

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

### Introduction

# Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- Most service development in Fidelity is done in Java. Java is the most used programming language in the world.
- It is important to be able to translate an object-oriented design into working code.
- Test-Driven Development (TDD) is a useful Agile technique that helps in producing high-quality software.

# Course Outline

- Chapter 1      Introduction to Java
- Chapter 2      Test-Driven Development
- Chapter 3      Implementing Classes
- Chapter 4      Implementing Polymorphism
- Chapter 5      Effective Programming
- Chapter 6      Useful Classes
- Chapter 7      Constants and Enumerations
- Chapter 8      Abstract Classes and Interfaces
- Chapter 9      Java Collections Framework

# Course Outline (continued)

- Chapter 10 Refactoring
- Chapter 11 Exceptions
- Chapter 12 API Design
- Chapter 13 Implementing Design Patterns
- Appendix A Java Basic Syntax
- Appendix B Gang of Four Design Patterns

# Course Objectives

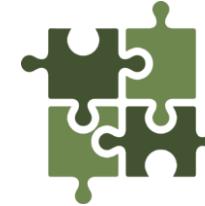
In this course, we will:

- Practice TDD in Java
- Write Java code to represent UML classes
- Apply Inheritance in Java
- Use effective programming practices
- Use a variety of classes to support different goals
- Use abstract classes and interfaces
- Refactor code to improve design while developing code
- Implement exceptions to handle unusual issues
- Effectively utilize the functional programming features of Java

# Key Deliverables



Course Notes



Project Work



There is a Skills Assessment for this course

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 1: Introduction to Java

# Chapter Overview

In this chapter, we will explore:

- Using Java classes to create instances of objects
  - Most common Java usage pattern
  - Declare, Construct, Assign, Call methods
- The existing String class
  - Reading Javadoc
  - Understanding a method's signature
  - Working with immutable objects

# Chapter Concepts

## Assumed Knowledge

---

Creating and Calling Objects

---

Using Existing Classes: String

---

Chapter Summary

---

# Assumed Knowledge

- This course focuses mainly on applied use of Java and important concepts
  - We assume knowledge of basic Java syntax
  - Appendix A covers the basics and includes suggestions for additional reading
- In particular, we assume you are familiar with these topics:
  - Program structure (e.g., all code in methods, { ... })
  - Data types (primitives vs. objects)
  - Operators (e.g., =, +, -, ==, >, ?...:, &&)
  - Control structures (e.g., if...else, switch, while, for)
  - Arrays ( [ ] )
  - Managed memory model (garbage collection)
- Java is from the C family of languages
  - Knowledge of any other language in that family will help

# Chapter Concepts

Assumed Knowledge

---

## **Creating and Calling Objects**

---

Using Existing Classes: String

---

Chapter Summary

---

# Typical Object Usage

- The following steps form the most common usage pattern in Java:
  - Declare, Construct, Assign, Call methods
  - Understand this pattern and you can understand most Java code

Naming variables after their class is a typical Java idiom

```
// DECLARE variable of specific class type
Mentor mentor;

// CONSTRUCT an object, ASSIGN it
mentor = new Mentor();

// CALL method, save result
String firstName = mentor.getFirstName();
```

# Declaration

## Java is a strongly typed language

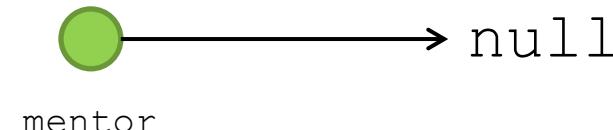
- Before any variable is used, it has to be declared
- Variable's type determines the methods you can call

```
Mentor mentor; // DECLARE variable mentor to be of type Mentor
```

## Declaration only creates a reference

- It is not yet referring to any instance of an object (garbage reference)
- Instance variables are a special case: automatically set to point to null
- If not ready to assign to an instance, better to explicitly assign reference to null

```
Mentor mentor;  
// OR  
Mentor mentor = null;
```



# Construction

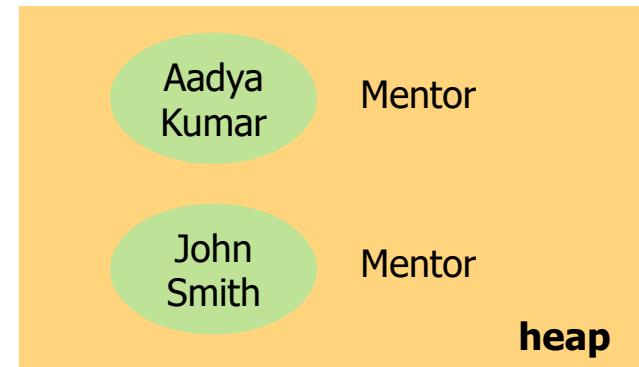
- Use `new` to explicitly create `Mentor` object in memory

```
new Mentor(); // CONSTRUCT object instance
```

- The `Mentor()` is a call to the constructor
  - Special method used to create objects of a class
  - Often can pass an initial value or leave empty
- The constructor is always preceded by the Java keyword `new`

- Constructor call returns reference to object instance
  - Creates object in memory (on the heap)

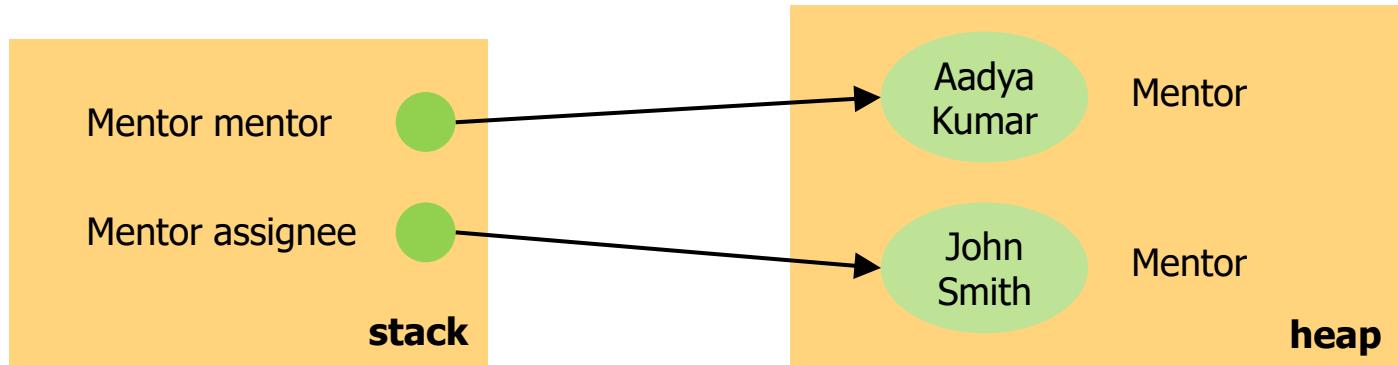
```
new Mentor("Aadya", "Kumar")
new Mentor("John", "Smith")
```



# Assignment

- Equals sign used to assign a variable to reference an object instance
  - *Note:* Java references are just pointers that cannot be manipulated directly
  - Best practice to declare variable where you can immediately assign it a value

```
Mentor mentor = new Mentor("Aadya", "Kumar");
Mentor assignee = new Mentor("John", "Smith");
```



# Usage

- Now, any methods called via reference variable act on the object in memory. For each reference and invoked function, a Stack block is created.

```
String fullName = mentor.getFullName(); // Aadya Kumar  
fullName = assignee.getFullName(); // John Smith
```

- When a new method is invoked, a new block of memory will be created in the Stack
- The returned value is stored in the Stack belonging to the invoking function
- When the method ends, that block will be erased
- The next method invoked will use that empty block
- This “last in, first out” method makes it easy to find the values needed and allows fast access to those values

# Packages

- All Java code must be contained in a package
  - The package statement must be the first line of the source file

```
package com.fidelity.objects;
```
  - If you leave this out, the code is put into the default package
- Among other things, a package acts as a namespace so that two classes with the same name won't be confused:
  - E.g., `java.awt.List` and `java.util.List`
  - To guarantee uniqueness, usually start with the reverse domain name
- We will cover packages in more detail later on
  - For now, just choose a package name that describes what you are doing
  - Get into the habit of putting all your code in packages

# Packages (continued)

- To reference a class in a different package, you must either import it or use the fully-qualified name (FQN):

```
package com.fidelity.packages;  
  
import com.fidelity.objects.Mentor;  
  
public class UseMentor {  
  
    public com.fidelity.business.Mentor copyMentor(Mentor mentor) {  
        return new com.fidelity.business.Mentor(mentor.getFullName());  
    }  
}
```

Two different classes  
with the same name

The unqualified Mentor is the one imported  
(com.fidelity.objects.Mentor)

- Try to avoid doing this for code under your control, as it just confuses people
- Classes from `java.lang` can be used anywhere without import

# Chapter Concepts

Assumed Knowledge

---

Creating and Calling Objects

---

## **Using Existing Classes: String**

---

Chapter Summary

---

# String Class Is a Special Case

- Strings are so common Java provides shorthand for:

- Creation

```
String s = "hello world";
String empty = "";
```

- Concatenation

```
String s1 = "hello";
String s2 = "world";
String s3 = s1 + " " + s2;
```

- Converting anything to string for concatenation

```
String s = "Your bill is $" + 47.5;
```

- Java does **not** support overloading operators for any other objects

# Useful String Methods

## ■ Useful String methods:

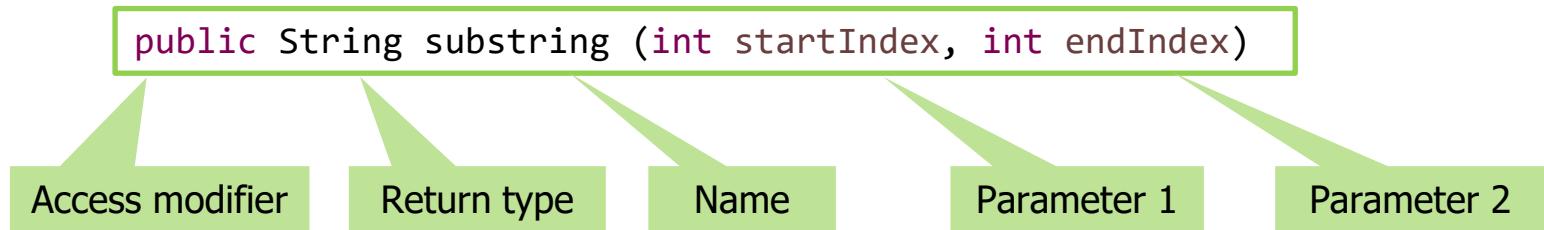
- `length()` returns number of characters in the string
- `charAt(int index)` returns character in the string at the given index, starting at 0
- `substring(int startIndex, int endIndex)` returns part of the string starting at the given index up to, but not including, given end
- `toUpperCase()` returns the string where all letters have been converted to uppercase

```
String s = new String("hello world");
int size = s.length();      // 11
char firstChar = s.charAt(0); // 'h'
String firstThreeChars = s.substring(0, 3); // "hel"
```

## ■ And many others ...

# Method Signature

- A method's signature defines its **contract** with client classes
  - Access: which classes can call it (any others, subclasses, others in its package, just itself)
  - Return type: what type is returned (type determines what client can call on the result)
  - Name: how to call the method
  - Parameters: number and types of arguments (type determines what method can call)



- Later, we will see that Exceptions are also part of a method's signature
- Since Java is statically typed, providing this information allows the compiler to verify the data passed and returned from methods will be valid
  - Necessary to discover possible errors **before** users run the code



# Try It Now: Look at String Documentation

10 min

- Examine the API documentation:

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

**Method Summary**

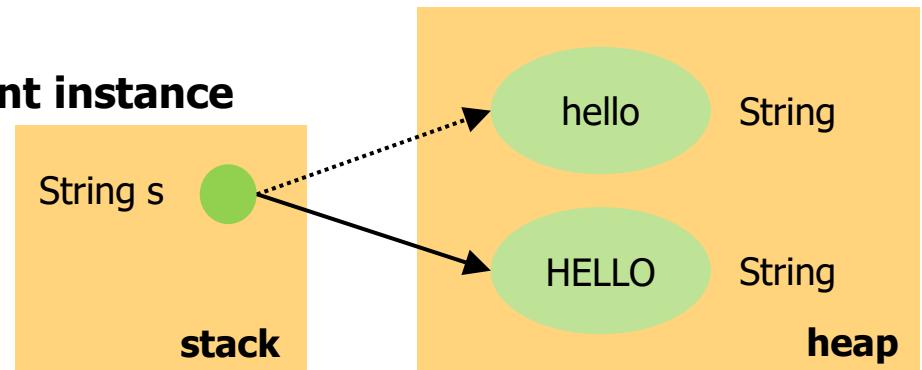
All Methods	Static Methods	Instance Methods	Concrete Methods	Deprecated Methods
Modifier and Type	Method and Description			
char	<code>charAt(int index)</code> Returns the char value at the specified index.			

- What questions do you have? Raise your virtual hand to join the discussion.

# Strings are Immutable

- String's characters cannot be modified after it has been created
  - **No methods** can change the object's fields
  - Its fields are not directly accessible or only copies of that field are
- This makes it safer for you to share `String` values
  - Also, more efficient and thread-safe for Java
  - Many of Java's standard classes are immutable
- So how to "change" it?
  - Assign result back to the original variable
  - Actually, assigns `s` to reference a **different instance**

```
String s = new String("hello");
// no effect
s.toUpperCase();
// now s references different instance
s = s.toUpperCase();
```





# Exercise 1.1: Using String Methods (Optional)

20 min

- If you would like some practice with `String`, or Java methods:
  - Please refer to your Exercise Manual to complete this exercise
- Use TDD—what possible strings would make good tests?
- Use JavaDoc—what `String` methods would be useful?
- Use Git—how often should you commit? Push?

# Chapter Concepts

Assumed Knowledge

---

Creating and Calling Objects

---

Using Existing Classes: String

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Using Java classes requires creating object instances
  - Follow common Java usage pattern
  - Declared type determines what methods can be called on the variable
  - All Java classes must be in a package
- The existing String class
  - Reading Javadoc
  - Understanding a method's signature
  - Working with immutable objects
- *Note:* Basic Java syntax can be found in Appendix A

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 2: Test-Driven Development

# Chapter Overview

In this chapter, we will explore:

- A Recap of TDD
  - Provides a rhythm for developing code that makes it more manageable
  - Write tests before writing code
- How to use JUnit
  - Annotations: choose what methods are part of the test
  - Assertions: choose what values to check as part of the test
  - Test Fixtures: ease creation of tests and reduce duplicated code
- How to test after finding a bug
  - Again, write the test before fixing the bug
- Debugging is a common and necessary part of programming
  - Often harder than creating the code itself
  - Requires a different set of skills
  - Debugger is a powerful tool to help you

# Chapter Concepts

## TDD with JUnit

---

Writing Tests

---

Test Fixtures

---

More JUnit

---

Debugging

---

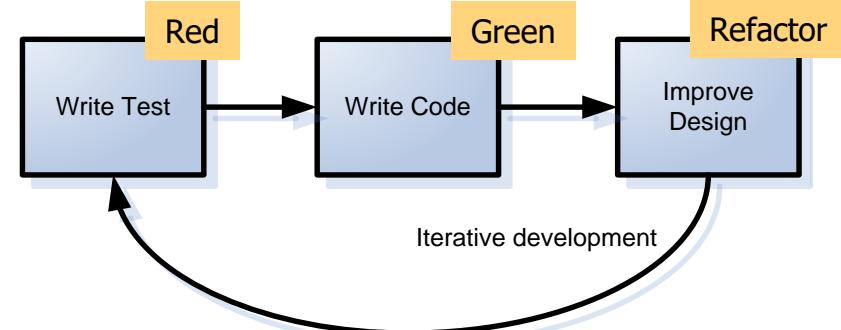
Chapter Summary

---

# Recap: Steps in TDD

■ Use these steps as a **rhythm** for code development

1. Write a test
  - Nails down “public face” of the class
  - Class, library tends to be easy to use
  - At this point, the test fails (“red”)
2. Implement code
  - Start off with a simple internal design for the code
  - Simplest possible implementation to get the test to pass (“green”)
3. Improve design without introducing new behavior
  - Called refactoring
  - Make sure that tests continue to pass, so no new bugs are introduced



TDD = “Only write code to fix a failing test”

# What Is JUnit?

- JUnit is a framework to aid unit testing
  - Developed by Erich Gamma and Kent Beck
  - Forms the basis of unit testing frameworks in many other languages
  - A set of classes (UI elements also built to add support in Eclipse)
    - Simply add appropriate jar to classpath of application or as a dependency in Maven
- Many consider unit tests to be the single most important testing tool
  - Checks a single method or a set of cooperating methods
  - Tests in isolation, not the complete program
  - For each test, you provide a simple class that provides
    - Parameters to the methods being tested
    - Expected results from those methods

<http://www.junit.org/>

# Java Annotations

## ■ Java provides **annotations**

- Can be used by Java code or external tools to mark code to act upon
- Meta information about the code
- Annotations start with @

## ■ For example, Java uses annotation @Deprecated to marks a method as obsolete

- Method still runs when called
- But Eclipse strikes through it as a hint to developers

```
String s = "hello";
s.getBytes(0, 10, result, 0);
```

## ■ @Test annotation signals to JUnit that this method should be run as a test

- Historically, methods also tend to start with the word “test” (but do not have to)

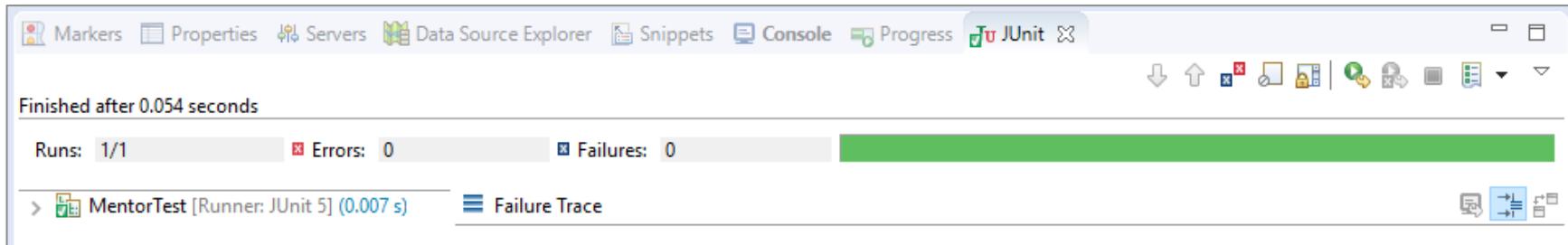
# Writing Specific Tests

- Best practice to write a separate test class for each class to be tested
  - The test class is a plain Java object with methods that have the @Test annotation
  - This test class is called a Test Case
  - Put the Test Case in the same package as the class being tested
- Best practice for each method in Test Case to apply just one test
  - Use long names that describe what the method is testing
  - Actual test is an assertion that result should match expected value
  - A test method with no assertion passes by default (bad practice!)

```
@Test
void testFullName() {
    String expected = "Jane Doe";
    Mentor mentor = new Mentor("Jane", "Doe");
    String actual = mentor.getFullName();
    assertEquals(expected, actual, "Full name should be Jane Doe");
}
```

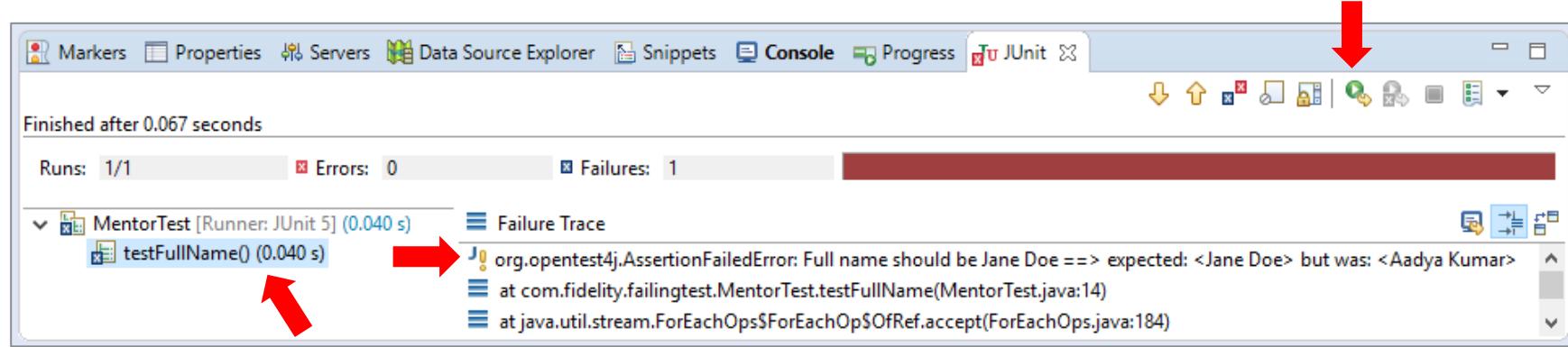
# Running JUnit from Eclipse

- Add build path dependency (use one of these)
  - Have Eclipse automatically add appropriate JUnit version
  - **Maven | Add Dependency** or edit pom.xml
- Right-click a class to test
  - **New | Other | Java | JUnit Test Case**
  - If using Maven, put the test case in `src/test/java`
- Right-click test class you have made
  - **Run As | JUnit Test**



# Other Useful Eclipse JUnit Features

- Re-run the JUnit test
  - Click the green play in the JUnit menu or right-click a specific test



- Can provide messages that display in the Runner
  - Click directly on tests to see where they failed
- Right-click project or package to re-run all tests in that scope

# Chapter Concepts

TDD with JUnit

---

## **Writing Tests**

---

Test Fixtures

---

More JUnit

---

Debugging

---

Chapter Summary

---

# Testing Strategies

- Discuss what might go wrong when introducing new feature
  - This helps generate tests and especially inputs later
  - Include positive and negative (error producing) tests
- Ensure that every feature or found bug has tests
  - It is your responsibility as a professional
- Do not test blindly: every test should have a clear purpose
- Design your code to be as easily testable as possible
- Divide what needs to be tested into different cases or categories
  - Look especially at the boundaries between cases
- Test the business logic: especially conditionals and calculations
- Do not test methods that are too simple to break (like trivial getters and setters)

# Overall Steps in Writing a Unit Test

■ The overall steps in testing involve:

- Prepare test data
  - Later, we will see how Test Doubles or Mocks can help with this
- Perform operations with system under test
- Assert state
  - Or use Test Doubles to validate behavior
- Destroy test data (and Mocks)

■ Preparing could be:

- Creating necessary objects
- Connecting to resources
- Creating files or database tables

■ Destroying could be:

- Closing files or database connections

# How Do You Check the Output Is Correct?

- Calculate correct values by hand
  - E.g., for a payroll program, compute taxes manually
- Supply test inputs that provide simple ways to get the answer
  - E.g., square root of 4 is 2 and of 100 is 10
- Verify that the output values fulfill certain properties
  - E.g., square root squared = original value
- Use a simple algorithm: slow but reliable method to compute a result for testing purposes
  - E.g., use `Math.pow` to calculate  $x^{1/2}$
  - Do **not** simply rewrite the code

# Different Assertion Types

## It is a good practice to describe what you are testing in your assertion

- `assertNotNull(Object actual, String message)`
- `assertTrue(boolean condition, String message)`
- `assertFalse(boolean condition, String message)`
- `assertEquals(Object expected, Object actual, String message)`
- `assertEquals(double expected, double actual, double delta, String message)`

Expected value

Computed value

Tolerance allowed

This message will be part of  
JUnit report if this test fails

## Assertions are typically imported statically so you do not have to type the class `Assertions`

```
import static org.junit.jupiter.api.Assertions.*
```

## API documentation:

<https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/org/junit/jupiter/api/Assertions.html>

# Write Test Logic First

- By writing the test before implementing, we get red error squiggles

The screenshot shows a Java code editor in Eclipse. A tooltip is displayed over the line `String actual = mentor.getFullName();`. The tooltip contains the message "The method getFullName() is undefined for the type Mentor" and two quick fix options: "Create method 'getFullName()' in type 'Mentor'" and "Add cast to 'mentor'". A green callout bubble points to the tooltip with the text "Eclipse offers to do the work for you". Another green callout bubble points to the cursor position with the text "Hover here". A green arrow points from the text "Click here" to the cursor position.

```
1 package com.fidelity.simpletest;
2
3+import static org.junit.jupiter.api.Assertions.*;
4
5
6
7 class MentorTest {
8
9     @Test
10    void testFullName() {
11        String expected = "Jane Doe";
12        Mentor mentor = new Mentor("Jane", Doe");
13        String actual = mentor.getFullName();
14        assertEquals(expected,
15    }
16
17 }
```

- But, more importantly, writing the tests first is part of the design process
  - What parameters should be passed? What returned?

