CS 5004: Lecture 7

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- Main method
- Midterm logistics
- Midterm review

`main` method

Java applications

Applications written in Java:

- Many Android phone apps
- IntelliJ
- Other IDEs e.g. Eclipse
- Original version of Minecraft
- Many more

Java applications

*.jar = a runnable Java application

source code

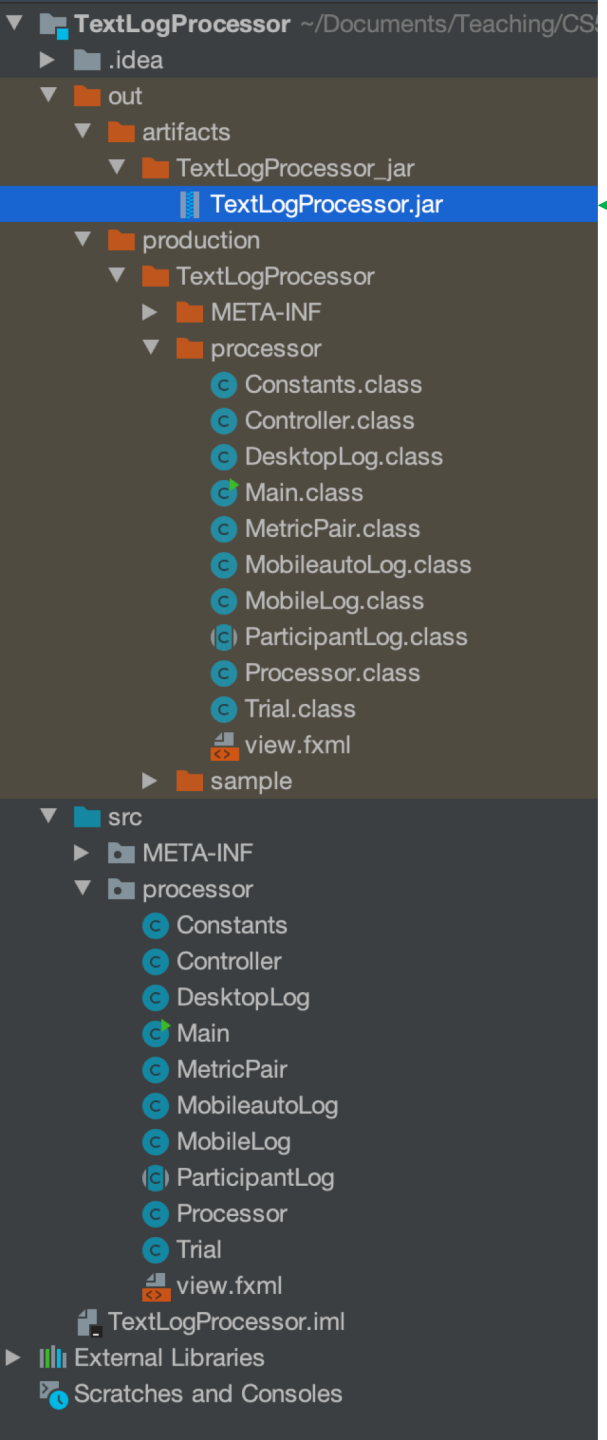
`*.java`

compiled code

`*.class`

application

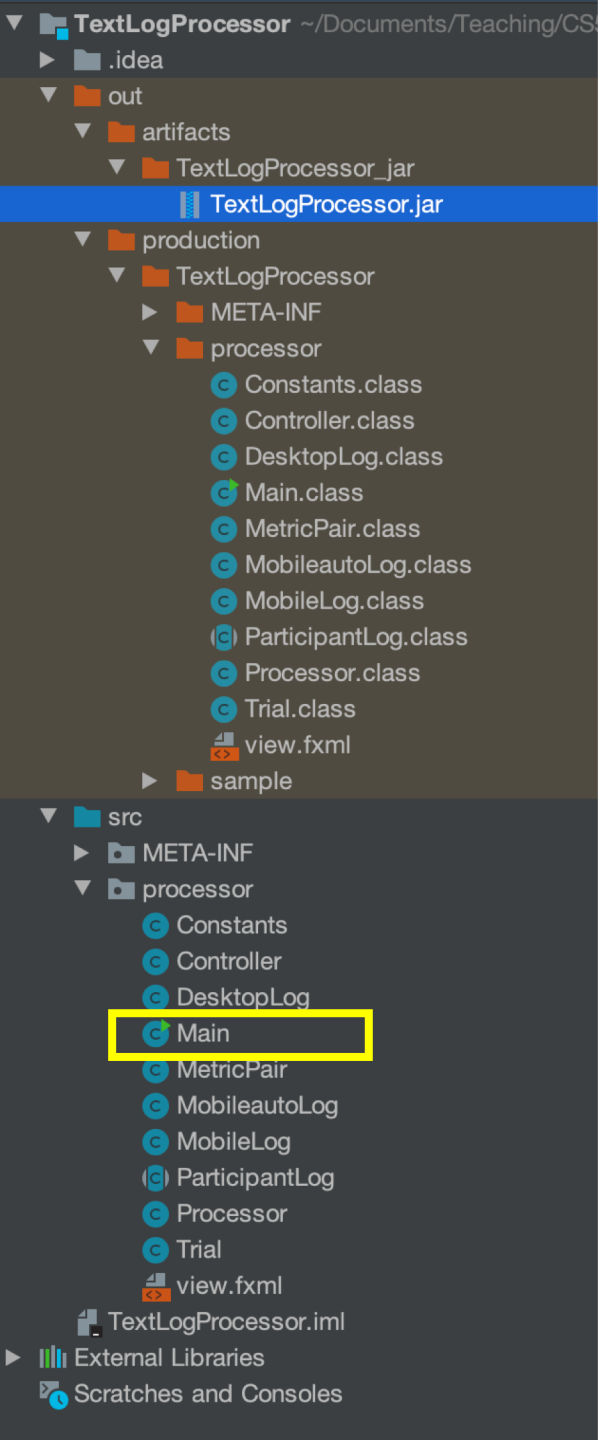
Example Java application



application

compiled code

source code



Example Java application

Before the application can be created, the program needs an **entry point**

- A *single class* that runs when the application is run
- Handles input from user, if any
  - GUI
  - command line
- Coordinates the ”business logic”
- Handles output to user, if any
  - GUI
  - command line

Defining an application entry point

- Some class, often called **`Main`**
- Must have a **`main`** method
  - Signature must be exactly as follows...

```
public static void main(String[] args) {
   // business logic here
}
```

Defining an application entry point

- Some class, often called **`Main`**
- Must have a **`main`** method
  - Signature must be exactly as follows...

```
public static void main(String[] args) {
   // business logic here
}
```

Command line arguments passed in here

