

Consider the following code.

```
LocalDate today = LocalDate.now();
LocalDate yesterday = today.minusDays(1);
```

Mutators and Accessors

The **mutator** method `minusDays(long)` allows the user to modify the underlying instance field representing the days without providing explicit access.

An explicit access might look like this:

```
yesterday = today.days - 1;
```

Mutators and Accessors

Likewise, an **accessor** method will provide access to the underlying instance field.

```
// Gets the month value from 1 to 12
int month = today.getMonthValue();
```

Mutators and Accessors

Why use these methods instead of providing direct access?

Mutators and Accessors

Why use these methods instead of providing direct access?

- Future-proof - if the underlying representation needs to change, it does not affect how other methods and classes utilize the call.
- Robust - by abstracting the underlying field, errors can be checked and handled when bad input is given.

this - The Implicit Parameter

Consider the following mutator method:

```
public void setName(String n) {
    name = n;
}
```

There is an obvious (explicit) method parameter: `String n`.

this - The Implicit Parameter

Consider the following mutator method:

```
public void setName(String n) {
    name = n;
}
```

There is also an implicit parameter: this.

this - The Implicit Parameter

this refers to the object that the method is being called on.

Thus, the method shown before can also be written as

```
public void setName(String n) {
    this.name = n;
}
```

Mutators and Accessors

A natural and reasonable question to have at this point is...

Why use mutators and accessors over declaring a `public` instance field?

Mutators and Accessors

The simple answer is that developers will make mistakes.

By restricting the way in which instance fields can be updated, we are adding more definition to a class.

Mutators and Accessors

Consider an `public` instance field that stores an e-mail address.

It is perfectly reasonable to assume that the e-mail could be set to an invalid address
(e.g. `not_a_real_address@domain`).

Mutators and Accessors

If this was assigned to a `public` instance field, there may or may not be any validation to make sure the address is valid.

When the address is used later on in some other context, it could lead to other issues.

Mutators and Accessors

If the instance field was `private` with a corresponding mutator method, additional validation code could be added to make sure invalid addresses cannot be entered.

Accessors that Return Objects

Be careful when creating an accessor that returns an object.

Consider the following accessor which returns an instance field.

```
public Date getCreationDate() {
    return creationDate;
}
```

Accessors that Return Objects

In this example, `creationDate` is an instance field of type `Date`.

In this case, a reference to the object is returned.

Calling a method such as `setTime()` will modify the underlying data of the original object.

Accessors that Return Objects

Example: `OrderReferenceExample.java`

Accessors that Return Objects

If you are working with a mutable object that is returned in a call, consider making a clone of it.

Of course, if your intent is to return a mutable object then no change is necessary.

Designing Classes

How do you know when to define a class?

Designing Classes

How do you know when to define a class?

Look at the *nouns* in your project description.

Designing Classes

Example

Create an app where users can share images with each other. Users can add individual images to their favorites list.

Designing Classes

We can easily identify a **user** class.

Some actions are explicit:

- Post images
- Add other images to their favorites

Designing Classes

Other behaviors are implicit or come from domain knowledge.

- Users need to be able to **log in** and **log out**.
- If a user can add images to a favorites list, that list should an instance field in the class.

Designing Classes

Another Example

Design a class that represents a player in a Role Playing Game. The player should have a **name**, some **hit points**, possibly an **inventory**, and whatever other properties you can think of.

Designing Classes

Let's begin to put together a class based on the previous description.

Example: `Player.java`

Static Fields and Methods

So far, the instance fields and methods we have used for the `Player` class have all required that an instance of the object be created first.

We also have the ability to create `static` fields and methods.

Static Fields and Methods

Consider the following instance field.

```
private static int playerId = 1;
```

This field will be shared by all instances of the class. That includes having the ability to modify it.

Static Fields and Methods

We can also make these fields constant using the `final` keyword.

We have seen this used in the Java API, specifically the `Math` class.

```
public static final double E = ...;
```

Static Fields and Methods

The constants from the `Math` class are both `public` and `static`.

This implies two things:

1. They are accessible outside of the class.
2. They do not require an object to be created in order to use them (no implicit parameter).

Static Fields and Methods

`static` methods are useful when we have an action related to our class that is defined by explicit parameters.

An example of this would be `Math.max(double a, double b)`.

Static Fields and Methods

This method does not require an instance of an object to use, nor does it require any internal or `private` instance fields.

Static Fields and Methods

A special case of `static` methods involve constructors.

Consider the method `public static LocalDate now()`.

Static Fields and Methods

This method will create a new instance of `LocalDate` and set the time based on the current value of the local clock.

Methods such as these, which return an object as part of their definition, are called **factory methods**.

Static Fields and Methods

Factory methods allow us to vary the types of object instantiation we can perform in our classes.

They also allow us to write method names that are more specific instead of being stuck with the class name itself.

Using the Class

Now that we have a custom class to work with, let's implement and use it in a program.

Typically, the classes you create will not include a `main` method.

Instead, you will use them as part of a larger program.

Using the Class

In the next example, we will instantiate an object of type `Player` similarly to any other object.

```
Player p1 = new Player("Vex'ahlia");
```

We are able to use any of the constructors that we defined in the original class definition.

Using the Class

A few key points about constructors:

- They are always called with the `new` operator.
- They share the same name as the class.
- They have no return value.
- If none are defined, a default constructor is assigned (sets all values to 0).

Using the Class

Example: `PlayerTest.java`

Running the Code

In this example, we have multiple source code files.

When compiling with C, we would give as input all code files needed for our program.

With `javac`, we only need to input the source code file containing the `main` method.

Running the Code

`javac` will behave similar to `make` in that it will look for any dependences referenced in the main code file.

If any of those dependencies need to be re-compiled, it will automatically do so.

Securing the Constructor

Any object variable can hold a `null` value to indicate that there is no object.

What happens if we try to pass a `null` value as the name when creating a `Player`?

```
Player p1 = new Player(null);
```

Securing the Constructor

Initially, nothing bad will happen until we attempt to call a method on the name, a `String` object.

By that time, it will not be clear that the error originated within the constructor.

Securing the Constructor

There are a couple ways to resolve this:

- Check for `null` using a control statement.
- Use `Objects.requireNonNullElse`
- Use `Objects.requireNonNull`

Example: Modify `Player.java`

Destructors in Java

Unlike C++, Java has no destructor method.

The garbage collector will automatically return all unused memory.

However, other resources that are open will not be automatically reclaimed.

Destructors in Java

Resources such as file handles must be closed as soon as you are done with them.

The best way to ensure this happens is to keep your resource accesses self-contained.

For example, create a method which opens the file, writes the data, and then closes the handle.

`static` Imports

We can statically import packages which may be convenient for writing more compact code.

Consider the statement:
`Math.sqrt(Math.max(Math.min(a, b), c))`.

This statement chains together multiple `static` methods from the `Math` library.

`static` Imports

We can forego the explicit reference to the `Math` class by statically importing it:

```
import static java.lang.Math.*;
```

We can now reference the methods directly (e.g. `sqrt(max(min(a, b), c));`).