# Positive vs. Negative Testing

- Developers find it easy to produce positive tests, since that describes what they want the code to do
  - However, that ignores how the system behaves under illegal input, which is just as important
- Positive testing is the type of testing that can be performed on the system by providing the **valid data as input**
- Negative testing is a variant of testing that can be performed on the system by providing **invalid data as input**
  - For example:
    - Testing for 0 (zero) when using as a divisor
    - Testing for negative numbers when calculating a square root
    - Testing for empty string when a value is required



HANDS-ON  
EXERCISE

40 min

## Exercise 2.1: Practicing TDD

- Start by following the instructor
  - Then complete this exercise described in the Exercise Manual
- Use the TDD rhythm
  - Write the test first; run it and make sure it fails (red)
  - Write only write enough code to make test pass, no more
  - Run the test again (green)
  - Repeat
- Use Eclipse—see what code Eclipse can generate for you
- Use your brain—think about what kinds of tests to make to check each line you wrote

# Chapter Concepts

TDD with JUnit

---

Writing Tests

---

## Test Fixtures

---

More JUnit

---

Debugging

---

Chapter Summary

---

# Don't Repeat Yourself

- Currently, we may be creating the same instances for each test
  - Just because we are writing tests is no reason to abandon good coding practices
- Objects that are created to run tests against are called a **Test Fixture**

```
@Test  
public void testSimple() {  
    EmailGenerator g = new EmailGenerator();  
    assertEquals("doe.jane@fidelity.com",  
                g.makeEmailFromName("Jane Doe"));  
}  
  
@Test  
public void testExtraMiddleSpaces() {  
    EmailGenerator g = new EmailGenerator();  
    assertEquals("doe.jane@fidelity.com",  
                g.makeEmailFromName("Jane    Doe"));  
}
```

# Test Fixtures: @BeforeEach and @AfterEach

- Fields are shared between tests
  - Where to set up?
- Mark initialization method with `@BeforeEach` to run before each test to create common objects or simple resources
- Mark disposal method with `@AfterEach` to run after each test completes to close resources
- Historically, these methods are named `setUp()` and `tearDown()`

```
private EmailGenerator g;

@BeforeEach
public void setUp() {
    g = new EmailGenerator();
}

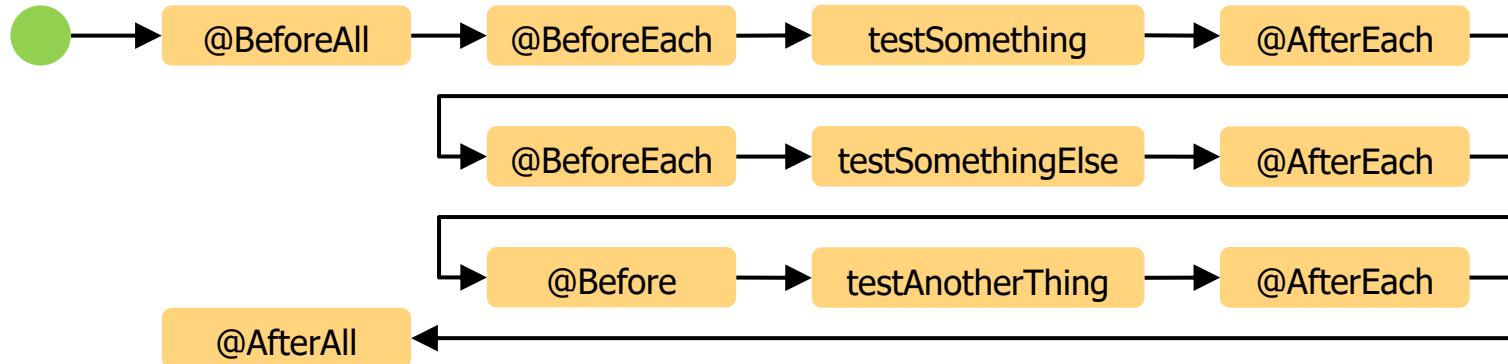
@AfterEach
public void tearDown() {
    g = null; // Not needed in this case
}
@Test
public void testSimple() {
    assertEquals("doe.jane@fidelity.com",
                g.makeEmailFromName("Jane Doe"));
}
```

# Class-Wide Setup

- Sometimes, need to do a common setup **once** for **all** tests
- Mark class-wide initialization method `@BeforeAll` to run once before any tests are run to open expensive resources like database connections
- Mark class-wide disposal method `@AfterAll` to run once after all tests have completed to close resources

# Order of Testing Methods

- In order that different tests do not interfere with each other:
  - @BeforeAll and @AfterAll methods are each called **once**
  - @BeforeEach and @AfterEach methods are called before and after **every** test
- Order in which tests are run is **not** guaranteed





HANDS-ON  
EXERCISE

## Exercise 2.2: Refactoring Test Fixtures

20 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

TDD with JUnit

---

Writing Tests

---

Test Fixtures

---

## More JUnit

---

Debugging

---

Chapter Summary

---

# TDD: Fixing Bugs

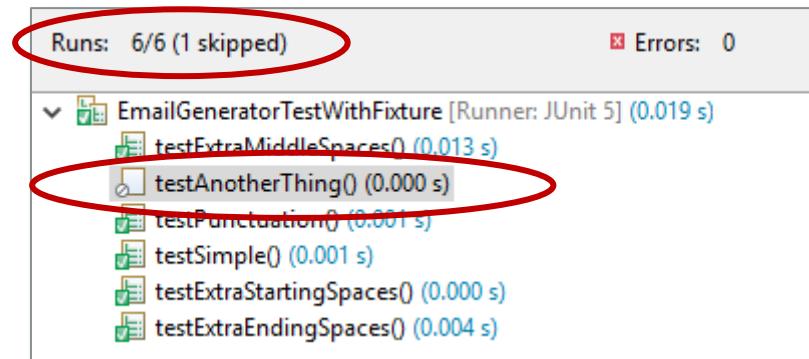
- What should happen to get the bug fixed in production?
  - Do not rush a fix into production
  - Use the same TDD process
  - Until a test is written that fails, cannot prove that the “fix” works
- First, write a test that captures the situation, and fails (“red”)
  - Review the code and data to identify the cause of the bug
- Then, fix the code implementation to get a “green”
  - This proves the fix caused the test to pass

# Disabling JUnit Tests

- Can mark a test as being disabled

```
@Test  
@Disabled("Not yet implemented")  
public void testAnotherThing() {  
    fail("Not yet implemented");  
}
```

- Unlike simply commenting out the test, the disabled test will be reported



# Regression Testing

- Always run all tests created for your project (e.g., before merging with master)
  - Detects new code that breaks existing tests
  - Automate so it is easy to include in process
- With Eclipse, you can run all tests in project or package directly:
  - Right-click package or project
    - **Run As | JUnit Test**
- With Maven, you can run all tests in project:
  - Right-click project
    - **Run As | Maven Test** (runs all tests in the project)
  - Or the equivalent command: `mvn test`

# Running Tests Under Maven

- Right-click project
  - Run As | Maven test

```
[INFO] -----  
[INFO] T E S T S  
[INFO] -----  
[INFO] Running com.fidelity.failingtest.MentorTest  
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.02 s <<< FAILURE! - in com.fidelity.failingtest.MentorTest  
[ERROR] testFullName  Time elapsed: 0.017 s <<< FAILURE!  
org.opentest4j.AssertionFailedError: Full name should be Jane Doe ==> expected: <Jane Doe> but was: <Aadya Kumar>  
at com.fidelity.failingtest.MentorTest.testFullName(MentorTest.java:14)  
  
[INFO] Running com.fidelity.simpletest.MentorTest  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in com.fidelity.simpletest.MentorTest  
[INFO]  
[INFO] Results:  
[INFO]  
[ERROR] Failures:  
[ERROR]   MentorTest.testFullName:14 Full name should be Jane Doe ==> expected: <Jane Doe> but was: <Aadya Kumar>  
[INFO]  
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0  
[INFO]  
[INFO] -----  
[INFO] BUILD FAILURE  
...  
...
```

# JUnit 4

- JUnit 4 has been in use for more than 10 years, so there is an established code base
- Significant differences:
  - Project dependencies are much simpler (in Maven or imported directly)
  - @Before, @After, @BeforeClass, @AfterClass
  - assertTrue(String message, boolean condition)
  - @Ignore (not @Disabled as used in JUnit 5)

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

4.12 is latest

# Test Suites

- If you need more control, you can create Test Suites in JUnit 4
- JUnit Test Suites support multiple Test Cases
  - Still write one test class per class
  - Combine appropriate combination in suite
- By default, Maven will not run Test Suites
  - This is usually the appropriate behavior
  - Maven test runner is called Surefire
  - Older versions (such as the default) include \*Test, Test\*, & \*TestCase
  - To include Test Suites, either update the Surefire plugin version or change the includes (or both)
  - <http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>

```
@RunWith(Suite.class)
@SuiteClasses({
    TestFeature1.class,
    TestFeature2.class,
    // add other classes as needed
})
public class FeatureTestSuite {
    // class is empty, used only
    // as holder for annotations
}
```

# Chapter Concepts

TDD with JUnit

---

Writing Tests

---

Test Fixtures

---

More JUnit

---

## Debugging

---

Chapter Summary

---

# Debugging Advice in Quotes

- If debugging is the process of removing software bugs, then programming must be the process of putting them in. —Edsger Dijkstra
- Irreproducible bugs become highly reproducible right after delivery to the customer.  
—Michael Stahl's derivative of Murphy's Law
- I don't care if it works on your machine! We are not shipping your machine! —Vidu Platon
- The wages of sin is debugging. —Ron Jeffries
- It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free. —Steve McConnell
- When debugging, novices insert corrective code; experts remove defective code.  
—Richard Pattis
- More than the act of testing, the act of designing tests is one of the best bug preventers known. —Boris Beizer

# Debugging Strategies

- Figure out the exact steps to reproduce the bug reliably (write yourself a bug report)
  - This helps establish the context of the bug
- Write in small chunks to minimize the impact of new code (JUnit can help with this)
  - Focus on the last thing you changed
- Talk it out with someone else (they do not even have to understand the problem)
  - Describing the code often requires you to look at it more closely
- Challenge your assumptions about how the code should work
- Use the scientific method: develop a hypothesis and find actual data to back it up
- Get up and walk around, think about something else, even sleep on it
  - Debugging code is more mentally demanding than writing code!
- Keep a journal of your bugs (and review occasionally to look for common issues, patterns)

# What Is a Debugger?

- Program that runs your program to help you analyze its run-time behavior
  - The larger your program is, the harder it is to debug simply by print statements
  - Lets you stop and restart your program, step through it, and see contents of variables
- Key features
  - Breakpoints: stop execution at a specific line of code
  - Single-stepping: step one line of code, either into a method or just getting its value
  - Inspecting variables: see the values of all variables and all fields of a class
- Eclipse debugger has more useful features
  - Watchpoints: stop execution if a field is read or modified or reaches a specific value
  - Tracepoints: print out message without halting the program at specific points of interest
  - Edit variables: **change** the value of any variable at the current debugging position
  - Display View: **evaluate code** within the context of the current debugging position
  - Hotswap Fixing: **edit code** and resume running at the current debugging position

# Debugging in Eclipse

- Debug Perspective changes the layout of screen and give access to other views
  - Instead of Run As..., choose Debug As...



- You can resume to next break point, stop, or step through or over the code



# Debugging Code

- Toggle breakpoints by double-clicking line number area, or right-click (use menu)

The screenshot shows a Java debugger interface. On the left, a code editor displays a snippet of Java code with line numbers 15 through 19. Line 16 is highlighted in green and contains a breakpoint marker (a green diamond). The code itself is:

```
15     double incomeTotal = 0;
16     for (int year = 0; year < numYears; year += 1) {
17         incomeTotal += getYearlyPay(hourlyWage, vacationDays);
18         hourlyWage *= (1 + percentRaise);
19     }
```

A callout arrow points from the line 16 breakpoint to a table on the right, which lists variable values. The table has two columns: Name and Value.

Name	Value
↳ no method return value	
↳ this	IncomeCalculator03Test
↳ hourlyWage	18.25
↳ vacationDays	12
↳ percentRaise	0.0
↳ numYears	1
↳ incomeTotal	0.0

- Different views for program stack, variable values, break points

The screenshot shows the stack trace for the suspended thread [main]. The stack consists of six frames:

- IncomeCalculator03.getCareerPay(double, int, double, int) line: 16
- IncomeCalculator03Test.testOriginalOneYear() line: 44
- NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: 62
- NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
- DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43



HANDS-ON  
EXERCISE

## Exercise 2.3: Debugging Derby

20 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

TDD with JUnit

---

Writing Tests

---

Test Fixtures

---

More JUnit

---

Debugging

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- A Recap of TDD
  - Provides a rhythm for developing code that makes it more manageable
  - Write tests before writing code
- How to use JUnit
  - Annotations: choose what methods are part of the test
  - Assertions: choose what values to check as part of the test
  - Test Fixtures: ease creation of tests and reduce duplicated code
- How to test after finding a bug
  - Again, write the test before fixing the bug
- Debugging is a common and necessary part of programming
  - Often harder than creating the code itself
  - Requires a different set of skills
  - Debugger is a powerful tool to help you

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 3: Implementing Classes

# Chapter Overview

In this chapter, we will explore:

- Writing Java code to represent UML classes
  - Store state by implementing instance fields
  - Implement constructors and other methods
  - Restrict access to objects, use getters/setters
- Implementing class behavior using methods
  - Methods can use any instance field
  - Make coupling between classes clear by using parameters and return values

# Chapter Concepts

## Constructors, Getters, and Setters

---

Implementing Behavior with Methods

---

Chapter Summary

---

# Custom Classes Follow from Class Diagram

- Member Variables/Fields
- Constructors
- Other Methods

Mentor
-firstName: String
-lastName: String
+calculatePay(): double
+getFullName(): String

```
public class Mentor {  
    private String firstName;  
    private String lastName;  
  
    // Public Methods  
    public double calculatePay() {  
        double pay = 0;  
        // perform pay calculation  
        return pay;  
    }  
  
    // Constructors  
  
    // Getters and setters  
}
```

# Fields Implement Attributes

- Fields are variables that belong to individual object instances
  - Each instance has its own copy of all field variables
  - Any field not explicitly set is initialized to 0, null, or false
- Encapsulation: fields should be `private`
  - Restricts when variable's value can be changed, making code easier to understand
  - Forces users of the class to use methods to access fields
  - Allows you to change implementation without affecting users' code
- Scope: any method of the class can access all fields

```
public class Mentor {  
    private String firstName;  
    private String lastName;
```

# Constructors Initialize Object's Fields

- Constructor is named after class and should **only** initialize fields
  - Extra code reduces object's flexibility, potentially increases coupling
  - Object should be fully initialized after constructor finishes (beware extra `init` methods)
- Use `this.` to disambiguate variables

```
public class Mentor {  
    private String firstName;  
    private String lastName;  
  
    // Constructors  
    public Mentor(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    ...  
}  
  
// somewhere else  
Mentor mentor = new Mentor("Jane", "Doe");
```

# Overloaded Constructors

- You may want to provide many ways of constructing objects for user convenience
  - The constructors all have same name (the class name)
  - But different types/numbers of parameters
  - Often used for providing default values

```
// Constructors
public Mentor(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

public Mentor(String firstName, String lastName, int id) {
    this(firstName, lastName);
    this.id = id;
}
```

For maintainability, have one constructor call another (must be first statement in method)

# Getter and Setter Methods

## Once initialized

- **Getter** return object's state
- **Setter** change object's state
- State is not necessarily a field

## Fields stay **private**

```
public class Mentor {  
    // Fields (aka instance variables)  
    // Public Methods  
    // Constructors  
  
    // Getters and setters  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    // etc  
}
```

# May Enhance or Challenge Encapsulation

## Enhance

- Preferred over direct access to fields
  - Allow you to change implementation later (e.g., name, type, internal representation)
  - Allow you to choose exact kind of access supported
- Setter methods should **validate** inputs
  - Think about the expectations of this field by other methods

## Challenge

- May create coupling between classes
  - Add **only** when needed
- Prefer immutable objects
  - Inherently thread-safe
- Fields with only a getter are read-only
  - Getters may still allow object's state to be mutated. How?

# Java Beans

- Many frameworks require Java Beans
  - Such as Spring and MyBatis
- A Java Bean is simply a class that satisfies certain conventions
  - Getter and setter names expected to be present in specific format
  - Properties defined by getters and setters, not presence of a field
  - No-arg constructor (some frameworks can handle initializing constructors as well)
  - Technically should implement Serializable (more on this later), but many don't

```
public class Mentor {  
    private String firstName;  
    private String lastName;  
  
    // Getters and setters  
    public String getFirstName()  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getFullName() {  
        return firstName + " " + lastName;  
    }  
}
```

Defines a field called "firstName"

Defines a read/write property called "firstName"

Defines a read-only property called "fullName"



# Exercise 3.1: Implementing the Model

15 min

- Please refer to your Exercise Manual to complete this exercise
- Use Eclipse—generate as much code as you can rather than write it yourself
- Use TDD—how would you test a constructor? Should you test a getter/setter?
- Use Git—how often should you commit? Push?

# Chapter Concepts

Constructors, Getters, and Setters

---

## Implementing Behavior with Methods

---

Chapter Summary

---

# Methods Implement Behavior

- Besides getters/setters, create methods to implement **behavior**
  - Can act upon the data of the object without exposing it
- For example: how to calculate pay?
  - Do not want to expose internal data and make every using class have to calculate

```
// from Mentor class
public double calculatePay() {
    return dayRate * numberOfDays;
}
```

- Every piece of data or functionality that is needed has to be available from:
  - A parameter to the method
  - A field of the class

# Overloaded Methods

- For convenience, may want to provide multiple signatures
  - For example: most often, the time of visit is the current time
  - Can make it easier on users by providing many ways to call method
- All methods have same name
  - But different types (or number) of parameters

```
// from Mentor class
public double calculatePay() {
    return calculatePay(0);
}

public double calculatePay(double bonus) {
    return dayRate * numberOfDays + bonus;
}
```

- For maintenance reasons, preferable to have one generic method
  - And all other overloaded methods call that one

# Single Responsibility Principle

- How to write code that follows the Single Responsibility Principle?
  - Most methods in a class should access most fields in the class
  - Let others create classes and pass them to you
  - Prefer to keep classes and methods short, doing only “one” thing
- Consider a Message class that supports email, SMS, tweets, Skype, etc.
  - Although this seems like a useful generalization, the class is doing too much
    - Each type of message has different restrictions, different content permitted
    - Some message types can be sent by different protocols (smtp, pop, etc.)
  - So, the class has two different reasons to change
- **Do not be afraid to create small classes or methods**



## Exercise 3.2: Adding Behavior

20 min

- Please refer to your Exercise Manual to complete this exercise
- Use Eclipse—what features help you understand what variables are available?
- Use TDD—how can you design your method to be testable?
- Use Git—how often should you commit? Push?

# Chapter Concepts

Constructors, Getters, and Setters

---

Implementing Behavior with Methods

---

## Chapter Summary

---

# Chapter Summary

In this chapter, we have explored:

- Writing Java code to represent UML classes
  - Store state by implementing instance fields
  - Implement constructors and other methods
  - Restrict access to objects, use getters/setters
- Implementing class behavior using methods
  - Methods can use any instance field
  - Make coupling between classes clear by using parameters and return values

# Key Points

- Program defensively to ensure code appropriately implements our designs
  - Encapsulation: keep fields private
  - Coupling: keep constructors simple and pass other objects in as parameters
  - Cohesion: keep methods short and focused on the class fields
- Overload constructors and methods judiciously
  - Most commonly to provide default values
  - Call a general version to prevent duplicating code
- Think before implementing getters and especially setters
  - Do others really need access to this state or is it better to add behavior to the class?