`java MyApp hello 9000`

`args` will equal **`["hello", "9000"];`**

A silly example...

- animals > Main.java

Midterm logistics

Midterm next week

Logistics (keep an eye on Piazza for changes)

- **Available between 3/10 and 3/17**
- **Once opened, you must submit within 6 hours**
- Focused on writing code (no multiple choice).
- Submit via GitHub – may be a small acknowledgement in Canvas as well.

Midterm topics

- Types of data
- Classes and objects
- Documentation
- Writing methods
- Unit testing
- Exceptions
- Enumerations and switch statements
- Inheritance
- Interfaces and abstract classes
- Abstract Data Types
- Lists
- Arrays

Exam format

Two sizeable questions:

- Focused on object-oriented design
- “Correct” solutions that do not follow OOD principles will not score well!

Exam format

- Don’t need to write Javadoc unless specifically requested
- Don’t need to test unless specifically requested
- Don’t need to generate equals / hashCode / toString unless you need to use them
- DO define constants (points deducted for magic numbers)

Study resources

Past midterms – Canvas > Today's module > Old midterms

- Ignore problem 1 on each exam
- Note the similarities over the semesters!
- Optional review sessions will be posted on Piazza
- Focus on most recent exams
  - Note: almost all past exams were on paper so some of the instructions might not apply!

