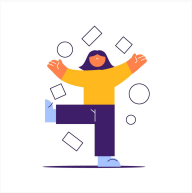
Lec 19: Inheritance Contd., Queues

Sam McCauley

April 9, 2026

Admin



- Lab today
  - Get started early
  - Remember that attendance is required. If you're done, just come in and talk to me or Lida about your solution
- Midterm next Friday! I'll start posting materials soon. Focus will be on topics since last midterm.
- Also I'll catch up on grading soon, but I'm prioritizing the practice midterm until it's posted

try/catch

Handling Exceptions

- When something goes wrong, Java often throws an exception (and the program crashes)
- Sometimes we'd really like it if the program didn't crash, and something else happened instead.
- Example: `Integer.parseInt(str)` will throw an exception if `str` does not hold an integer. We'd like it if the program did not crash, and instead just asked the user to input a different value
- Enter: `try/catch`

try/catch

- Put the line of code that may throw an exception in the `try` block
- The `catch` block states what exception it is looking for. If the line of code does give the exception, then rather than crashing, the `catch` block is run and then execution continues.

```
try {
  int i = Integer.parseInt(str);
} catch (NumberFormatException e) {
  System.out.println("Error: Input is not a valid integer.");
}
```

Let's try this out.

try/catch

- Can catch any kind of exception
- But, not always the best way to handle errors.

```java
int numerator = 10;
int denominator = 0;
//this is better handled with: if(denominator == 0)
try {
    int result = numerator / denominator;
} catch (ArithmeticException e) {
    System.out.println("Error: Cannot divide by zero. ");
}
```

When to use try/catch

- If can use an if statement instead, probably should
- Not always possible (or at least feasible)
- Example: I don't know how to write an `if` statement to say "will `Integer.parseInt()` be able to parse this string" So in this context, `try/catch` is appropriate

try/catch

- We'll only use exceptions to handle bad user input in this course
- But, is much more powerful than this one use case
- To catch an exception, look up the type, and figure out a way to handle it gracefully
- We won't cover in this class, but Java has extensive tools for dealing with exceptions: methods can "pass them up" to other methods by themselves throwing exceptions, can move them around/store information in them (they are objects!), etc.

Inheritance: Some Final Rules and Comments

Constructors and Inheritance

- The first line of a subclass' constructor *must* call the parent constructor.
- If you don't call it, Java will add a `super()` call for you
- Note that this means that—if you do not specify the parent constructor call—the parent class *must* have a constructor with no arguments. If it doesn't, your code won't compile!
- You may not call `super()` (with or without arguments) anywhere else; it has to be the first line of the subclass' constructor.
- When creating a subclass, ask yourself: which parent constructor should be called before I do anything else?

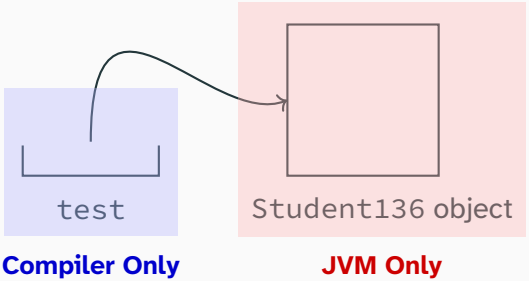
Calling Grandparent Methods

- Question: if we use `super.method()` to call the parent (not overridden) version of `method()`, how do we call the grandparent's version? Maybe something like `super.super.method()`?
- Answer: You *can't* in Java, at all. A subclass only calls its own methods, and its parent's methods.
  - Keeps things simple on larger projects: when modifying a class, only need to worry about children calling the method (rather than a whole tree of descendants)
  - There are examples where using the grandparents' method allows you to dodge parent checks/invariants.
  - If the *parent* method has a `super` call, you can indirectly call the grandparent method that way. But it has to be through the parent; you can't go above their head

What Method Gets Called?