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 4: Implementing Polymorphism

# Chapter Overview

In this chapter, we will explore:

- Applying inheritance in Java
  - Inheritance can lead to reusable code
  - The `extends` keyword is used to represent inheritance
- Implementing inheritance effectively
  - How should you organize your code within your hierarchy?
  - Support SOLID principles

# Chapter Concepts

## Inheritance

---

Guidelines

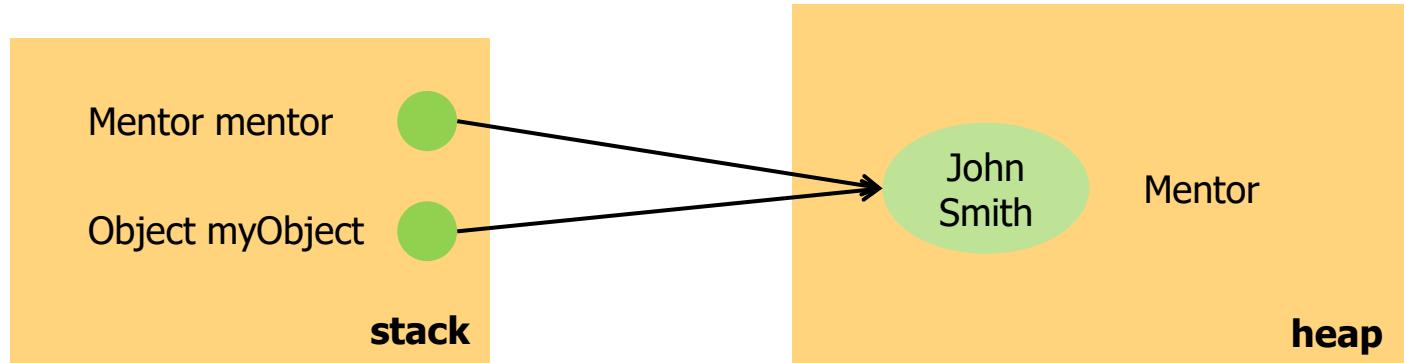
---

Chapter Summary

---

# Polymorphism Review

- Any variable with the type of `Object` can reference any Java object (IS-A)
- Variable's type determines **which** methods you can call
  - An `Object` reference, even if pointing at a `String`, **cannot** call `String` methods
  - Java requires types on declaration and construction because they **can be different**
- The method **invoked** depends on which object instance in memory executes the call
  - Called *polymorphism*



# Upcasting

- Normally, one declares and creates an object of the same type

```
Mentor mentor = new Mentor("John", "Smith");
```

- Can use a super type instead
  - And assign to it a subclass object
  - Have it reference a different object later on

```
Object anything = new Mentor("John", "Smith");
// sometime later
anything = "Hello World";
```

- Referring to a subclass object with a superclass reference is called upcasting
  - The superclass reference is more general, more reusable
  - Always safe: no special syntax required

# What Can You Do with an Upcast Reference?

- You can make methods more flexible by receiving a super, more abstract, type
- What data types can be passed into this method?

```
public void exampleMethod (Number numericValue) {  
    // can call only methods defined by Number  
}
```

- Look at documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html>
- What methods can be called on the passed object inside this method?

# Inheritance Implements Polymorphism

- The `extends` keyword signifies inheritance in Java

```
public class FullTimeMentor extends Mentor {
```

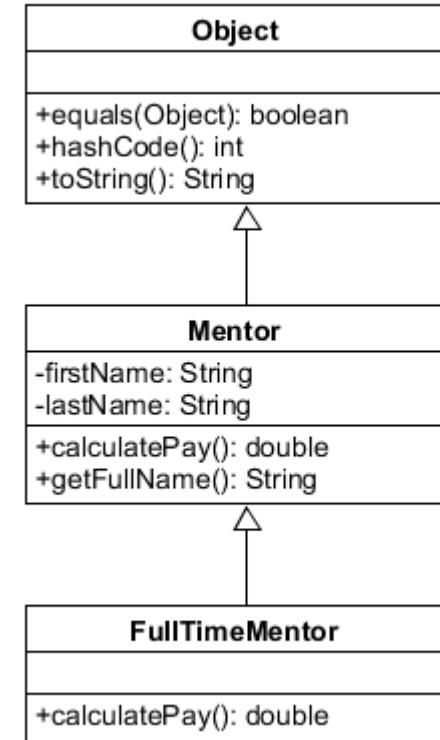
- By default, the superclass is `java.lang.Object`
  - Even if you don't write it explicitly!

```
public class Mentor extends Object {
```



```
public class Mentor {
```

- These two forms are exactly equivalent (the second is typical)



# Constructing Subclasses

- Subclasses must define their **own** constructors
- The `super` keyword is used to call superclass constructor
  - Must be **first** line of constructor; superclass must be constructed **before** subclass
  - Best practice to use `super` rather than initializing superclass variables yourself

```
public class FullTimeMentor extends Mentor {  
    private double payPerWeek;  
  
    public FullTimeMentor(String firstName, String lastName, double payPerWeek) {  
        super(firstName, lastName);  
        this.payPerWeek = payPerWeek;  
    }  
}
```

- If call to `super()` is omitted, compiler will insert call to no-args constructor
  - If this does not exist, code cannot be compiled

# Inherited Methods and Fields

- Subclasses inherit **all** methods and fields from the superclass
  - Duplicating any field declaration creates a completely separate field from the superclass
  - Only duplicate method declarations in subclass if you plan to change the implementation
- Subclasses can access methods and fields if:
  - They are `public` or `protected` (even if in different package)
  - They are `package` (default) and in same package
- Users can call the superclass methods on the subclass object
  - As if it was defined and implemented in the subclass

```
@Test
void testFullName() {
    String expected = "Jane Doe";
    Mentor mentor = new FullTimeMentor("Jane", "Doe", 42.42);
    assertEquals(expected, mentor.getFullName(), "Full name should be Jane Doe");
}
```

# @Override Annotation

- `@Override` is another annotation
  - Compilation error if method does not have the same signature as a superclass method
    - Makes typos identify themselves!
  - Prevents subtle maintenance problems later on
  - Makes code easier to understand because it makes intentions clear

```
// From FullTimeMentor class
@Override
public double calculatePay() {
    return 0.0;
}
```

- Can be automatically generated by Eclipse



# Exercise 4.1: Creating an Inheritance Hierarchy

30 min

- Please refer to your Exercise Manual to complete this exercise

# Reusing Superclass Code

- Overriding replaces the entire method
  - Use `super` to call superclass implementation of the method
  - Can be called anywhere within the subclass method (unlike using `super` in constructor)
  - Be careful, forgetting `super.` before method call results in infinite recursion!

```
// In a hypothetical subclass of Mentor
@Override
public String getFullName() {
    return super.getFullName() + " (" + getId() + ")";
}
```

# **final Means Not to be Changed**

- Field is a constant
  - Once set, it cannot be assigned again
- Method cannot be overridden
- Class cannot be subclassed

```
public final class AccountOwner {  
    private final String name;  
    private final boolean taxable;  
  
    public AccountOwner (String name, boolean taxable) {  
        this.name = name;  
        this.taxable = taxable;  
    }  
  
    public final String getName () {  
        return name;  
    }  
    public final boolean isTaxable () {  
        return taxable;  
    }  
    // No setter methods: (why?)  
}
```

# Downcasting: Syntax

- If you want to call methods defined in the subclass, must use a downcast
- Special syntax needed because downcasts could fail if wrong type
  - Perform a sanity check by using `instanceof`

```
public void exampleDowncastMethod(Mentor mentor) {  
    if (mentor instanceof FullTimeMentor) {  
        FullTimeMentor ftm = (FullTimeMentor) mentor;  
        // now call subclass specific methods  
    }  
    // work with generic Mentor methods here  
}
```

- ***Try to avoid this type of programming***
  - *It is very difficult to maintain or extend*

# Chapter Concepts

Inheritance

---

**Guidelines**

---

Chapter Summary

---

# Open Closed Principle

- How to write code that follows the Open Closed Principle?
  - Open: create class hierarchies that represent an abstraction, designed to be extended
  - Closed: write methods that accept the most general superclass you can
- New subclasses, new functionality, can be added without changing this code
  - Working with a mentor's pay, should not matter what type of mentor

```
public void processPay(Mentor mentor) {  
    double pay = mentor.calculatePay();  
    // process pay as required  
}
```

- Beware of downcasting; it violates this principle

# Liskov Substitution Principle

- How to write code that follows the Liskov Substitution Principle?
  - Use classes based on behaviors, not on state
  - Enumerate class invariants whenever possible, especially in superclasses
  - Check if your subclass can pass tests designed for its superclasses
- Subclasses can be written by anyone to do anything (you are now part of a team)
  - May override a method to do nothing, change the return value, or even throw an error
  - May change superclass state in unexpected ways
- Methods rely on using references to superclasses without knowing exact subclass
  - Superclass: encapsulate yourself specifically for subclassing as needed
  - Subclass: study superclass expectations so you do not violate them
- **Beware of “unimplementing” a method; it violates this principle!**

# Super or Subclass?

- Where to put your code within an abstraction (a class hierarchy)<sup>1</sup>
  - Commonality: collect common, duplicated, code into the superclass (its interface)
  - Variability: distribute distinct code into appropriate subclasses (its secret)
- Design classes for extension
  - Use `protected` access modifier: creates subclass specific API
  - Implement `public` methods in terms of `protected` methods to support flexibility
  - Provide constructors that initialize any field subclass may want to specify
- **Usually, a bad idea to duplicate fields in subclasses**
  - **Generally, always a good idea to make fields private**

1. D.L. Parnas, "On the Criteria to be Used in Decomposing a System into Modules," *Comm. ACM*, 1972

# Chapter Concepts

Inheritance

---

Guidelines

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Applying inheritance in Java
  - Inheritance can lead to reusable code
  - The `extends` keyword is used to represent inheritance
- Implementing inheritance effectively
  - How should you organize your code within your hierarchy?
  - Support SOLID principles

# Key Points

- Inheritance can lead to reusable code
  - Duplication: use superclasses to factor out common code among subclasses
  - Abstraction: write methods that accept the most general class you can
- Eliminate duplicated code within, as well as, across classes
  - Only declare instance variables once within a class hierarchy
- How you write your code affects whether your design is implemented well
  - Your code should clearly communicate its purpose to the casual observer
  - The details of how your classes relate to each other determine if your code is:
    - Readable
    - Maintainable
    - Testable

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 5: Effective Programming

# Chapter Overview

In this chapter, we will explore:

- Java's inheritance hierarchy
  - Every class you write is part of it
- Java's standard methods
  - Know how and when to override these methods
  - Make your class fit in better with Java classes, especially collections
  - Understand why some methods should not be overridden

# Chapter Concepts

## Object's Standard Methods

---

equals

---

hashCode

---

toString

---

Methods to Treat Cautiously

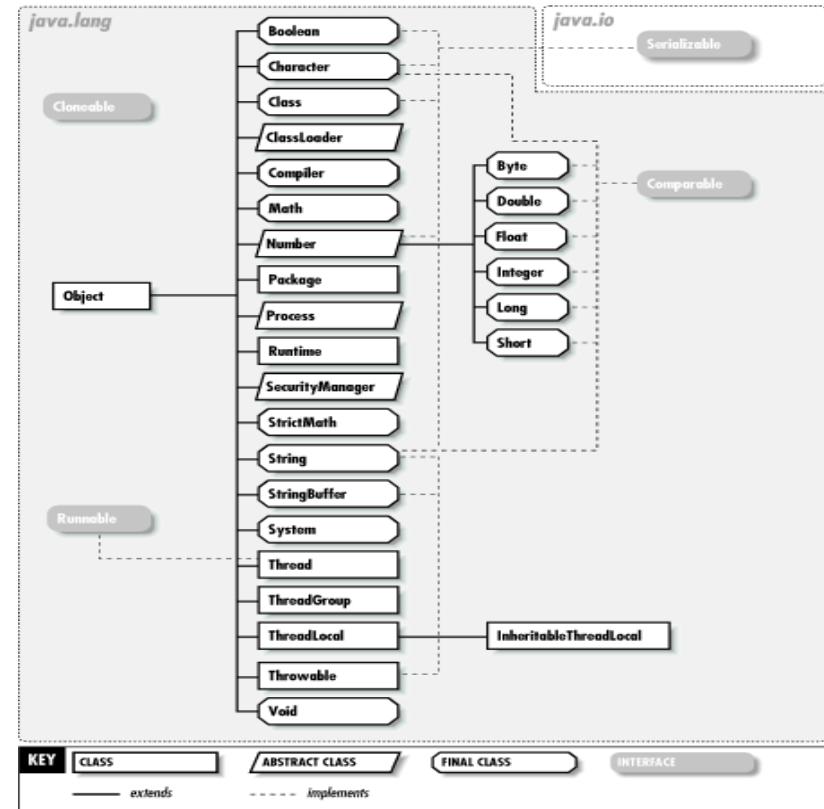
---

Chapter Summary

---

# Every Object is-a Object

- Any data type that is not a primitive is an object data type
- Every class extends from the `Object` class
- All objects inherit default method implementations
  - Subclasses can override inherited methods



# Object: The Ultimate Superclass

- The superclass of every object data type in Java is `java.lang.Object`
  - Designed for extension
- Whenever you are writing a new class:
  - These methods typically need to be overridden:
    - `equals()`
    - `hashCode()`
    - `toString()`
  - Two protected methods should rarely be overridden:
    - `finalize()`
    - `clone()`
  - Two key interfaces need to be considered:
    - `Serializable`
    - `Comparable`

<b>java.lang.Object</b>
<ul style="list-style-type: none"><li>+ <code>equals(obj: Object): boolean</code></li><li>+ <code>getClass(): Class</code></li><li>+ <code>hashCode(): int</code></li><li>+ <code>notify()</code></li><li>+ <code>notifyAll()</code></li><li>+ <code>toString(): String</code></li><li>+ <code>wait()</code></li><li>+ <code>wait(timeout: long)</code></li><li>+ <code>wait(timeout: long, nanos: int)</code></li></ul>

<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

# Useful Methods of Object

## ■ equals ()

- Returns boolean, compares if two objects are equal
- By default, compares if object references are pointing at the same object (same as ==)
- **Does not compare values**

## ■ hashCode ()

- Returns int, intended to digest fields stored in an instance into a single value
- Default value is dependent on the JavaVM implementation
- If two objects are equal, they should have the **same** hash code
- Useful for quickly finding objects in collections

## ■ toString ()

- Returns String, human readable version of the object
- By default, returns the class name and the hexadecimal version of the hash code
- Makes it possible to convert any object to a String for printing out

# Chapter Concepts

Object's Standard Methods

---

**equals**

---

hashCode

---

toString

---

Methods to Treat Cautiously

---

Chapter Summary

---

# An `equals()` Method Should Be Simple

- An `equals()` method should just test fields for inequality
  - The `String` object should not do case-insensitive equality checks
  - The `File` object should not consider symbolic links as identical objects
- An `equals()` method should not rely on unreliable resources
  - Example of mistaken design: `java.net.URL` does IP address lookup if asked to compare a host name with an IP address
    - Requires network access
    - Has problems if DNS-mapping changed while program is running

# Checking Strings for Equality

- Two ways to see if two `String`s have the same characters in the same order:

- `equals(String other)`
- `equalsIgnoreCase(String other)`

```
String s1 = ... ;  
String s2 = ... ;  
if (s1.equals(s2)) {  
    // do something if strings are value equal  
}
```

- Both these methods understand internationalization
  - Understand special characters and how to uppercase them

# When `equals()` Need Not Be Overridden

- The `equals()` method is widely used to compare objects for equality
  - Many search/find methods rely on `equals()`
  - Correct behavior is necessary for class to work with most libraries
  - The default implementation checks only for identity, rarely useful
- Do not have to override `equals()` **only if:**
  - Class is limited to specific number of instances
    - Singletons, enums, etc.
  - Class does not represent a value object
    - Services, for example, do not have to override `equals()`
  - Superclass has overridden `equals()`
    - And this subclass adds no extra state
  - You are confident that `equals()` method will never be invoked
    - Safer to override and throw `UnsupportedOperationException` in such a case

# Requirements of an equals () Method

■ A good equals () method should be:

- Reflexive
  - Object must be equal to itself
- Symmetric
  - If a.equals (b) is true, then b.equals (a) should be true
- Transitive
  - If a.equals (b) and b.equals (c) are true, then a.equals (c) should be true
- Consistent
  - The return value cannot change if objects are unchanged
- Able to handle null as a parameter

■ Possible to inadvertently break symmetry/transitivity

# Symmetry and Transitivity

- Symmetry and transitivity can be broken easily

```
public class Point {  
    private int x, y;  
    // constructors  
  
    public boolean equals (Object o) {  
        if (o instanceof Point) {  
            Point p = (Point)o;  
            return p.x == x && p.y == y;  
        }  
        return false;  
    }  
    // other methods
```

```
public class Point3D extends Point {  
    private int z;  
    // constructors  
  
    public boolean equals (Object o) {  
        if (o instanceof Point3D) {  
            Point3D p = (Point3D)o;  
            return super.equals(o) &&  
                p.z == z;  
        }  
        return false;  
    }  
    // other methods
```

- What's the problem with this code? Raise your virtual hand to share your thoughts.

# Solution: Symmetry

- The problem is that Point3D is an instanceof Point
  - But Point is not an instanceof Point3D (not symmetric)
  - The fix is to check the Class, rather than use instanceof:

```
public boolean equals (Object o) {  
    if (o != null && o.getClass().equals(this.getClass())) {  
        Point p = (Point)o;  
        return p.x == x && p.y == y;  
    }  
    return false;  
}
```

- But in a larger sense, the problem is that Point3D should not extend Point
  - Could use delegation to allow reuse while keeping relationship private

```
public class Point3D {  
    private Point xy;  
    private int z;  
    // etc.
```

# Chapter Concepts

## Object's Standard Methods

---

equals

---

**hashCode**

---

toString

---

## Methods to Treat Cautiously

---

## Chapter Summary

---

# When to Override hashCode()

- The hashCode method is used by collections such as HashSet and HashMap
  - Value objects will, at some point, be used as keys into a set or map
  - Program will not function properly if hashCode is incorrect
- Override hashCode() whenever you override equals()
  - Equal objects should have equal hash codes
  - The default implementation relies on object identity (the default equals behavior)
- The hashCode should try to spread objects across many hash buckets
  - No requirement that unequal objects must have different hash codes
  - **But equal objects must have equal hash codes**

# Chapter Concepts

## Object's Standard Methods

---

equals

---

hashCode

---

**toString**

---

## Methods to Treat Cautiously

---

## Chapter Summary

---

# The `toString()` Method

- Consider providing a `toString()` method for your classes
  - Makes them easy to display in a readable format
  - Consider whether `toString()` is for display or for debugging
    - Try to provide all the information in the object in that context
    - If impractical to return everything, return summary
  - If the format is fixed, provide static factory method to instantiate object from its output
    - Usually not a good idea, since it limits future development
    - Make sure to document whether the format is fixed or not
- If you provide some information in a `toString()` method:
  - You should provide access to that information in a getter method
  - **Do not force programmers to parse your string representation**



# Exercise 5.1: Using Eclipse to Generate Methods

15 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

## Object's Standard Methods

---

equals

---

hashCode

---

toString

---

## Methods to Treat Cautiously

---

## Chapter Summary

---

# Cloneable Is Broken

- The Cloneable interface controls the behavior of Object's clone() method
  - If a class implements Cloneable, clone() returns a field-by-field copy
  - Otherwise, clone() throws CloneNotSupportedException
- However, there are several issues with Cloneable and clone():
  - Object declares method as protected
    - So it cannot easily be invoked by client code
    - Cloneable interface does not do much good for the class's clients
  - Incompatible with final fields that refer to mutable objects
    - Not a constructor, cannot assign new value to field
  - clone() has to add information to object returned by super.clone()
    - Works only if superclass took the effort to implement clone()
- Provide a different way for clients to make copy of an object
  - A constructor that does field-by-field copying
  - A static factory method that does field-by-field copying

# Avoid Finalizers

- All objects have a `finalize()` method that can be overridden
  - Method is called by garbage collector (GC) before object is collected
  - Purpose was to clean up some specific types of resources
- The `finalize()` method should be avoided because finalizers:
  - Cause erratic behavior: no guarantee if or when GC happens
  - Can cause poor performance: GC can cause application to “pause”
  - Cause portability problems: GCs vary from platform to platform
- Nothing time-critical should be done by a finalizer
  - No guarantee of when it will be invoked
- Never depend on a finalizer to update critical persistent state
  - Possible that finalizer never gets invoked
  - Exceptions thrown from finalizers are silently ignored

# Termination Method

- To clean up resources, provide an explicit termination method
  - Typical names: `close()`, `dispose()`, etc.
  - Client code must call `close()` method once it is done with the object
  - Class should protect itself if it is invoked when it is no longer valid

```
public class Exhibit {  
    private Connection conn;  
    public Exhibit ( ... ) { conn = ... ; }  
    public void dispose () {  
        conn.close();  
        conn = null;  
    }  
    public void incrementVisitorCount () {  
        if (conn != null) {  
            ...  
        }  
    }  
}
```

# Chapter Concepts

## Object's Standard Methods

---

equals

---

hashCode

---

toString

---

## Methods to Treat Cautiously

---

## Chapter Summary

---

# Chapter Summary

In this chapter, we have explored:

- Java's inheritance hierarchy
  - Every class you write is part of it
- Java's standard methods
  - Know how and when to override these methods
  - Make your class fit in better with Java classes, especially collections
  - Understand why some methods should not be overridden

# Key Points

- Overriding `Object` methods determines how well your class fits with other Java classes
- Whenever you expect an object to be used within a collection:
  - Implement `equals()` to check values rather than identity
  - Implement `hashCode()` such that it returns the same value for objects that are equal
- Think about your primary purpose for writing `toString()`
  - Used by Debugger to represent an object
  - Used to output object to console or a log
  - Used to represent object to the user via browser or GUI