General test-taking tips

- Read the **whole** exam before starting
- Take time to think and plan your approach before jumping in

Tips for success in this exam

CS5004 Object-Oriented Design

not

CS5004 Programming in Java

Tips for success in this exam

CS5004 **Object-Oriented Design – follow OOD principles**

not

CS5004 Programming in Java

Tips for success in this exam

Apply OOD principles:

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Solutions that do not use OOD will earn negligible points, even if functionally correct

Tips for success in this exam

Apply OOD principles:

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

In practical terms:

- Use inheritance
  - **interfaces**
  - abstract classes
- Plan before doing
- The OOD solution is not always the easiest one to write
  - Often the easiest one to use

Tips for success in this exam

Practice applying OOD principles

- Code to an interface
- Use inheritance
- If you haven't applied or aren't comfortable with a particular concept, find a way to practice it
  - Rewrite older assignments to incorporate the concept

Tips for success in this exam

Unlike (Seattle, evening) CS 5001 last semester:

- You will lose points for details like magic numbers
- We care about design, not just correctness!

Midterm review

Review topics

- OOD principles
- Enums:
  - When to use
  - When to avoid
- When to use which type of inheritance
- Abstract Data Types
- Practice problems

OOD principles

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

- Encapsulation
  - Keep all the properties and behaviors associated with an object together
  - Methods that update an object should be written in the object's class
  - If you have information about an object's property split across multiple fields, those fields should be encapsulated in their own class
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
  - Keep implementation and interface separate
  - Interface – outlines what an object can do but doesn't actually do it
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
  - Expose only the necessary functionality to users of a class
- Polymorphism
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
  - Different classes/methods have the same name but handle different data types
  - An object can act as if it were a different (but related) data type
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance
  - Classes can extend or override functionality from other classes

When to use enums

An enum is…

- A way to represent a set of finite constants.

- Days of the week
- Possible outcomes of a game (win, lose, or draw).
- Directions (N, S, E, W).

An enum is…

- A way to represent a set of **finite** constants.
  - i.e. the set is not likely to change

- Days of the week
- Possible outcomes of a game (win, lose, or draw).
- Directions (N, S, E, W).

Uses of enums

Yes

To represent a reasonably **complete** range or set of values of a field

- E.g. Sizes: Small, Medium, Large

No

To represent a range of values described in a spec that wouldn't be a "complete" range outside of the spec

- Sports: Soccer, Football, Baseball

Uses of enums

Yes

When 1 or 2 pieces of functionality are dependent on the value

- e.g. Size – will the car fit in the spot?

Or, it's for information only

- e.g. State in address

No

When many pieces of functionality are dependent on the value

- e.g. Sport
  - Structure and update of Record
  - Calculation of points
  - Whether or not a tie is allowed

Why aren’t enums recommended in these cases?

To represent a range of values described in a spec that wouldn’t be a “complete” range outside of the spec.

- Specs change and grow
- Modifying code using enums in this situation is messy and error prone

Why aren’t enums recommended in these cases?

When many pieces of functionality are dependent on the value.

- Inheritance and polymorphism reduce code complexity and better support encapsulation

A sign that an enum should be subclasses…

Multiple if/switch statements branching on the value of an enum

- This suggests your enum values may actually be “types”

A sign that an enum should be subclasses…

Multiple if/switch statements branching on the value of an enum

- This suggests your enum values may actually be “types”

Practical tip: in a 5004 exam/assignment, favor a class over an enum unless specifically told to use an enum

Why?

- We want to assess your ability to apply object-oriented design, especially inheritance

Common exam issue

String or enum used to represent a type instead of classes

- Tempting because it feels like less to write
- DON’T DO IT

Practical tip: in a 5004 exam/assignment, favor a class over an enum unless specifically told to use an enum

Why?

- We want to assess your ability to apply object-oriented design, especially inheritance

When to use which type of inheritance

Types of inheritance

- Implementing an interface
- Extending an abstract class
- Extending a concrete class

When to use an `interface`

- More than one subclass must have the same method signatures
- ***Don't avoid interfaces in the exam*** –fundamental to OOD

When to use an `abstract` class

- More than one subclass will have the same implementation for at least one method
- More than one subclass will have at least some of the same fields
- …and you don't want client code to instantiate the shared implementation directly (e.g. because it's not a complete representation)

When to use an `abstract` class

Writing the exact same code in multiple classes?

- Put it in an abstract parent class… (use helper methods too)
- Call the method from the concrete class if needed e.g.
  - `super.methodName()` if overriding the same method
  - `this.methodName()` if it's a helper method

When to use an `abstract` class

Use **overriding** to combine abstract and concrete functionality

```
// In the abstract class
public double methodName() {
   ..
   return x;
}

// In a concrete child class
@Override
public double methodName() {
   return super.methodName() +
        this.myHelperMethod();
}
```

When to `extend` a concrete class

- You have a concrete class that it makes sense for clients to use directly
- You have a subclass that represents a “special case” of the existing class.