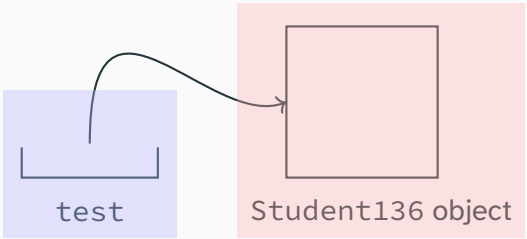
- One of the tricky parts of overridding methods is figuring out which version gets called
- Example: let's say we run something like the following:

```
1 Student test = new Student136("test", "test");  //Allowed!
    Every Student136 is a Student
2
3 String example = test.toString();  //does this call the
    Student version or Student136?
```



What Method Gets Called?

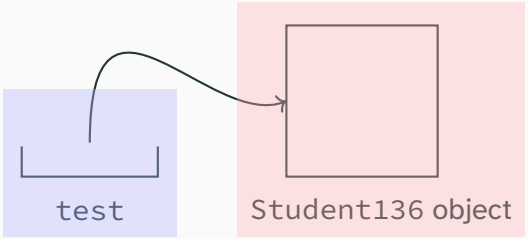
```
1  Student test = new Student136("test", "test");
2
3  String example = test.toString();
```



The JVM uses only the *object itself* to determine what version of the method to call. It does not care one bit about how the reference is stored.

What Method Gets Called?

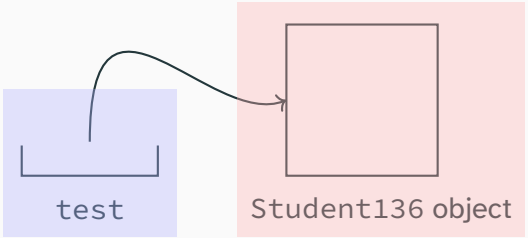
```
1  Student test = new Student136("test", "test");
2
3  String example = test.toString();
```



Type checking is something the compiler uses to make sure your code works—when the compiler is working, the object is not built yet, so the compiler only looks at types.

What Method Gets Called?

```
1  Student test = new Student136("test", "test");
2
3  String example = test.toString();
```



Answer: the JVM calls the `Student136` version. Storing it as a `Student` only affects compilation.

Inheritance Example

Inheritance

- Inheritance is not just about extending functionality
- It can also be used for *organizing* code
- Let’s look at a project which could easily be written in a single class. We’ll write it using inheritance instead

Encryption

- Use a secret key to jumble a message. A random person looking at the jumbled message (hopefully) can't read it
- But your friend with the same key can read it!
- Let's look at a couple classic (meaning bad) encryption schemes
- We'll use only capital letters; no punctuation
- I won't test you on these encryption methods. But I want to go over them quickly so that we understand how inheritance helps implement them. (Plus, encryption is fun!)

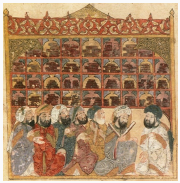
Caesar Cypher

- Key is some number *k* from 1–25
- Encrypt: Move each letter “ahead” by *k* letters, wrapping around if you go past Z
- Decrypt: Move each letter “back” by *k* letters

Viginère Cypher

- Key is an arbitrary string, let’s say the string is "ABD"
- Encrypt: move each letter “ahead” by the index of the corresponding letter in the key.
    - In our example, the 1st letter is moved ahead by 1 for `A`, the next by 2 for `B`, the next by 4 for `D`, then we loop around so the fourth letter is moved ahead by 1 for `A` again
- Decrypt: move each letter “back” by the index of the corresponding letter in the key.

Substitution Cypher



- Key consists of a list of 26 letters denoting how to map each letter to a new letter
- Encrypt: replace each letter with the new letter
- Decrypt: replace each letter with its original

Implementing All 3 Cyphers

Let’s think about the similarities here:

- All three will have an encrypt, decrypt function (which we’ll call to test them)
- All three need a key, though the type differs
- All three need a way to convert characters to integers and back
- The Viginère and Caesar cypher work pretty similarly. In fact, the Viginère cypher literally does the same thing, just with a rule to change the key on the fly

Design Plan

- We'll have a `Cipher` class with useful methods for all three ciphers
    - Its encrypt and decrypt methods won't do anything: the message will just map to itself
    - It will also have our `main` method, and a helper method for testing
- `CaesarCipher` and `SubstitutionCipher` inherit from the `Cipher` class, overriding the encrypt and decrypt methods
- `VigenereCipher` will inherit from `CaesarCipher`, overriding the encrypt and decrypt methods
- Let's look at the code.