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 6: Useful Classes

# Chapter Overview

In this chapter, we will explore:

- Existing Java classes: BigDecimal, Dates, and Time
  - Reading Javadoc
  - Understanding a method's signature
  - Working with immutable objects
- Why dealing with real numbers is difficult
  - Limited precision makes it impossible for any language to represent infinite possible values
  - Use class representation to avoid compounding simple numeric errors

# Chapter Concepts

**BigDecimal**

---

Date and Time Classes

---

Chapter Summary

---

# When to Use Float and Double

- Computations using float and double are not exact
  - Must provide approximations over an **infinite** range of numbers
  - float uses less memory/space and is faster
  - double provides greater accuracy

```
double x1 = 0.3;
double x2 = 0.1 + 0.1 + 0.1;
if (x1 == x2) {
    // FALSE! This code will never run because x2 = 0.30000000000000004
}
```

- If range of numbers is limited and you need great performance:
  - Do all computations using int or long
  - Manually scale and unscale the numbers
    - For example, multiply floats by 100,000 and round-off to nearest integer

# Using BigDecimal

- These approximations are not appropriate in many business situations
  - Require exact answers—a fraction of a penny adds up!
- BigDecimal can be used instead (note, it is *immutable*)
  - Provides precise rounding methods (legally mandated)
  - Supports more than 18 digits
    - Can represent 9-digit numbers using `int` and 18-digit numbers using `long`

```
BigDecimal a1 = new BigDecimal("1.09"); // preferred for double values
BigDecimal a2 = new BigDecimal(50);
// immutable, so remember to store the results of any calculation
BigDecimal a3 = a1.multiply(a2);
// scale based on the given power of ten and given mode
BigDecimal a4 = a3.setScale(0, RoundingMode.HALF_EVEN);
```

- API documentation:

<https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>

# Comparing Real Numbers

- These approximation issues make comparing real numbers hard
  - You cannot assume two real numbers will ever be **exactly** the same
  - Typically, provide a tolerance: how close do both numbers need to be
- Do **not** write these comparisons yourself: use Java's built-in versions

```
double d1 = -0.00001;
if (Double.compare(d1, 0.0) < 0) {
    // do something if d1 is negative
}

BigDecimal val1 = new BigDecimal(0.1);    // approximation
BigDecimal val2 = new BigDecimal("0.1"); // exact
if (val1.compareTo(val2) == 0) {
    // do something if they represent the same value
}
```

# Exercise 6.1: BigDecimal Practice



20 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

BigDecimal

---

## Date and Time Classes

---

Chapter Summary

---

# Date and Time

- Java 8 introduced new Date and Time classes in the `java.time` package
  - Fixed many issues in the original `java.util` version of the classes  
<http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html>
- Primary classes (note, all are *immutable*)
  - `LocalDate`
  - `LocalTime`
  - `LocalDateTime`
  - `ZonedDateTime`
  - `Instant`
  - `Period`
  - `Duration`
- API documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>

# LocalTime

- Represents a time without a date
  - Hour, minute, second, to nanosecond precision

```
LocalTime now = LocalTime.now();      // current time
LocalTime noon = LocalTime.NOON;      // 12:00pm
LocalTime midAfternoon = LocalTime.of(15, 30, 42);    // 15:30:42
// 12345th second of day (03:25:45)
LocalTime fromSecondsOfDay = LocalTime.ofSecondOfDay(12345);
```

# LocalDate

- Represents a date without time
  - Year, month, day

```
// current date
LocalDate currentDate = LocalDate.now();
// January 1, 2001
LocalDate turnOfTheCentury = LocalDate.of(2001, Month.JANUARY, 1);
// months values start at 1 (2017-08-01)
LocalDate firstAug2017 = LocalDate.of(2017, 8, 1);
// the 42nd day of 2001
LocalDate theday = LocalDate.ofYearDay(2001, 42);
```

# LocalDateTime

- Represents both a time and a date
  - Without a time zone

```
// time and date right now
LocalDateTime currentDateTime = LocalDateTime.now();
// Neil Armstrong's big step
LocalDateTime moonTime = LocalDateTime.of(1969, 7, 20, 10, 56);
// 2017-05-05 12:00
LocalDateTime mayday2017 = LocalDateTime.of(2017, Month.MAY, 5, 12, 0);
```

## ZonedDateTime

- Represents both a time and a date with a time zone

```
// 2017-02-20 12:00
LocalDateTime dateTime = LocalDateTime.of(2017, 02, 20, 12, 0);
// 2017-02-20 12:00, Europe/Copenhagen (+02:00)
ZoneId copenhagen = ZoneId.of("Europe/Copenhagen");
ZonedDateTime copenhagenDateTime = ZonedDateTime.of(dateTime, copenhagen);
// 2017-02-20 03:00, US/Pacific-New (-07:00)
ZoneId westCoast = ZoneId.of("America/Los_Angeles");
ZonedDateTime laDateTime = copenhagenDateTime.withZoneSameInstant(westCoast);
// -28800, offset between this time zone and UTC
int offsetInSeconds = laDateTime.getOffset().getTotalSeconds();
// a collection of all available zones
Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
```



## Exercise 6.2: Using Date/Time Methods

20 min

- Please refer to your Exercise Manual to complete this exercise
- Use TDD—what possible values would make good tests?
- Use JavaDoc—what `Date` and `Time` methods would be useful?
- Use Git—how often should you commit? Push?

# Chapter Concepts

BigDecimal

---

Date and Time Classes

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Existing Java classes: BigDecimal, Dates, and Time
  - Reading Javadoc
  - Understanding a method's signature
  - Working with immutable objects
- Why dealing with real numbers is difficult
  - Limited precision makes it impossible for any language to represent infinite possible values
  - Use class representation to avoid compounding simple numeric errors

# Key Points

- Many common useful Java classes are immutable to make Java more efficient, but not necessarily easier to use
- It is important to be vigilant in dealing with numeric errors
  - Use Java's built-in solutions rather than writing your own
  - BigDecimal class provides infinite precision numbers
- The Date and Time classes were updated for Java 8

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 7: Constants and Enumerations

# Chapter Overview

In this chapter, we will explore:

- Magic values are specific hard coded values within code
  - Reduce readability
  - Increase chances of errors
- Two ways of deal with magic values
  - Constants
  - Enumerated types
- How utility classes represent a group of related methods
  - Do not need to instantiate
  - Do not need instance fields

# Chapter Concepts

## Constants

---

Enumerated Types

---

Static Fields and Methods

---

Chapter Summary

---

# Named Constants

- Rather than hard coding “magic” numbers in your code, preferable to use named constant values

```
public class OverdraftAccount extends Account {  
    private static final int MAX_TIMES_OVERDRAWN = 10;  
    private int numTimesOverdrawn;  
}
```

- numTimesOverdrawn is an **instance** variable
  - Different OverdraftAccount objects have different values for this variable
  - John’s account may have been overdrawn 4 times, Susan’s may have been 2 times
- MAX\_TIMES\_OVERDRAWN is a **constant** (will always be 10 for all instances)
  - Denoted by marking the field as `final`
- MAX\_TIMES\_OVERDRAWN is a **class** variable
  - Denoted by marking the field as `static`
  - All OverdraftAccount objects have the same value for this variable
  - Efficient in this case because all instances would be guaranteed to have same value anyway

# Chapter Concepts

Constants

---

**Enumerated Types**

---

Static Fields and Methods

---

Chapter Summary

---

# The enum Keyword

- Java provides enumerated type which simplifies coding of related named constants

```
public enum PerformanceReviewResult {  
    BELOW,  
    AVERAGE,  
    ABOVE;  
}
```

- BELOW, AVERAGE, and ABOVE are treated as if static, final fields
  - Better readability than using hard coded int values
  - Better error checking than using named constant int or String values

```
emp.setPerformanceReview(PerformanceReviewResult.ABOVE);
```

# enum Is a Kind of Class

- Can have constructors, fields, and methods
- Say we want to be able to display a message to the user
- Usage does not change
  - All instances are constructed automatically

```
public enum PerformanceReviewResult {  
    BELOW("does not meet expectations"),  
    AVERAGE("meets expectations"),  
    ABOVE("exceeds expectations");  
  
    private String message;  
  
    private PerformanceReviewResult(String msg) {  
        this.message = msg;  
    }  
  
    // other methods  
    public String getMessage() {  
        return message;  
    }  
}
```

# enum Is a Kind of Class Hierarchy

- Each value can override methods
  - Define the method for the class
  - Treat the enum value as a subclass

```
public enum PerformanceReviewResult {  
    BELOW("does not meet expectations") {  
        @Override  
        public boolean isApplicable(int grade) {  
            return grade < 75;  
        }  
    },  
    // etc  
    ABOVE("exceeds expectations") {  
        @Override  
        public boolean isApplicable(int grade) {  
            return grade >= 95;  
        }  
    };  
  
    // Other methods  
    public boolean isApplicable(int grade) {  
        return false;  
    }  
}
```

# Useful enum Methods

- Enumerated type is a class, inherits some useful methods
  - `valueOf(String s)` converts between String and enumerated type
  - `toString()` provides the String name of the instance
  - `values()` returns array of all possible instances
    - Convenient way to get a collection of possible values, perhaps for UI display or checking

```
String userInput = "BELOW";
emp.setPerformanceReview(PerformanceReviewResult.valueOf(userInput));
String review = PerformanceReviewResult.BELOW.toString(); // BELOW
// check if user references any message text
for (PerformanceReviewResult r : PerformanceReviewResult.values()){
    if (r.getMessage().contains(userInput)) {
        // do something
    }
}
```



**15 min**

# Exercise 7.1: enum Practice

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

---

Constants

---

Enumerated Types

---

## Static Fields and Methods

---

Chapter Summary

---

# No Instances Required

- Math is a standard Java class that contains static fields and methods
- These class fields and methods can be used directly without creating an instance
  - Instance fields and methods require an object
- Multiple Math objects would be wasteful
  - Constants remain the same
  - Methods remain the same since they do not reference any state
  - There are no object instance values to store

```
double randomValue = Math.random();      // between 0 and 1. Ex: 0.43
double zeroToTen = Math.random() * 10;    // multiply to give range
double radius = Math.round(zeroToTen);
double circumference = 2 * Math.PI * radius;
```

- API documentation:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

# Utility Classes

- A utility class is a class that exists to group-related `static` methods
  - The class then merely provides a namespace for these methods
- Not meant to be instantiated, so make its constructor `private`

```
public class MentorUtilities {  
    private MentorUtilities() {  
        // prevent this class from being instantiated  
    }  
  
    // Other methods  
    public static void processPay(Mentor mentor) {  
        // work with generic Mentor methods here  
    }  
}
```

- Making class `abstract` or constructor `protected` does not work
  - Could still subclass and then instantiate

# Chapter Concepts

---

Constants

---

Enumerated Types

---

Static Fields and Methods

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Magic values are specific hard coded values within code
  - Reduce readability
  - Increase chances of errors
- Two ways of deal with magic values
  - Constants
  - Enumerated types
- How utility classes represent a group of related methods
  - Do not need to instantiate
  - Do not need instance fields

# Key Points

- Best practice: use named values that are verified by the compiler
  - Using strings directly means you need to worry about spelling errors
  - Using numbers means you need to check that you're getting the correct values
- Enumerated types give you added benefit over basic constants
- Utility classes should not depend on state—simply a set of useful methods

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 8: Abstract Classes and Interfaces

# Chapter Overview

In this chapter, we will explore:

- Abstract methods
  - Are methods that all concrete subclasses have to implement
  - Cannot instantiate abstract classes directly
- Abstract classes and interfaces
  - Lead to highly extensible code
  - Focus on behavior without worrying about implementation

# Chapter Concepts

## Abstract Classes

---

Interfaces

---

Serializable

---

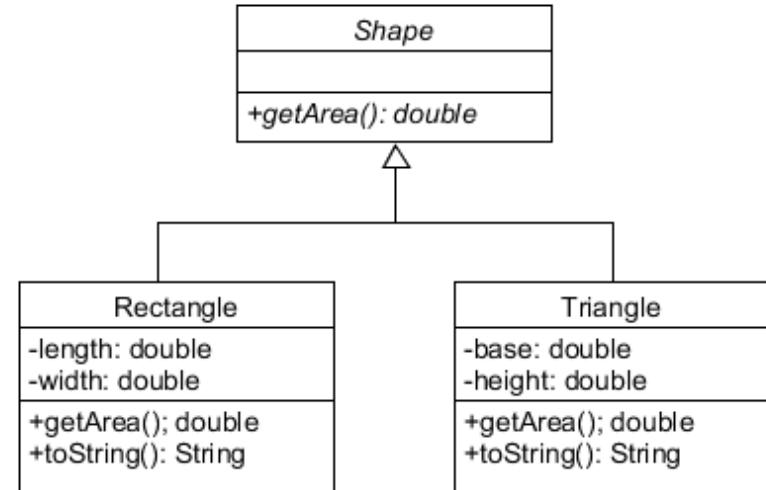
Chapter Summary

---

# Abstract Methods

- Sometimes the superclass may not know the implementation of a method
  - But needs to specify it for subclasses
  - Such methods are **abstract**
- A class with one or more abstract methods must itself be declared abstract
  - You can declare a class **abstract** if you want, even if all its methods are implemented
- You cannot create instances of an abstract class

```
// ERROR: this will NOT COMPILE  
Shape s = new Shape();
```



# Inheriting from an Abstract Class

- When subclassing an `abstract` class, you have two choices
  - Make the subclass concrete by implementing the `abstract` method
    - Syntax is just like overriding an inherited method
      - Write a method with exact same signature
      - Best practice is to also add `@Override` annotation
    - Make the subclass also `abstract` by not implementing it
  - To make a class concrete, you must implement **all** the abstract methods
    - Otherwise, subclass is itself `abstract`
  - Abstract classes can still have constructors
    - If it has fields, it should initialize those fields (don't leave this to subclasses!)
    - Automatically called by concrete subclasses when they are created

# Why Abstract Methods?

- Abstract methods define a **contract** enforced by the compiler
  - Allows superclass to define guidelines to subclasses without specifying implementation
  - Creates methods that are required to be implemented by subclass
- Abstract methods can be either `public` or `protected`
- If a new type of `Shape` is added:
  - Implementer provides `getArea()` method
  - Existing code continues to work (Open-Closed Principle)

```
// could return any subclass
Shape anyShape = getShapeSomehow();
// don't know, don't care which implementation is used
double area = anyShape.getArea();
```



# Exercise 8.1: Making abstract Superclass

10 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

Abstract Classes

---

**Interfaces**

---

Serializable

---

Chapter Summary

---

# Interfaces

- An interface is an extreme version of an abstract class
  - Only public abstract methods, no concrete methods
    - Except for static or default methods
  - Only public static final constants, no fields

```
public interface Animal {  
    void makeSound();  
    int getNumLegs();  
}
```

```
public interface Predator {  
    boolean isSuitablePrey(Animal animal);  
}
```

- Any class that implements an interface must:
  - Implement **all** of its abstract methods
  - Or declare itself as abstract

# Implementation Should Not Impact Interface

- Do not leak implementation details out of the Interface
  - Users should not depend on the implementation
  - Restrict the possibility of changing the implementation in the future
- This includes the types of exceptions that may be thrown
  - Wrap implementation specific exceptions in a custom defined exception

# Multiple Interfaces

- Interfaces are special
  - A class can **extend** only one class
  - A class may **implement** many interfaces

```
public class Lion implements Animal, Predator {  
    public void makeSound() {  
        // code here  
    }  
  
    public int getNumLegs() {  
        // code here  
    }  
  
    public int isSuitablePrey(Animal animal) {  
        // code here  
    }  
}
```

# Extending an Interface

- Consider this interface which has already been implemented by several classes

```
// example from Oracle JavaDocs
public interface TimeClient {
    void setTime (int hour, int minute, int second);

    void setDate (int day, int month, int year);

    void setDateAndTime (int day, int month, int year, int hour,
                         int minute, int second);

    LocalDateTime getLocalDateTime();
}
```

- What to do if we need to now add a new method to this interface?
  - And avoid the wrath of all the programmers whose class already implements it

# Default Methods

- Starting with Java 8, can add a default method
  - Typically defined in terms of other interface methods or static methods

```
// example from Oracle JavaDocs
public interface TimeClient {
    // methods as before

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), ZoneId.of(zoneString));
    }
}
```

- Can also be used to reduce potential duplicate code

# Prefer Interfaces to Abstract Classes

- Interfaces are far more flexible

- Client class can implement multiple interfaces and then be used in multiple contexts

```
public class SomeClient implements Interface1, Interface2 {  
    // okay  
}
```

- Abstract base classes are much more limiting

```
public class SomeClient extends AbstractClass1 {  
    // has to choose: AbstractClass1 or AbstractClass2?  
}
```

- If possible, move functionality from abstract classes into helper classes
    - Client classes that implement interface can then use those helpers

- If you have an abstract class, also provide an interface

# Storing Constants

- Do not use interfaces to store **only** constants—interfaces should be about behavior

```
public interface MuseumLimits {  
    public static final int MAX_VISITOR_COUNT = 1000;  
}  
public class SomeClient implements MuseumLimits {  
    public boolean allowVisitor() {  
        return numVisitors < MAX_VISITOR_COUNT;  
    }  
}
```

- To store constants, use most appropriate class or an enum

```
public class MuseumLimits {  
    public static final int MAX_VISITOR_COUNT = 1000;  
}  
public class SomeClient {  
    public boolean allowVisitor() {  
        return numVisitors < MuseumLimits.MAX_VISITOR_COUNT;  
    }  
}
```

# Chapter Concepts

Abstract Classes

---

Interfaces

---

**Serializable**

---

Chapter Summary

---

# Serializable

- One of the key Java interfaces
  - Implementing this indicates that objects can be persisted as a byte stream
  - The internal state of an object is persisted, the class definition is not
- It is a ***marker*** interface
  - There are no methods, but the presence of the interface indicates a capability
  - If serializability is important to a method, it can accept a Serializable parameter
- Strictly, all Java Beans should implement Serializable, but many don't
  - Why not?
  - Because many use cases have evolved beyond simple serialization (e.g., JSON)
  - However, they usually still follow serialization rules

# Serialization

- Do not implement `Serializable` unless you need to
  - It restricts future flexibility
  - Doing it well is non-trivial
- A class may be serializable if all fields are also serializable or declared `transient`
  - Note that all primitives are serializable
- Best practice to create field `SerialVersionUID`
  - Change value when class structure changes

```
public class SerialExample implements Serializable {  
  
    private static final long serialVersionUID = 3007667791011724669L;  
  
    ...  
}
```

# Serialization and Inheritance

- If a superclass implements `Serializable`, then so do all subclasses
  - This does not mean it is possible to serialize the subclass successfully
    - All fields must also be serializable or marked `transient`
- If a superclass does not implement `Serializable`:
  - Deserialization does not invoke a subclass constructor
    - But the superclass no-args constructor **will** be invoked, so it must exist
  - Superclass state will not be serialized or de-serialized unless done manually
    - Implement `readObject()` and `writeObject()` methods

# Chapter Concepts

Abstract Classes

---

Interfaces

---

Serializable

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Abstract methods
  - Are methods that all concrete subclasses have to implement
  - Cannot instantiate abstract classes directly
- Abstract classes and interfaces
  - Lead to highly extensible code
  - Focus on behavior without worrying about implementation

# Key Points

- Abstract methods implementation should not impact method signatures
- When to use an interface or an abstract class?
  - Abstract classes can contain concrete method definitions
  - Interfaces are more flexible
- Interfaces
  - Must implement certain interfaces to make use of Java standard classes
  - Some methods require that parameters passed in implement an interface

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 9: Java Collections Framework

# Chapter Overview

In this chapter, we will explore:

- How most applications must deal with collections of objects
  - Different problems are best solved with different collection types
  - Choosing the right collection type is an important implementation decision
- Java's Collection Framework
  - An excellent example of practical object orientation
  - Contains interfaces, abstract classes, and utility classes
  - Contains general methods that take interfaces rather than concrete classes
  - Uses parameterized types to make it more flexible and easier to use
- Looping over collections
  - Simplified `for-each` syntax for looping over every item in a list one by one
- How Java provides auto-boxing
  - Hides syntax needed to use primitives in lists
- Sorting using `Comparable` and `Comparator`

# Chapter Concepts

## Collections Framework

---

ArrayList

---

Using Collections

---

Primitive Wrapper Classes

---

Sorting

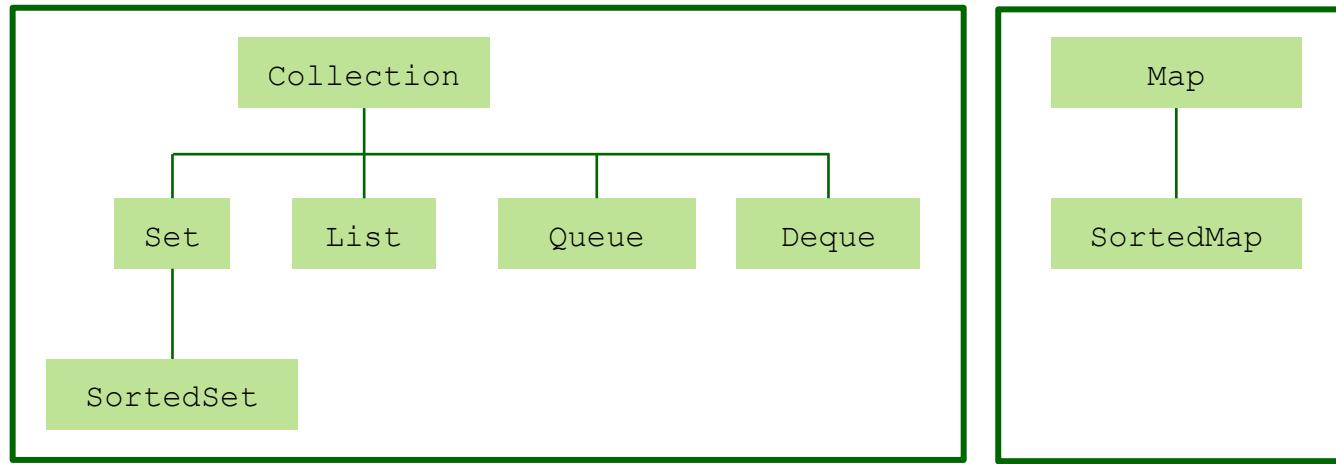
---

Chapter Summary

---

# The Collections Framework Interfaces

- Interfaces for storing and manipulating collection of items
- Each interface supplies methods to abstractly represent different data structures