Combining interfaces and abstract classes

When all subclasses need the same subset of methods but the implementation of some of the methods is different by subclass.

- Put signatures of all required methods in an interface
- Put the shareable implementations in the abstract class.

Combining interfaces and abstract classes

When all subclasses need the same subset of methods AND the same subset of fields.

- Put signatures of all required methods in an interface
- Put the fields and “super” constructor in an abstract class

When an interface is unnecessary

- All subclasses need the same set of methods AND implementation is identical (or very similar) for all of these methods
- Otherwise, use an interface
  - Interface should be the default

Inheritance is a hierarchy

Combine interface and abstract classes to suit the problem

- An interface > one abstract class > concrete subclasses
- An interface > one high level abstract class > multiple lower level abstract classes > concrete subclasses
- One abstract class > concrete subclasses
- One interface > concrete subclasses

Common exam issue

All calculations for all child classes implemented in one method in an abstract class.

Violates:

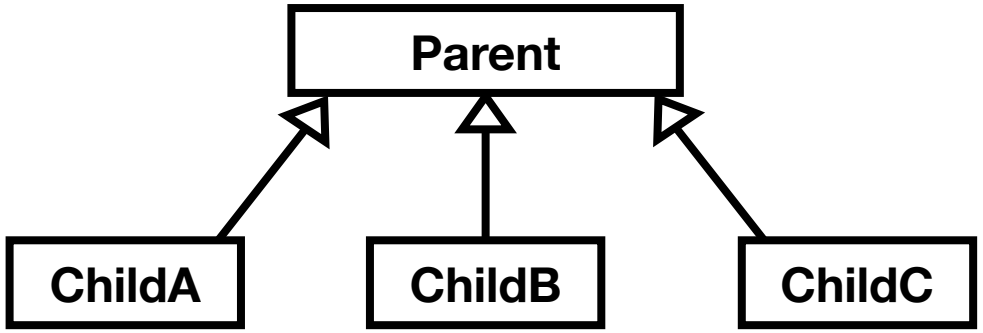
- OOD: Encapsulation
- OOD: Inheritance
- General good practice (avoid giant methods)

Encapsulation

- Keep all the properties and behaviors associated with an object together
- Methods that update an object should be written in the object's class
- **If some piece of code is specific to a particular class, put it in that class**

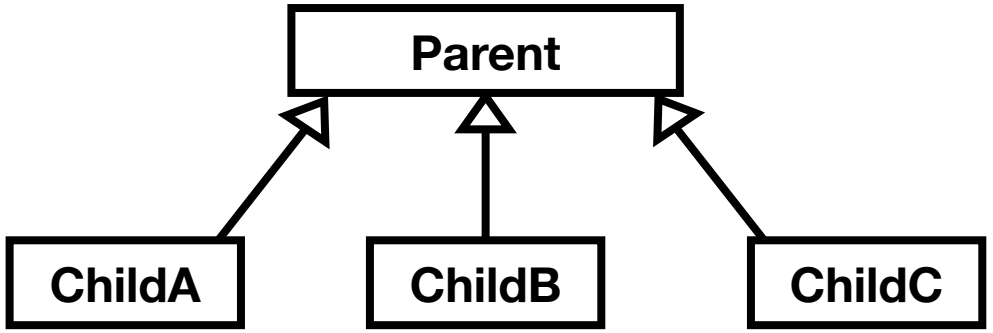
Inheritance

- Child classes inherit properties and behavior from parent classes
- **Parent classes know nothing of their child classes**
  - **One-way relationship**
  - Only exception: static creator methods for an ADT implementation → you *can* opt to put these in the interface



