# CS 251
# Intermediate Programming
# Methods and Classes

Brooke Chenoweth

University of New Mexico

Spring 2024

# Methods

- An operation that can be performed on an object
- Has return type and parameters
  - Method with no return type (`void`) is often called **procedure** in other languages.
  - Method that has a return type, may be called **function** in other languages.
  - In Java - They are all called methods
- Can be overloaded. What does *that* mean?
- Many already available methods. . . (See JDK API)

# What is Overloading?

- A method is identified by its name, and the types of its arguments. You can declare several methods with the same if the argument types differ.
- Argument order is important!
- It is not possible to declare two methods that only differ in the return type. (Why?)
- Can not declare one method `circleArea` with the radius as argument, and one `circleArea` with the circumference, unless radius and circumference have different types.

# Overloading Example

```
public class OverloadExample {

  public static void writeType(int x) {
    System.out.println("int");
  }

  public static void writeType(char c) {
    System.out.println("char");
  }

  public static void writeType(String s, float f) {
    System.out.println("String + float");
  }

  public static void main(String[] args) {
    writeType(1);
    writeType('a');
    writeType("Hello", 3.1415);
  }
}
```

# The `static` keyword

- Static methods and Static variables
- Mixing static and non static
- Programming style - use a small main
  - Only contain method calls

# Static variables

- Constants - `public final static`
- Variables - `public static`
- When do we use them? Why?
- Static vars in other classes. How to use them?

# Static methods

- What is a static method?
- When do we want to use them?
- `main` is always static
- Example: `Math` class

# Put a main in any class!

- May be something new.
- Very useful for testing
- No need to compile "whole" program, just class
- Remember previous slide. Instantiate object to test it.
- Other reasons for multiple main methods?

# Nonstatic from static

- Problems with calling non-static methods from a static method. Why?
- Solution 1: Instantiate an object and invoke method with it.
- Solution 2: Add a parameter to take object of object type and use it. (Won't work for `main`)

# Wrapper classes

- Integer, Character, & Double
- Has many useful static methods.
- Initialization and casting between primitive
- Autoboxing and unboxing

# What is an object?

Each object has certain data and behavior

- An example: *student*
  - Data: age, endurance, intelligence, ...
  - Behavior: code, drink, workout, sleep, ...
- Another example: *car*
  - Data: power, top-speed, shape, color, etc...
  - Behavior: start, accelerate, break, turn

# What is a class?

- A class is a blueprint from which objects are created.
- An object created from a class is an *instance* of that class.

# Example class

```
public class Student {
  private int age, endurance, intelligence;

  public Student ( int age, int endurance, int intelligence ) {
    this.age = age;
    this.endurance = endurance;
    this.intelligence = intelligence;
  }

  public void drink ( String what ) {
    if ( what == "milk" ) {
      endurance++;
    } else if ( what == "alcohol" ) {
      if (age >= 21) {
        intelligence = intelligence - 5;
      } else {
        System.out.println("You are too young to drink!");
      }
    } else {
      System.out.println("Don't drink " + what + "!");
    }
  }
}
```

# Find mistakes!

- What's wrong with the program on previous page?

# The `String` trap

- Why shouldn't you compare two strings with the == operator?
- Reference types!
  - A reference to a place in memory - a comparison with the == operator compares addresses of memory.
  - Are the two references both refering to the same object?
- When comparing two objects, usually want to use `equals` method.

# Example class revisited

```
public class Student {
  private int age, endurance, intelligence;

  public Student ( int age, int endurance, int intelligence ) {
    this.age = age;
    this.endurance = endurance;
    this.intelligence = intelligence;
  }

  public void drink ( String what ) {
    if ( what.equals("milk") ) {
      endurance++;
    } else if ( what.equals("alcohol") ) {
      if (age >= 21) {
        intelligence = intelligence - 5;
      } else {
        System.out.println("You are too young to drink!");
      }
    } else {
      System.out.println("Don't drink " + what + "!");
    }
  }
}
```

# Class vs Instance variables

Instance variables

- Non-static fields
- Every object has its own
- Need instance to use

Class Variables

- Static fields
- Associated with *class*, not a particular object
- Can be manipulated without an instance

# Class and Instance Variable Example

```
public class Student {
   // These are instance variables
   private String name;
   private int id;

   // This is a class variable
   private static int numberOfStudents = 0;

   public Student ( String name ) {
      this.name = name;
      // Give each student a unique ID
      this.id = ++numberOfStudents;
   }

   // More methods here...
}
```

# Access Modifiers

| | |
|---|---|
| `public` | Accessible to all |
| `private` | Only this class |
| `protected` | Only this class and its subclasses[1] |
| *package-private* | No modifier.[2] This class and others in same package. |

[1] and also same package

[2] Students usually use this by accident when they forget the modifier.

# Access Modifier Tips

- Don't expose your guts!
- Use `private` unless you have a good reason not to.
- Avoid public fields except for constants. (Use getter/setter)

# Encapsulation - Creating an API

- API = Application Programming Interface
- Define a set of rules for an object (i.e., what public methods should be available)
- A well defined API will help in large projects
- Will reduce time for redesign and integration
- Is what we will strive to achieve
- Will require a certain design component in later programming projects

# How to think object orientation

- Look at problem description – Identify the following:
  - Verbs (possible methods)
  - Nouns (possible objects - or instance variables)
- Think early on how these objects will interact - Diagrams!
- What information (possibly objects) will need to be passed between them
- Then what? Put the design to test - have someone else critique it.
- Revise your design - Start Implementation

# Creating the API

- First part of implementation is to realize the API
  - Create all classes, with method stubs only
  - Write initial documentation for each object and method - This way you clearly know what each method is supposed to do, and might find flaws in the design when you think about it more.
  - Use Javadoc for your comments – Creates nice webpages for the API automatically.

# Encapsulation Guidelines

1. Place comment in front of class to define how to think about the class.
2. All instance variables should be declared `private`
3. Provide mutator and accessor methods for state change
4. Use comment before each method, describing it's use
5. Make all helper methods `private`
6. Use `/** */` comments for API comments and `//` for implementation details

# Writing javadoc comments. (Page 1)

```
/**
 * A class to demonstrates the usage of javadoc documentation
 * features.
 * @author Brooke Chenoweth
 * @version 1.1
 */
public class JavadocDemo {

  private String name;   // Name of the object
  private int desc;      // Description of object

  /** The constant number of the democonstant. */
  public static final int DEMOCONSTANT = 5;

  /**
   * The default constructor. Defines the empty JavadocDemo
   * class with default values set. Typically implicitly
   * called by subclasses.
   */
  public JavadocDemo () {
  }
```

# Writing javadoc comments. (Page 2)

```
  /**
   * Preferred constructor.
   * @param      desc     Describes the entity in the object.
   * @param      name     Names the entity in the object.
   */
  public JavadocDemo (int desc, String name) {
    this.name = name;
    this.desc = desc;
  }

  /**
   * Changes the name of the object and makes sure name is valid.
   * @param      name     Proposed new name for object
   * @return              True if name accepted, false if not.
   */
  public boolean changeName ( String name ) {
    if ( name.equals(this.name) ) {
      return false;
    }
    this.name = name;
    return true;
  }
}
```