- API overview:

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

# Overview of Collection Framework Interfaces

## List

- Contains items in sequential order
- Can be rearranged as needed

## Set

- Contains only unique items
- Can be used to check for existence of objects

## Queue

- Contains items to be processed in order received
- Deque allows to add or remove from front and back of container

## Map

- Contains key/value pairs
- Associates simple value with more complex one

# Overview of Collection Framework Abstract Classes

- All collection interfaces have multiple concrete implementations
  - Different data structures have different trade-offs
  - Provide abstract classes to collect common code
- Primary abstract classes (most likely you will never reference these directly)
  - AbstractCollection
  - AbstractSet
  - AbstractList
  - AbstractQueue
  - AbstractMap
- More general to use the interfaces in your code
  - Just examining these as a case study of generally well designed code

# Overview of Collection Framework Utility Classes

- Collection algorithms and other general methods grouped into utility classes
  - Collections
  - Arrays
  - Objects
- Primary static methods provided (take appropriate collection to access)
  - binarySearch
  - fill
  - max
  - min
  - reverse
  - shuffle
  - sort
- API documentation (most commonly used utility):  
<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

# Exercise 9.1: Study the Collections Classes (Optional)



20 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

Collections Framework

---

**ArrayList**

---

Using Collections

---

Primitive Wrapper Classes

---

Sorting

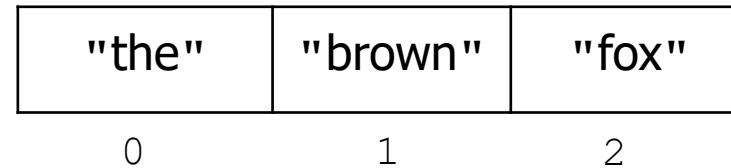
---

Chapter Summary

---

# ArrayList

- The most commonly used collection is ArrayList
  - Sequence of values of the same type
  - Stored in the order they are added
  - Grows as items are added, no need to know size beforehand



- ArrayList implements List interface
  - There are other implementations of List
  - Best practice to declare variables to be as general as possible
    - Usually means the interface type

# Creating an ArrayList

- A collection is just another object in Java
  - Can create with or without an initial size

Variable declared  
of type `List`, but  
`ArrayList`  
instantiated

```
// CREATE list containing NO items, let Java use a default size
List<String> words = new ArrayList<String>();

// CREATE list containing NO items, with an initial capacity
List<String> words = new ArrayList<String>(100);
```

- In both cases, size of collection is 0
  - If you know the expected size, it is more efficient to provide it
  - Does not affect the number of items the list can hold (unlike traditional arrays)
- API documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

# Collections Use Generic Class to Indicate <type>

- Specify the type when declaring the desired data type

```
List<String> words = new ArrayList<String>();
```

- Can use the same class with a different data type

```
List<LocalDate> dates = new ArrayList<LocalDate>();
```

- Advantages

- No need to cast; cast is automatic and implicit
- Expands the ability to reuse code
- Type issues caught at compile time

- Generics added in Java 5

- Original collection classes explicitly stored only Object references, hard to work with
- Still stored internally as Object references, notation is “syntactic sugar”

# Diamond Syntax

- Java 7 introduced a convenient shorthand

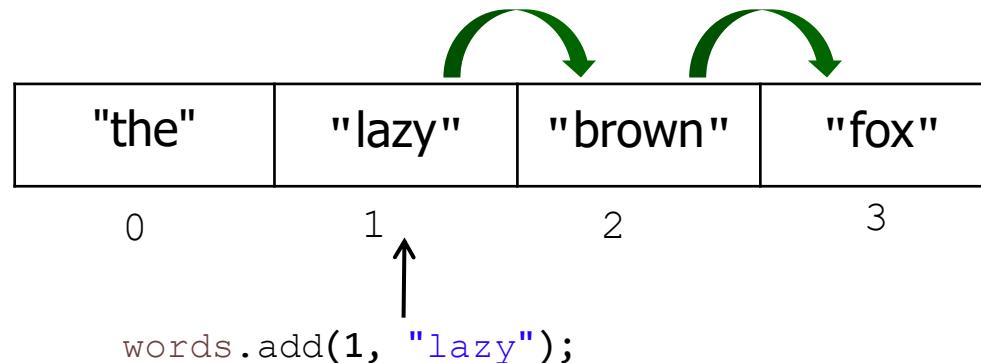
```
List<String> words = new ArrayList<>();  
List<LocalDate> dates = new ArrayList<>();
```

- Compiler can infer type in constructor based on assignment
- The diamond syntax improves readability
  - But you still may see legacy code that uses the old style (shown on the previous slides)

# ArrayList Methods

## ■ Useful methods:

- `size()` returns current number of items in the list
- `get(int i)` returns the *i*th item, starting from 0
- `add(Object o)` adds given item to the end
- `add(int index, Object o)` inserts given item at the given index
- `set(int index, Object o)` replaces item at the given index with given item
- `remove(Object o)` removes first occurrence of given item



# Chapter Concepts

Collections Framework

---

ArrayList

---

## Using Collections

---

Primitive Wrapper Classes

---

Sorting

---

Chapter Summary

---

# Generalized for-each Loop

- To access each element of a collection in order, use the `for-each` notation

```
// describe what this code does, using a specific example to make it clear
String acronym = "";
for (String w : words) {
    acronym += w.substring(0, 1).toUpperCase();
}
```

## Pros:

- Greater readability, easier to write

## Cons:

- Can only access one item at one time in order
- Cannot traverse two collections at once
- Cannot remove items
- Only accessing, not assigning

## Exercise 9.2: Looping Over ArrayList



15 min

- Please refer to your Exercise Manual to complete this exercise
- Use TDD—what possible values would make good tests?
- Use JavaDoc—what `ArrayList` methods would be useful?
- Use Git—how often should you commit? Push?

# Using Collections Generally

- All collection constructors allow for passing one collection into another to create a copy
  - For example, a `TreeSet` can be created from a `HashSet` or from a `List`

```
public Set<String> processWords(HashSet<String> words) {  
    Set<String> result = new TreeSet<>(words);  
    // do something  
    return result;  
}
```

- Simple `for-each` loop is also general: works for **any** collection

```
public void processWords(Collection<String> words) {  
    for (String w : words) {  
        // process each item, w, in turn  
    }  
}
```

# Getting Items in Collections by Index

- What if you want to go backwards or skip items?
  - Much less common case, so only use when you intend to access differently
- Random access is available through list's `get` method

```
public static void processWords(List<String> words) {  
    for (int i = words.size() - 1; i >= 0; i--) {  
        String w = words.get(i);  
        // process each item, w, in turn  
    }  
}
```

- Pros:
  - Can access anywhere in the collection
- Cons:
  - Random access is very slow on some collections
  - Not available on all collections (not available on `Set`, `Queue`, or `Map` interfaces)

# Iterator

- Iterator object is optimized for its collection (this is how for-each loop is implemented)

```
public void processWords(Collection<String> words) {  
    Iterator<String> iter = words.iterator();  
    while (iter.hasNext()) {  
        String w = iter.next();  
        // process each item, w, in turn
```

## ■ Primary methods

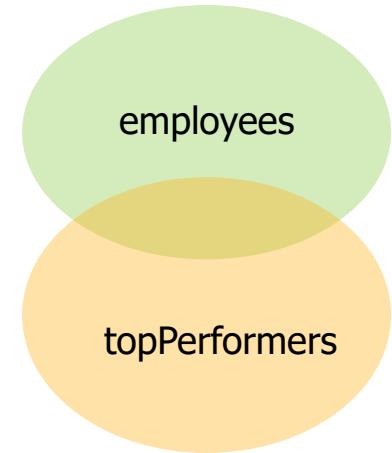
- `hasNext()` used to check there are still items to iterate over
- `next()` returns the next item
- `remove()` removes current item (this is the **safest** way to remove item from collections)
  - *Note:* some collections will throw an `UnsupportedOperationException`

## ■ Some iterators have more methods

- `ListIterator` can go backwards

# Example: Sets

- Holds unique items
  - **Uses equals() method of class**
  - Adding/removing and combining is fast
  - Searching for a particular element is slow
- Supports set theory operations from math
  - Union: `addAll()`
  - Intersection: `retainAll()`
  - Asymmetric set difference: `removeAll()`
- Different implementations
  - `HashSet`: items kept in essentially random order (**uses hashCode()**)
  - `TreeSet`: items kept in sorted order (**must implement Comparable**)
- API documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>



# Chapter Concepts

Collections Framework

---

ArrayList

---

Using Collections

---

## Primitive Wrapper Classes

---

Sorting

---

Chapter Summary

---

# Primitive Wrapper Classes

- Primitives **cannot** be used in Java collections
  - To treat primitive types as objects, you must use wrapper classes
- Most useful wrapper classes (*Note: all are immutable*)
  - Integer
  - Double
  - Character
  - Boolean

```
Integer num = new Integer(13);
num = new Integer(num.intValue() + 1);
ArrayList<Integer> values = new ArrayList<>();
values.add(num);
```

# Auto-Boxing

- Auto-boxing automates using primitive wrapper classes
  - Converts between primitive type and its corresponding wrapper class as needed

```
// this auto-boxed code is equivalent to the previous slide
Integer num = 13;
ArrayList<Integer> values = new ArrayList<>();
values.add(num + 1);
```

- Here's what Java is automating for you:
  - Auto-unbox `num` into an `int`
  - Adds 1
  - Auto-box result into a `new Integer`
  - Adds reference to newly created wrapper object in list

# Primitive Wrapper Class Features

- Can be created from Strings

```
Integer num1 = new Integer("13");
// ERROR: this does not work
num = new Integer("a");
// any other value is false
Boolean flag = new Boolean("true");
```

- Have useful constants

- Numbers: minimum and maximum value
- Characters: locale and architecture dependent settings
  - Direction of text
  - Newline character
  - Folder separation character

# Primitive Wrapper Classes Documentation

## ■ API documentation:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

<https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html>

<https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html>

## Field Summary

### Fields

#### Modifier and Type

static int

#### Field and Description

**MAX\_VALUE**

A constant holding the maximum value an int can have,  $2^{31}-1$ .



## Exercise 9.3: Primitive List Practice (Optional)

15 min

- Please refer to your Exercise Manual to complete this exercise
- Use TDD—what possible values would make good tests?
- Use JavaDoc—what `ArrayList` and wrapper class methods would be useful?
- Use Git—how often should you commit? Push?

# Chapter Concepts

Collections Framework

---

ArrayList

---

Using Collections

---

Primitive Wrapper Classes

---

**Sorting**

---

Chapter Summary

---

# Implementing the Comparable Interface

- To sort objects can use `Collections.sort(List<T> items)`
- Item's class must implement `Comparable<T>`
  - Defines object's "natural" ordering
  - *Note:* generic type `T` should match item's type
- Method to be implemented is:  

`public int compareTo(T other)`
- The expected return value indicates relative ordering between the current item and the other
  - Return any negative number if `this < other`
  - Return any positive number if `this > other`
  - Return 0 if `this.equals(other)`
- API documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

# An Example

- Standard Java value objects are typically Comparable
  - Look to reuse built in comparison methods, rather than write your own

```
public class Customer implements Comparable<Customer> {  
    private String name;  
    private LocalDate firstPurchase;  
  
    // alphabetically by name first, by earliest date if names are same  
    @Override  
    public int compareTo(Customer other) {  
        int result = name.compareTo(other.name);  
        if (result == 0) {  
            return firstPurchase.compareTo(other.firstPurchase);  
        }  
        return result;  
    }  
    // other methods
```

# Exercise 9.4: Practice Implementing Comparable



20 min

- Please refer to your Exercise Manual to complete this exercise

# Multiple Ways to Sort a List

- `sort()` is an overloaded method
- The second version takes a second parameter: how to sort the collection
  - An object that must implement `Comparator`
- Method signature:  

```
public int compare(T o1, T o2)
```
- Similar to `Comparable`, with its `compareTo()` method
  - Returns `int` to show relative ordering (negative, 0, or positive)
  - But takes two objects since it is a separate class from those being compared
    - Like `compare()` method of `Integer` in previous example
- API documentation:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>

# An Example

- Typically defined as a small, separate, class
  - Remember, comparing real numbers is complex, so reuse Java's implementation

```
//in file BalanceComparator.java
public class BalanceComparator implements Comparator<Account> {
    @Override
    public int compare(Account a1, Account a2) {
        return a1.getBalance().compareTo(a2.getBalance());
    }
}
```

- When needed, pass in an instance of the `Comparator` to `sort()` method

```
// in client class Java file
Collections.sort(accounts, new BalanceComparator());
```

# Exercise 9.5: Practice Implementing Comparator



15 min

- Please refer to your Exercise Manual to complete this exercise

# Anonymous Inner Class

- If you will not use the comparator elsewhere, can write an anonymous inner class
  - Most people prefer this to creating a separate class for each Comparator
- Keyword `new` is used with the interface, creating concrete instance where it is needed
  - Provide any abstract method implementations when defining class
  - Typically, only used for short implementations, longer code makes it hard to read

```
// in some Java file
Collections.sort(accounts, new Comparator<Account>() {
    @Override
    public int compare(Account a1, Account a2) {
        return a1.getBalance().compareTo(a2.getBalance());
    }
});
```

- Inner classes have access to methods and fields of current class
  - No need to expose `private` fields or methods

# Exercise 9.6: Anonymous Inner Class



**10 min**

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

Collections Framework

---

ArrayList

---

Using Collections

---

Primitive Wrapper Classes

---

Sorting

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- How most applications must deal with collections of objects
  - Different problems are best solved with different collection types
  - Choosing the right collection type is an important implementation decision
- Java's Collection Framework
  - An excellent example of practical object orientation
  - Contains interfaces, abstract classes, and utility classes
  - Contains general methods that take interfaces rather than concrete classes
  - Uses parameterized types to make it more flexible and easier to use
- Looping over collections
  - Simplified `for-each` syntax for looping over every item in a list one by one
- How Java provides auto-boxing
  - Hides syntax needed to use primitives in lists
- Sorting using `Comparable` and `Comparator`

# Key Points

- Most applications must deal with collections of objects
  - Different problems are best solved with different collection types
  - Choosing the right collection type is an important implementation decision
- Looping over collections
  - Simplified `for-each` syntax for looping over every item in a list one by one
- `ArrayList` is Java's most used collection
- Sorting using `Comparable` and `Comparator`
  - Anonymous inner classes can be used to implement small interfaces within your code
- Java provides object versions of primitive types
  - Have useful constants and methods, but cannot be combined like primitives
  - Auto-boxing hides syntax needed to use these classes like primitives

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 10: Refactoring

# Chapter Overview

In this chapter, we will explore:

- Refactoring to improve your design as you develop code
  - Built into the agile process
  - Built on small, clear, steps
- How refactoring requires unit tests to verify you did not introduce bugs
  - Check and balance within your coding process
  - Provides feedback and helps make process more predictable
- Eclipse to help you refactor

# Refactoring

- Remember, refactoring is focused on improving your code's design
  - Built into the agile process as you learn more, plan to handle new requirements
  - Disciplined technique made up of small, clear steps
- Like TDD, relies on rhythm to work quickly, safely
  - Test, small change, test, small change
- Refactoring is so important it is built in many IDEs, including Eclipse
  - In addition to generating basic code, it can improve your existing code!

Refactor	Navigate	Search	Sash Window
Rename...			⌘V
Move...			
Change Method Signature...			⌘C
Extract Method...			⌘M
Extract Local Variable...			⌘L
Extract Constant...			
Inline...			⌘I
Convert Local Variable to Field...			
Convert Anonymous Class to Nested...			
Move Type to New File...			
Extract Interface...			
Extract Superclass...			
Use Supertype Where Possible...			
Pull Up...			
Push Down...			
Extract Class...			
Introduce Parameter Object...			
Introduce Indirection...			
Introduce Factory...			
Introduce Parameter...			
Encapsulate Field...			
Generalize Declared Type...			
Infer Generic Type Arguments...			

# Chapter Concepts

## Duplicated Code

---

Recognizing Polymorphism

---

Dependency Inversion Principle

---

Chapter Summary

---

# Don't Repeat Yourself (DRY Principle)

- Duplicated code is worst code smell—why?
  - Intention is to reduce repetition in code and in information in general
- Extract Method is the easiest way to deal with duplicated code
  - Think about what parameters to send so that the method can serve a more general purpose
  - Think about where to put the method: current class, superclass, separate helper class
  - Think about naming the method to reveal its intention, reduce comments
- These methods are called **helper** methods and are typically private or protected

# Chapter Concepts

Duplicated Code

---

## **Recognizing Polymorphism**

---

Dependency Inversion Principle

---

Chapter Summary

---

# Recognizing the Need for Polymorphism

- Polymorphism makes your code more flexible because it supports Open-Closed Principle
  - Encouraged by Java language and compiler to provide clear paths for extension
- Often inheritance hierarchies are designed during analysis phase
  - Sometimes discovered within code
  - Look for places where you use a “type” id to distinguish between different behavior
- Replace Conditional with Polymorphism
  - Do you have a series of conditional code based on an object’s state?
  - Is the conditional’s structure duplicated (a different form of the worst code smell)?
  - Do the different conditions affect different state in the object?

# Chapter Concepts

Duplicated Code

---

Recognizing Polymorphism

---

## **Dependency Inversion Principle**

---

Chapter Summary

---

# Dependency Inversion Principle

- Coupling is the bane of reusability
  - Reuse means having to take everything it depends on as well
  - Depending on abstractions (interfaces mostly) makes code more reusable, flexible
- Depending directly on concrete class is a form of Inappropriate Intimacy code smell
- Extract Interface
  - What methods of concrete class can be generalized from the implementation?
  - Once this interface exists, are there other classes that could implement it?
  - Once this interface exists, who is responsible for creating the concrete class(es)?



**30 min**

## Exercise 10.1: Refactoring Code

- Please refer to your Exercise Manual to complete this exercise
- Use Eclipse—let refactoring menu help you rather than writing it yourself
- Use TDD—verify your tests work before and after refactoring
- Use Git—what commit messages should you use to communicate your goals?

# Chapter Concepts

Duplicated Code

---

Recognizing Polymorphism

---

Dependency Inversion Principle

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- Refactoring to improve your design as you develop code
  - Built into the agile process
  - Built on small, clear, steps
- How refactoring requires unit tests to verify you did not introduce bugs
  - Check and balance within your coding process
  - Provides feedback and helps make process more predictable
- Eclipse to help you refactor

# Key Points

- Sometimes abstractions are found during analysis
  - Sometimes abstractions are discovered as you write code
  - Relentless refactoring is required to maintain good design
- Abstractions are Java's preferred path to support extension
  - Remember, everything is an object
  - Prefer small, single-purpose objects supported by interfaces over large concrete classes
- Sometimes the most complexity in your code is in the interactions between objects

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 11: Exceptions

# Chapter Overview

In this chapter, we will explore:

- Exceptions
  - Used to handle errors gracefully
  - Interrupts the normal flow of the program
  - Allows you to handle it in a meaningful way separately
- How Java provides two different kinds of exceptions
  - Checked: compilation error if not handled or declared
  - Unchecked: recommended to handle, but can be ignored until appropriate
- How to use exceptions to validate input to business objects
  - Test for valid input and that your exceptions are properly thrown
- Custom exceptions
  - Can be created for application-specific error conditions

# Chapter Concepts

## Exceptions

---

Testing Exceptions

---

Custom Exceptions

---

Using Exceptions Effectively

---

Chapter Summary

---

# What Are Exceptions?

- Exceptions are thrown when there are issues executing code
  - Make sure the calling code is aware there was an issue
  - Gives calling code a chance to correct issue
- What lines of code might have an issue when this code executes?

```
public int parseAndCalculateResult(String arg1, String arg2) {  
    int val1 = Integer.parseInt(arg1.trim());  
    int val2 = Integer.parseInt(arg2.trim());  
    return val1 / val2;  
}
```

- Hint:* never trust string input!

# When Expecting Numerical Inputs ...

- If null values passed in: NullPointerException



## Failure Trace



java.lang.NullPointerException



at exceptions.Calculator.parseAndCalculateResult(Calculator.java:6)



at exceptions.CalculatorTest.testNullValue(CalculatorTest.java:12)

- If invalid numeric values passed in: NumberFormatException



## Failure Trace



java.lang.NumberFormatException: For input string: "abc"



at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)



at java.lang.Integer.parseInt(Integer.java:580)



at java.lang.Integer.parseInt(Integer.java:615)



at exceptions.Calculator.parseAndCalculateResult(Calculator.java:6)



at exceptions.CalculatorTest.testNonNumericValue(CalculatorTest.java:17)

# Stack Trace

- Stack trace shows complete list of methods called to get to the line that caused an error
  - Source file and line number
  - Shows chained exceptions if one exception was used to create another
- In JUnit (and Eclipse if error is unexpected), method calls are hyperlinks to the code

☰ Failure Trace

J! java.lang.ArithmetricException: / by zero

☰ at exceptions.Calculator.parseAndCalculateResult(Calculator.java:7)

☰ at exceptions.CalculatorTest.testDivideByZero(CalculatorTest.java:25)

- What can we do to prevent the program from crashing with a stack trace message?