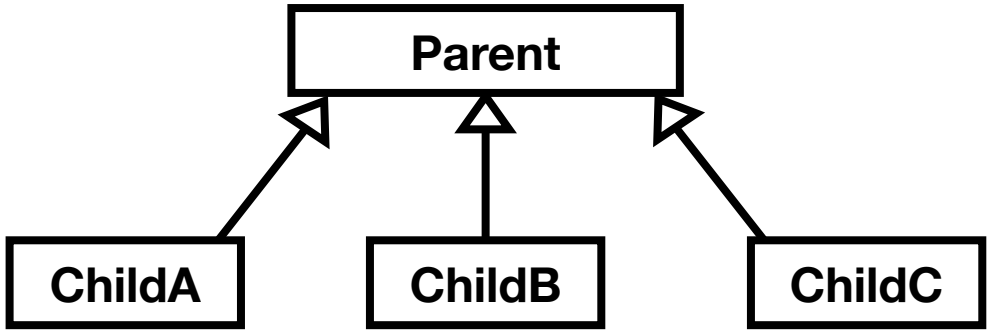
Inheritance

Code common to all three children should be in Parent



Inheritance

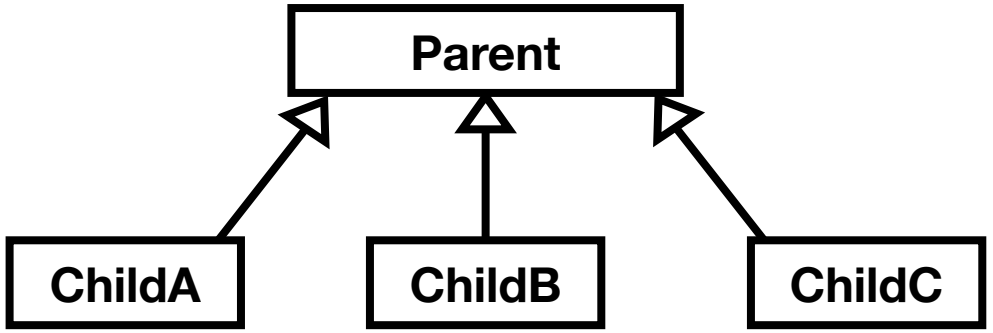
Parent should not contain any references to children



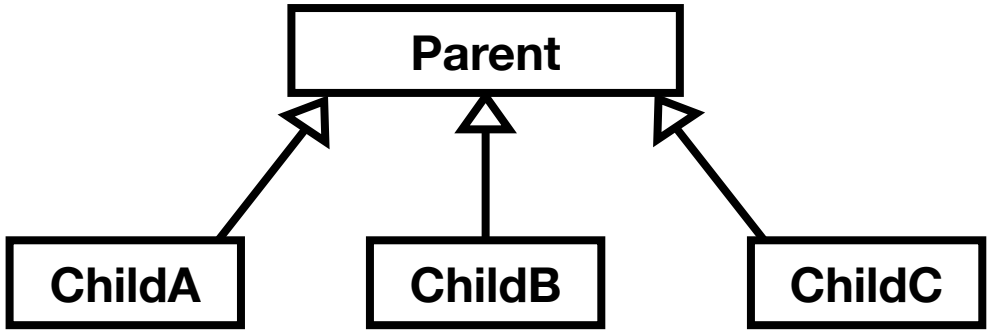
Inheritance

Parent should not contain any references to children

- No checking child type: `instanceOf ChildA`
- Instead – put ChildA-specific code in ChildA class.



Inheritance



Parent should not contain any references to children

- No overloading with child type:

```
methodName(ChildA child)
methodName(ChildB child)
methodName(ChildC child)
```

- Instead – override `methodName` in child classes

Abstract Data Types

Abstract Data Type

- A specification for a data type
- Specifies public methods (what the client can use)
- Doesn’t get into implementation details

Specifying an ADT

- Done from the **client**'s perspective
- What will users of this ADT need it to do (what public methods will they need)?
- Write an **`interface`** containing all method signatures
  - Except anything static that would require knowledge of implementing classes

Implementing an ADT

- You choose underlying data structure
- Your implementation must match the specification
- For collections of things (e.g. List, Stack, Set), will be one of the following:
  - Array
  - Linked list - a recursive or self referencing data structure
- Add **`private`** helper methods as needed

Writing tests for an ADT on the exam

- Write tests from the interface
  - No need to write separate tests for each node class
- Terminology:
  - ”Black box tests” – tests written without knowledge of implementation details (e.g. lecture 6 – ADT tests written before implementation)
  - “White box tests” – tests written with knowledge of implementations details (e.g. all tests you’ve written)

Break (10 mins)

Practice

Practice

- We'll take a look at a past exam together (Spring 2019)
- Practice problems on Canvas
  - Focused on strategy for the design problems as well as content
- We will go over each problem toward the end of class