# try and catch

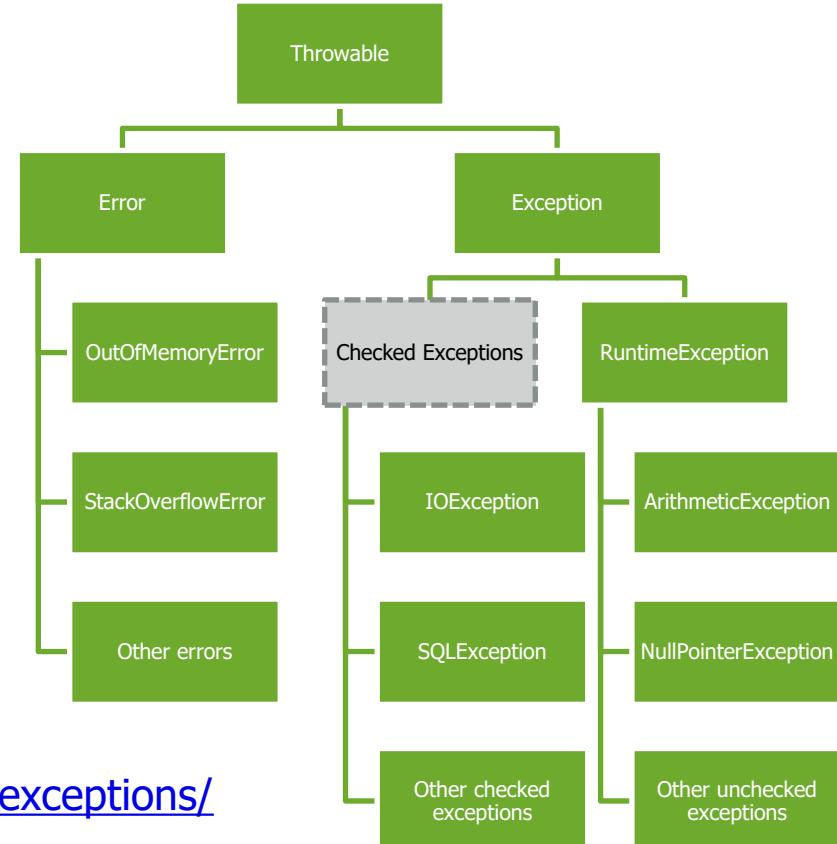
- If an exception might occur, can surround code with a try-catch block
- If an exception is thrown on any line within try block:
  - Execution immediately jumps to the catch block
  - If no exception, statements in the catch block are not executed

```
public int parseAndCalculateResult(String arg1, String arg2) {  
    try {  
        int val1 = Integer.parseInt(arg1.trim());  
        int val2 = Integer.parseInt(arg2.trim());  
        return val1 / val2;  
    } catch (Exception e) {  
        System.out.println("Invalid arguments given, cannot calculate");  
        return 0;  
    }  
}
```

What exceptions will be handled by this catch block?

# Exception Hierarchy

- Catching `Exception` catches all subclasses
  - Not recommended
- Helpful methods inherited from `Throwable`
  - `getMessage()`
    - Readable explanation of what went wrong
  - `printStackTrace()`
    - Lists complete set of calls, with line numbers that led to error
    - Clickable within Eclipse
- Documentation:  
<https://docs.oracle.com/javase/tutorial/essential/exceptions/>



# **RuntimException**

- With the code we have written so far, we were not forced to handle exceptions
- Errors due to logic flaws typically throw a `RuntimException`
  - Do not need to be declared
  - Do not need to be caught
  - If thrown, will travel up the call stack to whoever catches it
- Examples of `RuntimException`
  - Arithmetic overflow
  - Calling methods on a null pointer reference
- In the future, we will look at checked exceptions
  - Enforced by the compiler
  - Such as `SQLException`

# Multiple Catch Blocks

- Use multiple catch blocks to take specific actions
  - Put in **reverse** hierarchical order—the first matching catch block handles the exception

```
public int parseAndCalculateResult(String arg1, String arg2) {  
    try {  
        int val1 = Integer.parseInt(arg1.trim());  
        int val2 = Integer.parseInt(arg2.trim());  
        return val1 / val2;  
    } catch (NullPointerException e) {  
        System.out.println("Not enough arguments given, cannot calculate");  
        return 0;  
    } catch (NumberFormatException e) {  
        System.out.println("Invalid arguments given, cannot calculate");  
        return 0;  
    } catch (Exception e) {  
        System.out.println("Unexpected error, cannot calculate");  
        return 0;  
    }  
}
```

# Multiple Exceptions in One Catch Block

- Catch multiple specific exceptions in one block to take same action
  - Usually better to catch specific exceptions so you know what to expect

```
public int parseAndCalculateResult(String arg1, String arg2) {  
    try {  
        int val1 = Integer.parseInt(arg1.trim());  
        int val2 = Integer.parseInt(arg2.trim());  
        return val1 / val2;  
    } catch (NullPointerException | NumberFormatException | ArithmeticException e) {  
        System.out.println("Invalid arguments given, cannot calculate");  
        return 0;  
    } catch (Exception e) {  
        System.out.println("Unexpected error, cannot calculate");  
        return 0;  
    }  
}
```

# Try-Catch-Finally

- You can make sure that code executes whether or not an exception is thrown
  - Using a `finally` block, most useful for taking care of resources

```
public int parseAndCalculateResult(String arg1, String arg2) {  
    try {  
        int val1 = Integer.parseInt(arg1.trim());  
        int val2 = Integer.parseInt(arg2.trim());  
        return val1 / val2;  
    } catch (NullPointerException | NumberFormatException | ArithmeticException e) {  
        System.out.println("Invalid arguments given, cannot calculate");  
        return 0;  
    } finally {  
        System.out.println("Always executed");  
    }  
}
```



HANDS-ON  
EXERCISE

# Exercise 11.1: Catching Exceptions

20 min

- Please refer to your Exercise Manual to complete this exercise

# Ensuring Valid Data

- When constructing business objects, should never be able to exist in an invalid state
  - May come from user input so it cannot be trusted
  - May require validation that cannot be done by the database
  - May or may not be optional data, object cannot be created without it
- Must validate data passed in to constructor against the class invariants
- But, what to do if an error is detected?
  - Usually cannot continue to create the business object
  - Usually not useful to catch and handle error in the constructor

# Throwing an Exception

- Can generate a new exception by using the keyword `throw`
  - Object created after `throw` must be a subclass of `Exception` or `RuntimeException`

```
public Mentor(String firstName, String lastName, int id) {  
    if (id < 0) {  
        throw new IllegalArgumentException("id may not be negative");  
    }  
    this.id = id;  
    this.firstName = firstName;  
    this.lastName = lastName;  
}
```

IllegalArgumentException is a standard Java unchecked exception

- If you catch one type of exception, you can create new exception using the caught exception

# Check Your Setters and Avoid Duplicated Code

- Checking the constructor arguments is not enough
  - If your class has setters (i.e., mutable state), you need to check those also
  - Avoid writing the checks twice: call setters directly from your constructor

```
// Constructors
public Mentor(String firstName, String lastName, int id) {
    setId(id);
    this.firstName = firstName;
    this.lastName = lastName;
}

// Getters and setters
public void setId(int id) {
    if (id < 0) {
        throw new IllegalArgumentException("id may not be negative");
    }
    this.id = id;
}
```

# Chapter Concepts

Exceptions

---

## **Testing Exceptions**

---

Custom Exceptions

---

Using Exceptions Effectively

---

Chapter Summary

---

# Why Test for Exceptions?

- Very important that error handling receives appropriate testing
  - Many regression bugs are because of change in the way errors are handled or reported
    - Client code may expect certain behavior on certain types of inputs
    - Cannot change that behavior willy-nilly
  - Important to use tests to specify error handling
  - Use tests to maintain backward compatibility of error handling
- When testing “back-end” code, there may not be any other place that catches exceptions
  - Typically, throwing is done in the model (more errors possible)
  - Typically, catching is done in the view (closer to the user, more possibilities to handle)

# Testing If Exception Is Thrown

- Easiest way to test if an exception has been thrown is use `assertThrows()`
  - Lets JUnit know you expect this test method to throw a specific exception
  - Note the `.class` extension on the class name

```
@Test  
void testInvalidId() {  
    assertThrows(IllegalArgumentException.class, () -> {  
        new Mentor("Aadya", "Kumar", -1);  
    });  
}
```

# Writing Test for Exception Throwing

assertThrows () returns the exception

- If you want to process the exception, capture the returned value
- Say, if multiple conditions throw the same type of exception but different messages

```
@Test  
void testInvalidIdMessage() {  
    Exception e = assertThrows(IllegalArgumentException.class, () -> {  
        new Mentor("Aadya", "Kumar", -1);  
    });  
    assertEquals("id may not be negative", e.getMessage(),  
                "Expected appropriate message");  
}
```

# Testing for Exception in JUnit4

- Easiest way to test if an exception has been thrown is use parameter expected
  - No assert needed, just the setup code

```
@Test(expected=IllegalArgumentException.class)
public void testInvalidId() {
    new Mentor("Aadya", "Kumar", -1);
}
```

- And, to test the exception, use this style

```
@Test
public void testInvalidIdMessage() {
    try {
        new Mentor("Aadya", "Kumar", -1);
        fail("Should have thrown exception");
    } catch (IllegalArgumentException e) {
        assertEquals("Expected appropriate message",
                    "id may not be negative", e.getMessage());
    }
}
```



# Exercise 11.2: Throwing Exceptions

20 min

- Please refer to your Exercise Manual to complete this exercise

# Chapter Concepts

Exceptions

---

Testing Exceptions

---

## Custom Exceptions

---

Using Exceptions Effectively

---

Chapter Summary

---

# Creating a Custom Exception

- Custom exceptions are any Java class:
  - That extends either `Exception` (checked) or `RuntimeException` (unchecked)
- Often may want to catch a standard exception (such as `SQLException`)
  - And then wrap it inside an application-specific (custom) exception
  - Nesting exceptions preserves the stack trace information
- Creating a custom exception is easy, they are typically just constructors
  - All methods are inherited, nothing more to do
  - Eclipse will generate all constructors for you, so no excuse for not having them all



HANDS-ON  
EXERCISE

## Exercise 11.3: Creating a Custom Exception

20 min

- Please refer to your Exercise Manual to complete this exercise
- Use Eclipse—let it generate the custom exception class for you!
- Use Javadoc—review the generated constructors against the Javadoc of Exception

# Checked Exceptions

- Direct subclasses of Exception must be explicitly handled somehow
  - May be caught using try-catch-finally
  - Or may be thrown by the enclosing method (added to method signature)

```
public int parseAndCalculateResult(String arg1, String arg2) throws CalculationException {  
    try {  
        int val1 = Integer.parseInt(arg1.trim());  
        int val2 = Integer.parseInt(arg2.trim());  
        return val1 / val2;  
    } catch (NullPointerException e) {  
        throw new CalculationException("Not enough arguments given, cannot calculate", e);  
    } catch (NumberFormatException e) {  
        throw new CalculationException("Invalid arguments given, cannot calculate", e);  
    } catch (Exception e) {  
        throw new CalculationException("Unexpected error, cannot calculate", e);  
    }  
}
```

Unhandled checked exception  
requires throws declaration

# Chapter Concepts

Exceptions

---

Testing Exceptions

---

Custom Exceptions

---

## **Using Exceptions Effectively**

---

Chapter Summary

---

# When to Use Exceptions

- Throwing an exception should indicate a serious error condition
  - That the method cannot handle itself
  - So “throw” the problem back to the calling method
- If situation is routine, return something the caller can use
  - A boolean
  - An empty object (especially an empty collection)
  - An invalid object (perhaps a subclass of the expected type of class)
  - Even `null` may be okay (last resort since it may lead to a `NullPointerException`)
- If client code will have to do something totally different:
  - Throw an exception
  - Exception should help the calling code determine an alternative route

# Dealing with Exceptions

- Libraries throw exceptions when they expect client code can deal with the error better
  - If you don't know how to deal with exception, either:
    - Pass it straight through (or)
    - Catch it and rethrow as higher-level exception
  - If you do know how to deal with exception:
    - Add catch block for specific type of exception
    - Handle error there
  - ***Do not simply catch and ignore exceptions***
- Exceptions are meant to allow caller to centralize error handling
  - ***Do not deal with exceptions line-by-line***

# Be Specific, Throw Early, Catch Late

## ■ Be specific

- Do not throw `Throwable`, `Exception`, `RuntimeException`
- Throw a standard exception appropriate to the situation or a custom exception

## ■ Throw early

- Throw the exception as soon as the need is identified
- The stack trace will then indicate the appropriate line

## ■ Catch late

- Catch the exception when it can be handled correctly
  - As soon as possible, but not too soon
- More information is available as we go up the call stack
- Rarely makes sense to catch exceptions in the “back end” since the kind of application using it is unknown

# Core API Exceptions

## ■ Use exceptions defined by the core API whenever possible

- `IllegalArgumentException`
  - For bad parameter values
- `NullPointerException`
  - For things that should not be null
- `IllegalStateException`
  - Logical flaws
- `UnsupportedOperationException`
  - When implementing an interface, but not able to provide reasonable implementation
  - Example: `remove()` method on Iterator on read-only collections

# What to Throw

- Create custom exception class
  - Avoid throwing low-level exceptions like `IOException` from your code
- Subclass `RuntimeException`
  - To create an unchecked exception
  - Results in cleaner code
  - Many frameworks (Spring, MyBatis, etc.) use this approach
- Unchecked exceptions still should be caught and handled properly
  - Document **all** thrown exceptions
- Use checked exceptions sparingly
  - The client code is forced to handle them
    - Only use when that might be possible and desirable

# Storing Data in an Exception

- Do not expect client code to parse your exception message
  - Instead, give access to data
  - Exception object should be capable of being used programmatically

```
public class BoxOfficeNotOpen extends Exception {  
    private LocalDate nextOpenTime;  
  
    public BoxOfficeNotOpen(LocalDate nextOpenTime) {  
        super("Box office is currently closed; we'll be back at " + nextOpenTime);  
        this.nextOpenTime = nextOpenTime;  
    }  
  
    public LocalDate getNextOpenTime() {  
        return nextOpenTime;  
    }  
}
```

# Chapter Concepts

Exceptions

---

Testing Exceptions

---

Custom Exceptions

---

Using Exceptions Effectively

---

## Chapter Summary

---

# Chapter Summary

In this chapter, we have explored:

- Exceptions
  - Used to handle errors gracefully
  - Interrupts the normal flow of the program
  - Allows you to handle it in a meaningful way separately
- How Java provides two different kinds of exceptions
  - Checked: compilation error if not handled or declared
  - Unchecked: recommended to handle, but can be ignored until appropriate
- How to use exceptions to validate input to business objects
  - Test for valid input and that your exceptions are properly thrown
- Custom exceptions
  - Can be created for application-specific error conditions

# Key Points

- Exceptions are used to handle errors separately, outside the normal program flow
  - Do **not** deal with exceptions line-by-line
  - Do **not** simply catch and ignore exceptions
- Throw exceptions if you need to signal your own error
  - Create custom exceptions for application-specific error conditions (it's easy)
  - Typically, throw unchecked exceptions (easier for other programmers to use)
- Testing for exceptions is just as important as testing for correct results
  - May be the only place that catches "back-end" exceptions
  - Typically, throwing is done from the back end
  - Typically, catching is done in the front end

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 12: API Design

# Chapter Overview

In this chapter, we will explore:

- The layers of an application
  - Separate presentation (view) from data and business objects (model)
  - Layers create encapsulation, increase flexibility
- Java packages to help organize your code
- Service objects to implement business logic (controller)
  - Implement application features by combining business objects
  - Can be messy, high coupled to specific applications
- How to create APIs for your layers
  - Think through how you want other programmers to use your classes
  - Service-oriented design focuses on making reusable APIs for service objects

# Chapter Concepts

## Packages

---

Application Layers

---

API Design

---

Service Objects

---

Chapter Summary

---

# Packages: Set of Related Classes

- Packages help to organize source code
    - On file system become folders
    - Can be nested like folders (path name should match package name)
  - To put classes in a package, simply start the file with this line
    - Name consists of one or more identifiers separated by periods
    - Best practice: look like reverse URL domains, all lowercase
- `package packageName;`
- Packages form a name space
    - Prevent name clashes by putting classes in separate directories
    - For example, `java.awt.List` and `java.util.List`
  - Since related, may share implementation details
    - Package-friendly permissions (no access modifier) allows access by classes in same package
    - Not a common practice

# Importing Packages

- Can always use fully qualified class name without importing
  - Do not need to import `java.lang` explicitly

```
java.util.List<String> words = new java.util.ArrayList<>();
```

- Tedious to use
  - `import` lets you use shorter class name
  - Simply shorthand, does not actually load class (unlike some other languages)
  - Can import all classes in a package using the “`*`”, but not generally good practice

```
import java.util.*;  
// ...  
List<String> words = new ArrayList<>();
```

- You do not need to import other classes in the same package
  - Default package has no name, no package statement
  - Thus, cannot be imported by any classes outside default package

# Common Java Packages

Package	Purpose	Example Class
java.lang	Language support, most common classes, imported by default	String
java.util	Collections, internationalization, and utility classes	ArrayList
java.math	Arbitrary-precision arithmetic classes	BigDecimal
java.time	Date and time classes	LocalDate
java.sql	Database classes	ResultSet
java.net	Networking classes	URL

# Chapter Concepts

Packages

---

## Application Layers

---

API Design

---

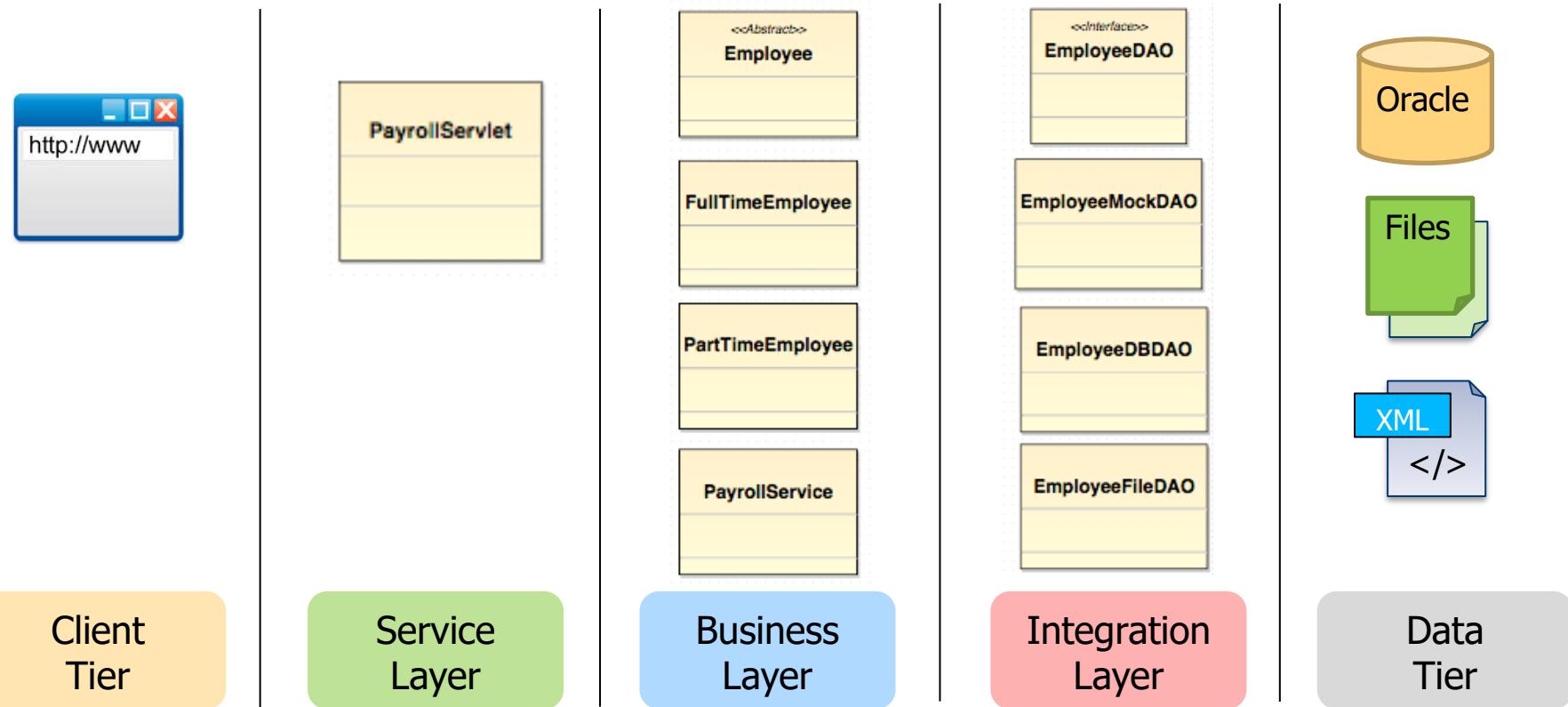
Service Objects

---

Chapter Summary

---

# An Example Application Project



# Why Create Layers?

- Layers encapsulate code's particular role and clarify design goals
  - Packages are an effective tool to represent different layers
- Create clear separations between layers by designing APIs
  - Remember how interfaces in collections framework expressed design goals
  - Abstract and concrete classes provided implementation
- API classes represent functionality and data that needs to be exchanged between layers
  - API is simply the public classes and methods you provide
  - Design principles help identify needed abstractions

# Chapter Concepts

Packages

---

Application Layers

---

**API Design**

---

Service Objects

---

Chapter Summary

---

# Why Is API Design Important?

## ■ Public API design

- Determines how other programmers (your users) will use your program
- That determines what others think of your program

## ■ Internal API design

- Good code is modular and each module has its own API
- Useful modules get reused
- Once a module is being used, changing it will cause problems

## ■ Designing the API up front can minimize others' pain

- And improve overall code quality (**both** your own and your users!)

# What Makes a Good API?

- Easy to learn
- Easy to use
  - Difficult to misuse
- Code that uses the API is easy to read
  - And easy to maintain
- Powerful enough to meet its requirements
- Easy to extend

# The API Should Do One Thing and Do It Well

- The Single Responsibility Principle applied to an API
  - The functionality should be easy to explain
  - Good descriptive names drive development

# The API Should Be as Small as Possible But No Smaller

- The API must satisfy its requirements
- When in doubt, leave it out
  - Just like recycling
  - It is easy to add to an interface, harder to remove

# Fail Fast

- For errors that cannot be handled by the API
  - Exceptions are part of your API
- Report errors as soon as possible
  - Compile time is best
    - Static typing
    - Generics
- Runtime errors
  - Report after first bad method invocation

# Implementation Should Not Impact the API

- Do not leak implementation details out of the API
  - Users of the API should not depend on the implementation
  - Restrict the possibility of changing the implementation in the future
- This includes the types of exceptions that may be thrown
  - Wrap implementation-specific exceptions in an API defined exception

# Principle of Least Astonishment

- Users of the API should not be surprised by behavior
  - May take extra implementation effort
  - May even be worth some slight reduction in performance
- Users will be much happier

# Minimize Accessibility

- Maximize information hiding
- Make classes as private as possible
  - And data members
  - And methods
- Public classes should have no public fields
  - Except for constants
- Modules can be used independently
  - Built, tested, and debugged independently

# Names Matter

- Use self-explanatory, descriptive names
  - Much easier for users to understand
  - Avoid cryptic abbreviations
- Be consistent throughout the API
  - Use the same term for the same idea
- Strive to make code readable like prose
  - Write code that is simple and understandable, not more comments

# Documentation Matters

- “Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won’t see the components reused without good documentation.”

—D. L. Parnas, Software Aging,  
*Proceedings of 16<sup>th</sup> International Conference Software Engineering, 1994*

# Chapter Concepts

Packages

---

Application Layers

---

API Design

---

## Service Objects

---

Chapter Summary

---

# Service Objects Implement Business Logic

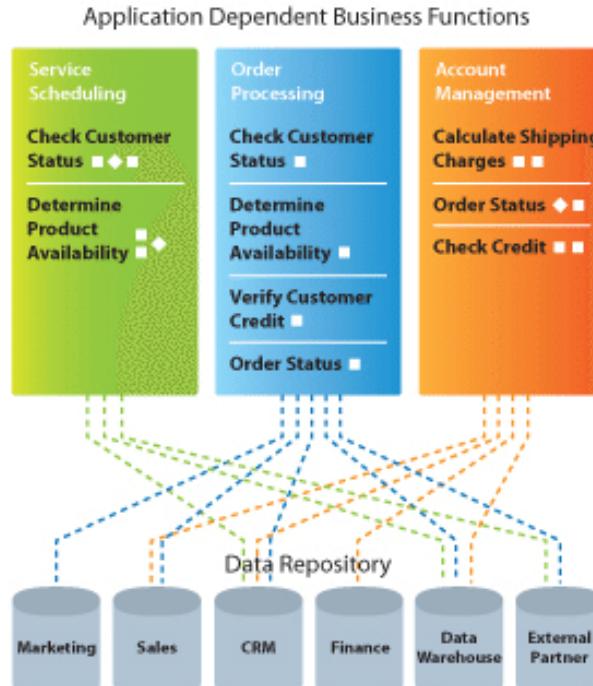
- Custom business rules often involve multiple business objects
  - Perform calculation over a collection of polymorphic objects
  - Check validity based on several different conditions
- Service objects are used to hold logic that do not belong on business objects
  - Contains knowledge of working with business objects
  - Its methods may implement the primary requirements of the application
  - It may be a utility class (if it has **no** private state)

# Service-Oriented Architecture

- Goal is to create reusable, loosely-coupled, interoperable services
  - Easy to create services that are very coupled to specific business applications
  - Especially since most services are created by different teams (no big picture)
- Applying SOLID design principles and designing each service as an API can **increase**:
  - Flexibility by making services smaller and less implementation dependent
  - Composability so services can be combined and reused to create larger applications
  - Abstraction so services can be deployed over multiple machines at multiple locations
- Applications can invoke each other only through well defined service-oriented APIs
  - APIs may be expressed in a variety of ways (REST, SOAP, etc.)
  - Web itself is an excellent example of SOA

# Before SOA

Closed - Monolithic - Brittle



# After SOA

Shared services - Collaborative - Interoperable - Integrated

Composite Applications



Reusable Business Services



Data Repository





HANDS-ON  
EXERCISE

30 min

# Exercise 12.1: Creating a Service Class

- Please refer to your Exercise Manual to complete this exercise
- Think about this class as an API
  - What package does it belong in?
  - Which methods should be public? Which private?
  - What state, if any, does it need?
- Use TDD—any different than testing business objects?

# Chapter Concepts

Packages

---

Application Layers

---

API Design

---

Service Objects

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- The layers of an application
  - Separate presentation (view) from data and business objects (model)
  - Layers create encapsulation, increase flexibility
- Java packages to help organize your code
- Service objects to implement business logic (controller)
  - Implement application features by combining business objects
  - Can be messy, high coupled to specific applications
- How to create APIs for your layers
  - Think through how you want other programmers to use your classes
  - Service-oriented design focuses on making reusable APIs for service objects

# Key Points

- Use layers (and packages) to organize your programs
- Use APIs to think about how each class is designed
  - Impacts your users (other programmers)
  - Create examples (tests) quickly and share with others to minimize changes later
  - Primary export from programs if they are used by others
- Apply good design practices to all levels of your program
  - From the smallest class to the largest program, every layer has an API
  - Try to minimize leakage of implementation details from one layer to another

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

# Chapter 13: Implementing Design Patterns

# Chapter Overview

In this chapter, we will explore:

- The role of design patterns
- Design patterns in Java

# Chapter Concepts

## Using Design Patterns

---

Examples of Design Patterns in Java

---

Factories: Creating Objects

---

Refactoring to Design Patterns

---

Chapter Summary

---

# Using Design Patterns

- So, you know Design Patterns
  - You not only own the GoF book, you have read it!
  - You know the difference between Strategy and Template, and when to use them!
  - So, you should be a pretty good software designer, right?
- What could possibly go wrong?
- "... patterns don't guarantee anything. They don't even make benefit likely. Patterns do nothing to remove the human from the creative process."  
—John Vlissides (one member of the Gang of Four)

# Beware of Over-Engineering

- Over-engineering
  - Making code overly flexible
  - Making code overly sophisticated (i.e., complicated)
- Often done to accommodate future needs
  - Because “you know it will be needed in the future”
- Results in code that is difficult to understand
  - And maintain
- Remember YAGNI!

# You Ain't Gonna Need It (YAGNI)

- The YAGNI principle
  - You Ain't Gonna Need It
- Principle in extreme programming
  - Don't add any functionality until you absolutely need it
  - Prevents "Feature Creep"
- "Always implement things when you actually need them, never when you just foresee that you need them." —Ron Jeffries (XP co-founder)

# Beware of Under-Engineering

- More common than over-engineering
- What are the causes?
  - Not enough time to do it right
  - Must add new features to an existing system that we don't fully understand
  - Unclear on good software design principles
- Results in poorly designed software
  - Difficult to maintain or extend
- “Data structures may be haphazardly constructed, or even next to non-existent. Everything talks to everything else. ... Code is duplicated. The flow of control is hard to understand, and difficult to follow. ... The code is simply unreadable, and borders on indecipherable.”

—*Big Ball of Mud*, Brian Foote & Joseph Yoder

# Refactoring and Patterns

- The refactorings in Martin Fowler's book are very fine grained
  - Useful, but generally in a very limited scope
- Often, it is beneficial to refactor code towards design patterns
  - Being careful not to over-engineer the design
- How to find the correct pattern?
  - Read the short description of the patterns (the Intent section)
  - Read the Applicability section to discover the problem the pattern addresses
- “There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else.”

—Martin Fowler

# Evolutionary Design

- How to become a better software designer
  - Study the evolution of great software designs
  - Not just the design that is the end result
- "... studying the evolution of great software designs will be more valuable than studying the great designs themselves. For it is in the evolution that the real wisdom lies."

—Joshua Kerievsky

- View patterns in the context of refactoring your code
  - Not just as reusable components
- "That which thy fathers have bequeathed to thee, earn it anew if thou wouldest possess it."

—Goethe

*Refactoring to Patterns*, Joshua Kerievsky  
*The Architecture of Open Source Applications*, Volumes I and II

# Chapter Concepts

Using Design Patterns

---

## **Examples of Design Patterns in Java**

---

Factories: Creating Objects

---

Refactoring to Design Patterns

---

Chapter Summary

---

# Java Contains Many Design Patterns

- Java came out around the same time as Design Patterns
  - So, heavily influenced by them
  - Stole some of their names for interfaces and classes—very confusing!
- Let's look at some real-world examples
  - Strategy: Comparator
  - Iterator: Iterator **and** Iterable, **also** Scanner
  - Template Method: AbstractList **and** AbstractSet
  - Observer: Observer **and** Observable, **also** event listening
  - Decorator pattern: Reader **and** Writer
- Remember, you can look at Java's source code if you want!

# Strategy Pattern

- Name: Strategy
- Problem: How to design a collection of related algorithms while allowing the particular strategy to vary independently of, and transparently to, the client?
- Solution: Declare a common interface for all supported algorithms. Introduce a Context object that has a reference to the selected Strategy. The client has a reference to the Context.
- Consequences:
  - The selected Strategy is hidden from the client
  - The Strategy can change dynamically without affecting the client
  - An abstract base class can replace the Strategy interface to define common Strategy elements
  - Some Strategies may not need information provided by the Context

# Strategy: Comparator<T>

## ■ Strategy

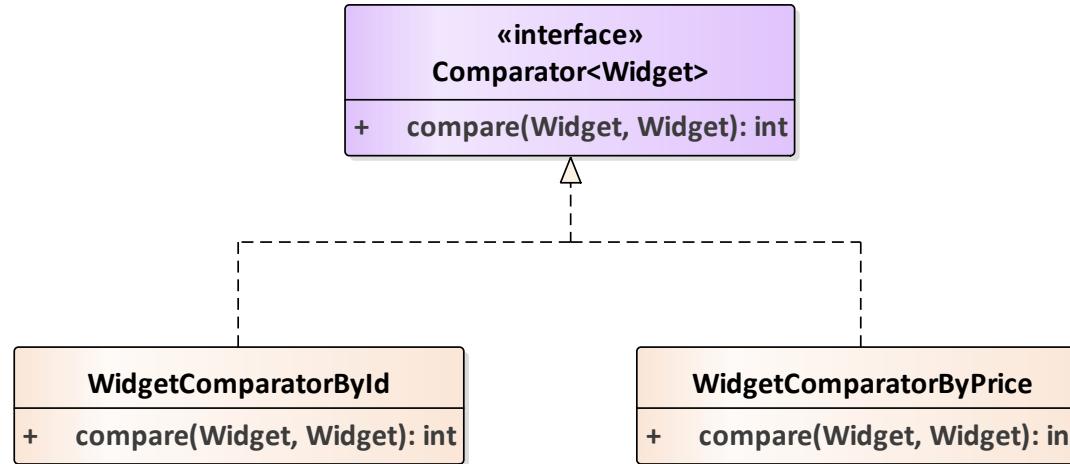
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

## ■ Java Comparator interface:

### ■ Interface Comparator<T>

- int compare(T o1, T o2)
- Compares its two arguments for order
- Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second

# Widget Comparators

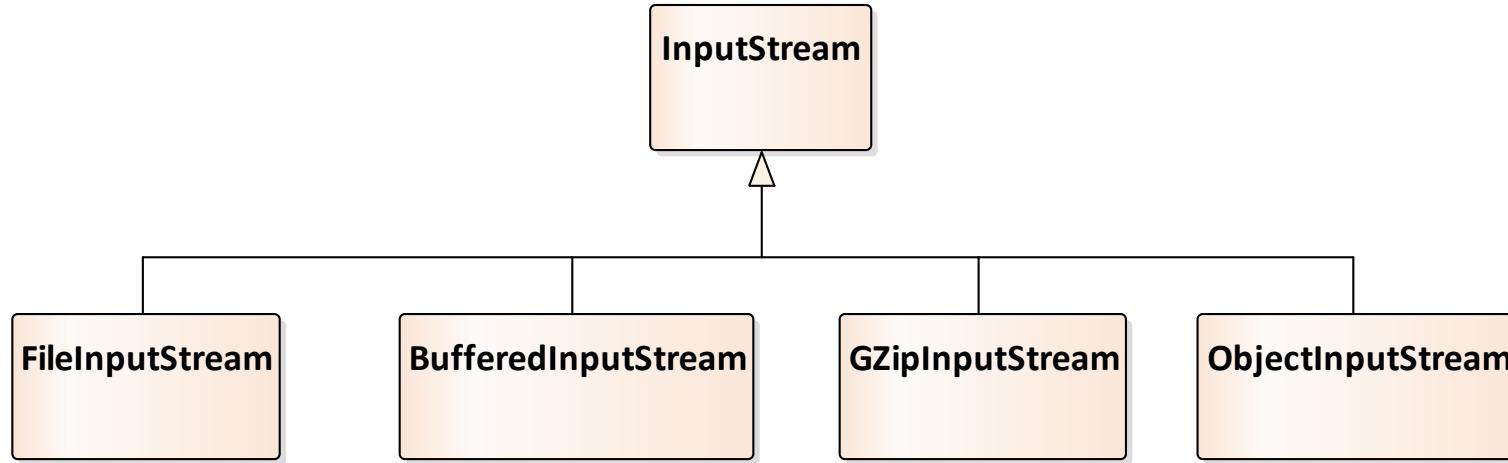


# Decorator: InputStream

- Scenario: We must read a compressed file that contains a collection of serializable Java objects. We can use several classes from the Java library and arrange them by using the Decorator pattern.
  - We will use a FileInputStream to read the file
  - For performance, we will use a BufferedInputStream
  - Since this is a compressed file, we will unzip it with the GzipInputStream
  - Finally, we will deserialize the contents into a collection of Java objects
- Each decorator delegates to the next decorator in the chain
  - Or the target
  - In this case, the FileInputStream that knows how to read the file

# Java InputStreams

- There are several types of InputStreams in the Java library



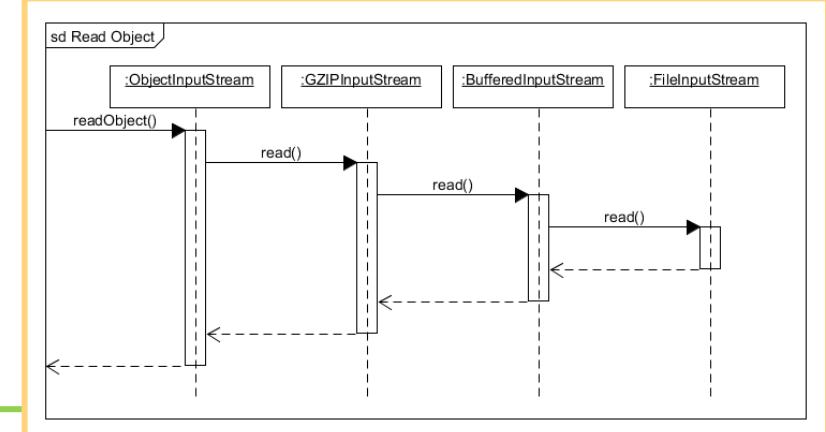
# Decorator: InputStream

- Each input stream is “wrapped” at the next level
  - Only FileInputStream needs to know how to read a physical file
  - But all streams know how to work with another stream

```
public List<Widget> readCompressedWidgetsFile(String path)
    throws FileNotFoundException, IOException, ClassNotFoundException {
    List<Widget> widgets = new ArrayList<>();

    try (ObjectInputStream objectStream =
        new ObjectInputStream(
            new GZIPInputStream(
                new BufferedInputStream(
                    new FileInputStream(path))))) {
        widgets = (List<Widget>) objectStream.readObject();
    }

    return widgets;
}
```



# Chapter Concepts

Using Design Patterns

---

Examples of Design Patterns in Java

---

## **Factories: Creating Objects**

---

Refactoring to Design Patterns

---

Chapter Summary

---

# Normal Way to Create Objects

- Normal way to create an object is to provide a constructor
  - Clients can invoke constructor to create objects

```
List<Gate> gates = ...  
Exhibit e = new Exhibit("Moore", gates);
```

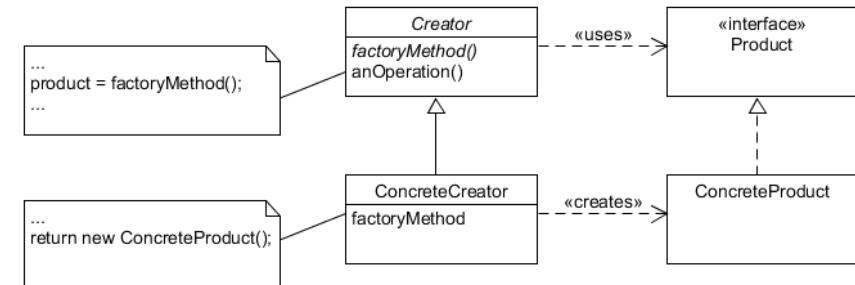
- Subclasses invoke constructor as part of their construction

```
public class VisitingExhibit extends Exhibit {  
    private String donor;  
    public VisitingExhibit(String name, List<Gate> gates, String donor) {  
        super(name, gates);  
        // etc  
        ...  
    }  
}
```

- Constructors are good if construction is just a process of initializing fields
  - What if there are several different classes to choose from?
  - What if construction is complicated?

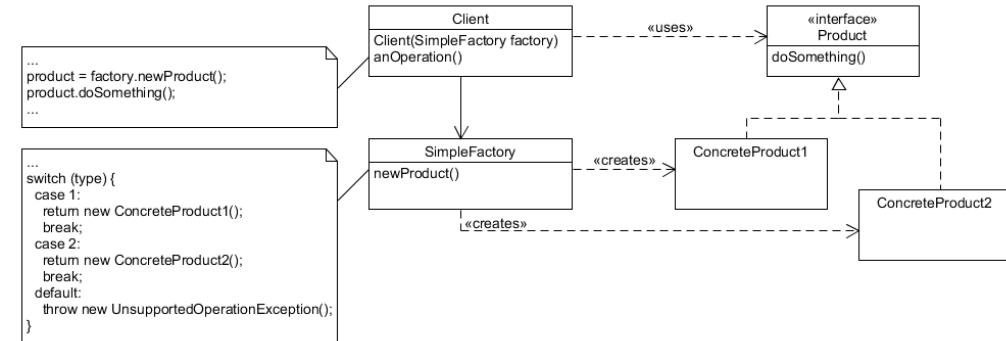
# Factory Method Pattern

- Gang of Four introduced the Factory Method Pattern
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Narrow range of application
  - Defer decision to subclass
  - “Template Pattern for object creation”
- However, some aspects apply more widely:
  - Situations where a class can't anticipate the class of objects it must create
  - When we want to encapsulate which of several classes is being used
- Gave rise to:
  - Simple Factory Pattern (or Factory idiom)
  - Static Factory Method Pattern



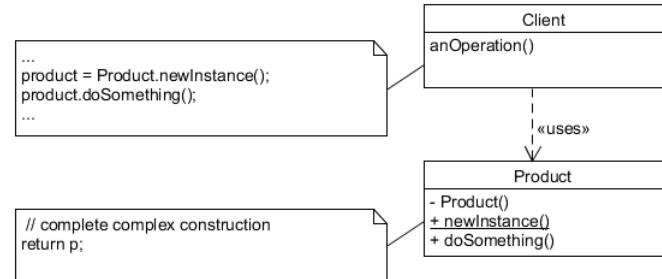
# Simple Factory Pattern

- Problem: How to obtain a reference to an object that implements an interface without programming to the implementing class. How to encapsulate object creation.
- Solution: Use a factory class that will create the object that implements the interface and return a reference to that object in response to client requests
- Implementation:
  - Factory may be supplied when the Client is constructed
  - How the factory determines which class to instantiate varies
- Widely used (e.g., Spring)
- Commentators disagree over whether this is a pattern or an O-O idiom
  - May also be called Factory Pattern



# Static Factory Method Pattern

- Problem: How to create an object with complex construction (e.g., dependencies). How to restrict number of instances of a class. How to distinguish between different constructors.
- Solution: Use a static factory method to return the instance. Make the constructor private or protected to avoid inadvertent use.
- Widely used in Java:
  - LocalDate.of(), .from(), .parse(), .now()
  - DriverManager.getConnection()
  - Boolean.valueOf()
  - BigInteger.probablePrime()
- Overlaps with other patterns:
  - Singleton
  - Flyweight
  - Object Pool



# Complex Construction

## Remember our Exhibit?

```
List<Gate> gates = ...  
Exhibit e = new Exhibit("Moore", gates);
```

- What if the Gate constructor needs an Exhibit?
  - Must create Exhibit to satisfy Gate and a list of Gates to satisfy Exhibit
- Leaving complex construction to the client code would be error-prone
  - Instead, create Static Factory Method

```
public Gate(Exhibit e, String name) {  
    this.e = e;  
    this.name = name;  
}
```

```
public static Exhibit newInstance(String name, int numGates) {  
    Exhibit e = new Exhibit(name);  
    for (int i=0; i < numGates; ++i) {  
        Gate g = new Gate(e, "Gate no." + (i+1));  
        e.addGate(g);  
    }  
    return e;  
}  
  
private Exhibit(String name) {this.name = name;}
```

Constructor made private to prevent inadvertent use.  
The class cannot be subclassed unless the constructor  
is made protected instead

# Restrict Number of Instances of a Class

- Constructors have to create a brand-new object each time
  - May be wasteful
- (Static) factory methods can create a pool of objects
  - And return appropriate instance to users as needed
  - Done by many API classes; e.g., `Boolean.valueOf()` uses a pool of two objects
- Pooled objects can offer additional guarantees
  - For example, Boolean can guarantee that:

```
Boolean a = ...  
Boolean b = ...
```

```
if (a.equals(b)) {  
    ...  
}
```

Equivalent

```
if (a==b) {  
    ...  
}
```

# Distinguish Between Different Constructors

- Constructors have to have name of class
  - Can lead to error-prone code

Constructor and Description
<b>BigInteger(byte[] val)</b> Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger.
<b>BigInteger(int signum, byte[] magnitude)</b> Translates the sign-magnitude representation of a BigInteger into a BigInteger.
<b>BigInteger(int bitLength, int certainty, Random rnd)</b> Constructs a randomly generated positive BigInteger that is probably prime, with the specified bitLength.
<b>BigInteger(int numBits, Random rnd)</b> Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to ( $2^{\text{numBits}} - 1$ ), inclusive.
<b>BigInteger(String val)</b> Translates the decimal String representation of a BigInteger into a BigInteger.
<b>BigInteger(String val, int radix)</b> Translates the String representation of a BigInteger in the specified radix into a BigInteger.

These two constructors do very different things, but have similar parameters

probablePrime
<pre>public static BigInteger probablePrime(int bitLength,  Random rnd)</pre>
Returns a positive BigInteger that is probably prime, with the specified bitLength. The probability that a BigInteger returned by this method is actually prime is at least 63.2%.
<b>Parameters:</b>
bitLength - bitLength of the returned BigInteger.
rnd - source of random bits used to select candidates to be tested for primality.
<b>Returns:</b>
a BigInteger of bitLength bits that is probably prime
<b>Throws:</b>
ArithmaticException - bitLength < 2 or bitLength is too large.
<b>Since:</b>
1.4
<b>See Also:</b>
<a href="#">bitLength()</a>

- Static factory methods can use names to describe what the methods do

# Common Names for Static Factory Methods

- One disadvantage of factory methods is that they don't stand out
  - Constructors listed separately in documentation
- Use commonly accepted names for static factory methods:
  - `of`, `valueOf`
    - To create simple objects like enumerations, etc. that have obvious “values”
    - A `valueOf` method typically does something akin to parsing
  - `newInstance`, `newValue`
    - To create objects; each call returns a different object
  - `getInstance`
    - May have returned this instance to a previous caller
    - Often used with pooled objects

# Factories Can Return Subclasses

- One key advantage of factories is that they can return subclasses
  - `Exhibit.newInstance()` can return any subclass of `Exhibit`
  - `new Exhibit()` has to instantiate a class of type `Exhibit`
- Uses:
  - Can hide the names of implementation classes
    - Clients can create class using parameters
    - Can use class as essentially the superclass/interface
    - Low “conceptual weight” for the API
  - List<Exhibit> exhibits = Collections.singletonList(e);
  - Can allow new implementations to be added and registered
    - Factory can read list of subclasses from config file and return appropriate object

# How to Create Objects

## ■ Two ways of creating objects

- Constructors
  - Use for simple initialization
- Factories
  - Use for more complex construction scenarios
  - Use for pooled objects
  - Use when different types of creation have to be distinguished by name
  - Use to hide details of which class is being instantiated

## ■ Don't just provide constructors out of habit

- Consider whether a factory is warranted
- If in doubt, just provide a constructor
  - Simpler, and "more normal"

# Chapter Concepts

Using Design Patterns

---

Examples of Design Patterns in Java

---

Factories: Creating Objects

---

## Refactoring to Design Patterns

---

Chapter Summary

---

# Identifying the Problem

- Identifying the problem is essential
  - And then locate a pattern that may help to solve it
- The inside front cover of *Design Patterns*<sup>1</sup> contains a concise description of each pattern
  - Use it to help identify potential candidate patterns
  - These are provided in Appendix B: Gang of Four Design Patterns
  - Remember to consider how to adapt a pattern to fit your particular situation
  - Design patterns are not written in stone!
- Working with patterns gets easier with experience
  - “The best way to get a feel for using patterns is, well, to *just use them.*”<sup>2</sup>

1. *Design Patterns*, Gamma, Helm, Johnson, Vlissides (the Gang of Four)
2. *Pattern Hatching*, John Vlissides (one member of the Gang of Four)

# Refactoring to a Pattern

- The first step is to fully understand the problem
  - Often this does not occur until later in development
  - More complete understanding of the problems in the application takes time
  - Applying the design pattern will then be a process of refactoring existing code
- Before you can determine the right pattern to use, the problem must be well understood
  - Don't try to determine all the patterns to use too early
  - Wait till you understand the problems more completely
- Select a pattern that suits the problem and adapt it to the particular situation
  - Patterns are basic recipes; they need work to suit your problem



HANDS-ON  
EXERCISE

# Exercise 13.1: Refactoring to a Pattern

30 min

- Follow the instructions in your Exercise Manual for this exercise

# Chapter Concepts

Using Design Patterns

---

Examples of Design Patterns in Java

---

Factories: Creating Objects

---

Refactoring to Design Patterns

---

**Chapter Summary**

---

# Chapter Summary

In this chapter, we have explored:

- The role of design patterns
- Design patterns in Java

# Key Points

- Design patterns provide proven solutions to commonly occurring software problems
- Design patterns are the target of refactoring
- Beware of under- and over-engineering your designs
- The best way to learn to use design patterns is to start using them!

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

## Course Summary

# Course Summary

In this course, we have:

- Practiced TDD in Java
- Written Java code to represent UML classes
- Applied Inheritance in Java
- Used effective programming practices
- Used a variety of classes to support different goals
- Used abstract classes and interfaces
- Refactored code to improve design while developing code
- Implemented exceptions to handle unusual issues
- Effectively utilized the functional programming features of Java

# Design Pattern Resources

- *A Pattern Language*, Christopher Alexander
- *The Timeless Way of Building*, Christopher Alexander
- *Design Patterns*, Gamma, Helm, Johnson, Vlissides
- *Patterns of Enterprise Application Architecture*, Fowler
- *Refactoring to Patterns*, Kerievsky
- *Enterprise Integration Patterns*, Hohpe & Woolf
- *Pattern Hatching*, Vlissides
- *Ajax Design Patterns*, Mahemoff
- *Head First Design Patterns*, Freeman & Freeman

# Keys to Success



- Practice looking for opportunities to use design patterns
- The original *Design Patterns* book is still useful—use it!
- Many other patterns books have been written for various types of environments and programming languages

# Fidelity LEAP

Technology Immersion Program

## Programming with Java

## Appendix A: Java Basic Syntax

# Appendix Concepts

## Basic Program Structure

---

Variables and Primitives

---

Blocks and Control Statements

---

Arrays

---

Char and Unicode

---

Java Compiler, VM, and Garbage Collector

---

Additional Reading

---

# Applications Start in `main()`

- The `main()` method should be defined exactly as shown

- Everything in Java is case sensitive!
- Whitespace does not matter

- Name of variable “`args`” is up to you

- Code within `main()` method will be executed to start program

- Program exits once all the lines of code in `main` have been executed
- Unless a separate thread was launched by application

- Most modern applications provide `main` for you so we do not focus on using it anymore
  - Desktop GUIs, Mobile, Web, Cloud apps all have specific configurations

```
package com.fidelity.hello;

public class HelloWorld {

    public static void main(String[] args) {
        // the obligatory first program
        System.out.println("Hello World!");
    }
}
```

# All Code is in Classes

- Every class is in a package
  - Package defines a unit of code
  - Acts as a namespace
  - Choose a descriptive name
- Most classes do not have `main()`
  - Only program entry points
- A class contains members:
  - Field definitions (data)
  - Method definitions (behavior)
  - This class contains 3 members
- In general, each class is in a separate file called  
`ClassName.java`

```
package com.fidelity.examples;

public class ExampleClass {
    private int result;

    public static void main(String[] args) {
        // main is static, not object-oriented
        // get out of main as quickly as possible
        ExampleClass me = new ExampleClass();
        me.dowork();
    }

    public void dowork() {
        // work hard here
        System.out.println("The result is " + result);
    }
}
```

# Java Naming Conventions

■ These are closely observed, so follow them to make your code easily understood

Type	Convention	Examples
Package	Usually the reversed domain name of the organization that owns the package followed by a project or department specific name. All lowercase. (Reversed to help with sorting.)	com.oracle.net com.fidelity.example
Class	A noun, or noun phrase. Mixed case, starting with a capital and with each internal word capitalized. Keep them short and descriptive. Avoid acronyms and abbreviations, except where the abbreviation is more commonly used (e.g., HTML) or so commonly used that it is expected.	class Employee class EncryptingProxy
Interface	Follow conventions for class, except that they may be adjectives. Interfaces often indicate a capability and may be suffixed with –able. Generally, do NOT use the leading capital I that is found in .Net and other Microsoft products.	interface Storable interface EmployeeDao
Method	A verb, or verb phrase. Mixed case, starting with lowercase and with each internal word capitalized. Methods starting with the verbs get, set, and is have special significance that should be respected.	run() runFast()
Variable	A noun, or noun phrase. Except for class constants, all variable names are mixed case, starting with lowercase and with each internal word capitalized. Should not start with _ or \$, even though both are allowed. Keep them short yet meaningful. Only use one-character variable names for temporary "throwaway" variables.	int i; long myWidth; String employeeName;
Constant	Class constants should be all uppercase with words separated by underscores	static final int MIN_WIDTH = 4;

# Appendix Concepts

Basic Program Structure

---

**Variables and Primitives**

---

Blocks and Control Statements

---

Arrays

---

Char and Unicode

---

Java Compiler, VM, and Garbage Collector

---

Additional Reading

---

# Declare Before Use

- In Java, you need to declare variables before you can use them
  - Specify what the type should be:

```
int x;  
double y;  
x = 32;  
y = x / 4.3;
```

- Can also combine declaration and usage:

```
int x = 32;  
double y = x / 4.3;
```

- All statements have to end with a semicolon

# Primitive Data Types

- There are eight Java primitive types

Type	Size	Usage
boolean		Simplifies logical operations
byte	1 byte (8-bits)	Access to OS memory
char	2 bytes	Represent letters and numbers (International/Unicode characters)
short	2 bytes	For whole numbers under 32K
int	4 bytes	Represent whole numbers
long	8 bytes	Very large whole numbers
float	4 bytes	Fractions such as 1.42
double	8 bytes	More precise fractions

# Special Notation for Literals

- Can use underscores to make values more obvious (since Java 1.7)

```
int bigNumber = 42981111;  
int bigNumber2 = 42_981_111;  
int lakh = 1_00_000;
```

- With float and long assignments, you must postfix a letter to literal values

```
float myFloat = 1.238f;  
long myLong = 2637125L;
```

- With char literals, use single quotes

```
char someLetter = 'B';  
char aUnicodeCharacter = '\u20b9';
```

# Expressions and Operators

```
x = x + 4;      // add
x = x - 4;      // subtract
x = x * 4;      // multiply
x = x / 4;      // divide: if x is an short/int/long this truncates
x = x % 4;      // modulo (remainder) 11 % 4 is 3

x += 2;         // add to self; -= *= /= also work

++x;            // increment x
x++;            // increment x
y = x++;        // postincrement: if x was 3, y will be 3 and x will be 4
y = ++x;        // preincrement: if x was 3, y and x will both be 4

b = ( x > 10 && y < 5 );      // AND      (b will have value false)
b = !( x > 10 || y < 5 );    // NOT, OR (b will have value true)
```

# Short-Circuit Operators

- The logical AND and logical OR are short-circuit operators
  - Java stops evaluating as soon as the Boolean answer is clear

```
int x = 5;  
int y = 4;  
boolean b = false;
```

Starting values: x=5, y=4. x>10 is false, expression cannot be true, so y++ is never executed

```
b = x > 10 && y++ < 5;
```

```
System.out.println("Evaluated to " + b + ", value of y=" + y);
```

Evaluated to false, value of y=4

```
b = x > 10 || y++ < 5;
```

Starting values: x=5, y=4. x>10 is false, so y++ < 5 is executed to see if expression might be true (y is incremented)

```
System.out.println("Evaluated to " + b + ", value of y=" + y);
```

Evaluated to true, value of y=5

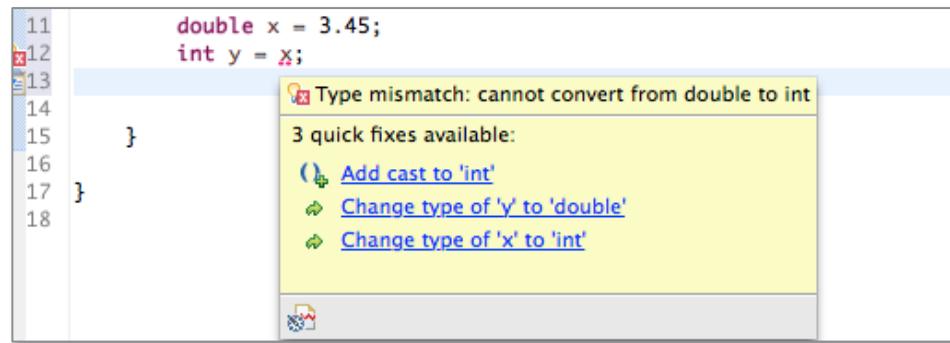
# Copying Primitive Values

- Assigning primitive to another variable with compatible data type results in separate primitives
  - Later changes to first primitive do not affect the other

```
int x = 3;  
double y = ++x;  
x = 5;  
// what is y?
```

y == 4

- If you try to assign a `double` value to an `int` reference, the compiler will not allow it



# Casting of Primitives

- Storing smaller values into larger ones, such as `int` into a `long` is safe
- Trying to store a larger data type in a smaller one is dangerous
  - You can lose precision
  - Casting is special syntax to let compiler know you really want to do this

```
int x = 32;

long y = x;          // safe: no special syntax needed

int z = (int) y; // need this special cast syntax
```

# Appendix Concepts

Basic Program Structure

---

Variables and Primitives

---

## **Blocks and Control Statements**

---

Arrays

---

Char and Unicode

---

Java Compiler, VM, and Garbage Collector

---

Additional Reading

---

# Blocks of Code

- You can group a bunch of statements together into a block of code
  - Called a compound statement
- Blocks of code are indicated by curly braces
  - Within a block of code, statements are processed in order

```
{  
    int x = 3;  
    int y = x + 15;  
    x = y - 23;  
}
```

- Curly braces denote scope in Java
  - `x` and `y` cannot be used outside the end-curly brace
  - Because they were declared inside the curly braces

# Conditional Statements

- if-then statements and if-then-else statements have the syntax shown
  - Final else is catchall in case none of the other conditions evaluated to true

```
if (x > 10) {  
    y = x / 4.3;  
}
```

```
if (x > 10) {  
    y = x / 4.3;  
} else if (x < 10) {  
    y = x / 3.4;  
} else {  
    y = x / 2.1;  
}
```

```
if (x > 10 && x < 10) {  
    y = x / 4.3;  
} else {  
    y = x / 2.1;  
}
```

- Also supports C-like ternary operators and switch statements

```
int z = (x > 3) ? 1 : 0;
```

# Coding Conventions

- Java supports replacing compound statements by a single statement

```
if (x > 10)
y = x / 4.3;
// What happens if maintainer adds a line here?
```

- The Java best practice is to always use compound statements
  - Makes it easy to add new processing to the if-block later

```
if (x > 10) {
y = x / 4.3;
// can add line here later without changing meaning
}
```

# switch Statement

- A switch statement can improve readability over multiple “if, else” blocks
- The evaluated expression can be byte, short, char, int, and, as of Java 1.7, String or enumerated type

```
public String foodChoice(String food) {  
    String foodResponse = "";  
  
    switch(food.toLowerCase()) {  
        case "pizza": // since there is no break, this will fall through  
        case "tacos":  
            foodResponse = "I love " + food;  
            break;  
        case "anchovies":  
            foodResponse = "I hate " + food;  
            break;  
        default:  
            foodResponse = "No thanks, I'm really not hungry!";  
            break;  
    }  
  
    return foodResponse;  
}
```

# Loops

- In addition to the for-each loop, Java supports while, do-while, and for loops:

```
while (x > 10) {  
    x = x / 3;  
    y = x + 4;  
}
```

```
do {  
    x = x / 3;  
    y = x + 4;  
} while (x > 10);
```

```
for (int i = 0; i < 10; i += 2) {  
    j = j + i;  
}
```

# Looping Over Strings with `for-each`

- Strings represent a variety of kinds of data:

- Letters

```
for (char c : str.toCharArray()) {
```

Consider "\s+" for more generic whitespace

- Words

```
for (String w : str.split(" ")) {
```

- Lines

```
for (String l : str.split("\r\n")+) {
```

Handles both Windows and Mac line endings

- Formatted data

```
for (String l : str.split(":")) {
```

- Same syntax for looping over any collection in Java (List, Array, etc.)

# break and continue

- You can skip part of a loop
  - Could `continue` loop with next iteration
  - Could also `break` out of loop completely

```
for (int x = 0; x < 10; x += 2) {  
    System.out.println(x);  
    if (x < 4) {  
        continue;  
    }  
    System.out.println("ok");  
}
```

0  
2  
4  
ok  
6  
ok  
8  
ok

```
for (int x = 0; x < 10; x += 2) {  
    System.out.println(x);  
    if (x > 4) {  
        break;  
    }  
    System.out.println("ok");  
}
```

0  
ok  
2  
ok  
4  
ok  
6

# Comments

- There are three types of comments in Java:

```
x = x + 4; // single-line comment
```

```
/*
A block comment.
*/
```

By default, Eclipse will add an asterisk to the start of every line

```
/**
This Javadoc comment documents the method that
follows.
*/
```

# Appendix Concepts

---

Basic Program Structure

---

Variables and Primitives

---

Blocks and Control Statements

---

## Arrays

---

Char and Unicode

---

Java Compiler, VM, and Garbage Collector

---

Additional Reading

---

# Declaring Arrays

- Arrays are a group of data, with the same datatype, whose size is fixed when it is created
  - Returned by many standard Java methods, so useful to know
  - But not often useful to create directly because size does not grow to accommodate data
- For example, `main()` method receives an array of `String`s
- The `[]` can be on the data type or the variable name
- Examples of declaring a variable as an array datatype:

```
int arrayName[];  
double doubleArray[];  
String[] myStrings;
```

# Creating the Array in Memory

- The dimension of the array must be given in order for it to be used
  - The dimension is not changeable once declared
- Can either:

- Use the keyword `new`:
  - Default values are assigned

```
int[] myIntArray = new int[3];
```

0	0	0
---	---	---

myIntArray [ ]

- Supply comma-separated list of literal values, using curly braces

```
double[] myDoubleArray = { 12.5, 4.2, 69.1 };
```

12.5	4.2	69.1
------	-----	------

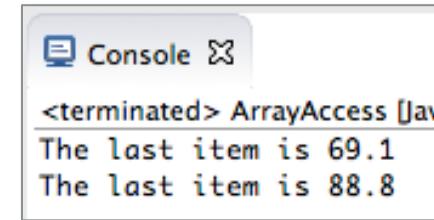
myDoubleArray [ ]

# Accessing Elements of an Array

- Elements of an array are accessed using the variable and square brackets
  - Special feature, all other Java objects use methods instead
- Array index starts with integer value of 0

```
double [] importantValues = { 12.5, 4.2, 69.1 };
System.out.println("The last item is " + importantValues[2]);

importantValues[2] = 88.8;
System.out.println("The last item is " + importantValues[2]);
```



# Finding Length of Array

- To determine the dimension of an array, use the `length` variable
- Returns the declared length of the array object, not how many positions are filled

```
int theAnswers [] = new int[4];
theAnswers[0] = 42;
theAnswers[1] = 4;

System.out.println("The length of the array is " + theAnswers.length);
```

# Looping Through an Array

- Best practice to loop through an array is to use its `length` field
- Remember, the first item has an index of 0, not 1

```
public double calculateTotal(double[] theValues) {  
    double total = 0;  
  
    for (int i = 0; i < theValues.length; i++) {  
        total += theValues[i];  
    }  
    return total;  
}
```

- Or use `for-each` syntax, if index is not needed

```
for (double value : theValues) {  
    total += value;  
}
```

# Creating Arrays of Arrays

- It is possible to have an array where each position holds reference to another array
- The syntax for creating an array of arrays:

```
int[][] grid = new int[2][4];
```

	0	1	2	3
0	[0] [0]	[0] [1]	[0] [2]	[0] [3]
1	[1] [0]	[1] [1]	[1] [2]	[1] [3]

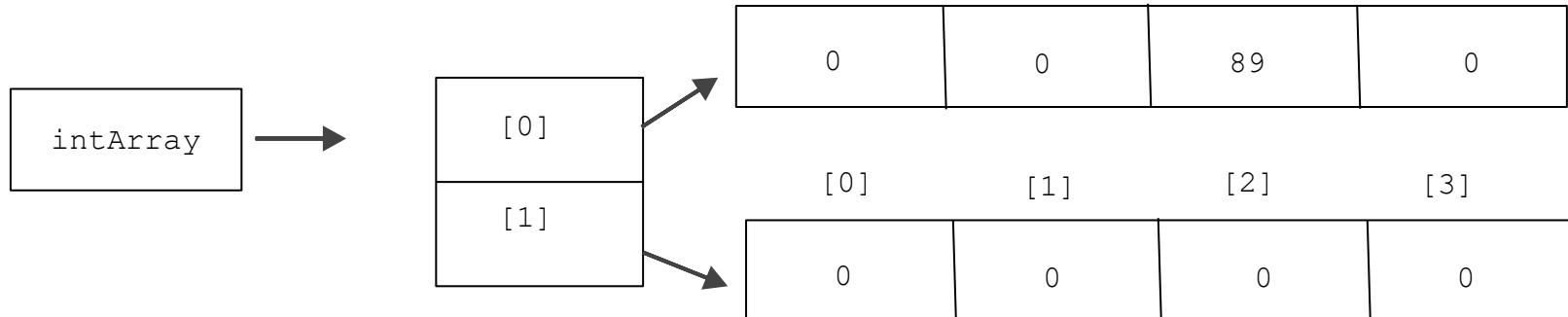
# Accessing Elements of Multidimensional Arrays

- Each array still has length

```
int gridlength = grid.length; // == 2  
int rowlength = grid[1].length; // == 4
```

- To store the value in row 0, column 2:

```
grid[0][2] = 89;
```



# Appendix Concepts

---

Basic Program Structure

---

Variables and Primitives

---

Blocks and Control Statements

---

Arrays

---

## Char and Unicode

---

Java Compiler, VM, and Garbage Collector

---

Additional Reading

---

# Can Use `char` Datatype for Escape Sequences

Escape Sequence	What It Will Do When Used in a Program
\n	New line. Position the screen cursor at the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop. <i>Tab spacing is every 8 columns starting with 1. (Columns 9, 17, 25, 33, 41, 49, 57, 65, 73 ...)</i>
\r	Carriage return. Position the screen cursor at the beginning of the current line—do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
\\"	Backslash. Used to print a backslash character.
\"	Double quote. Used to print a double-quote character. For example: <pre>System.out.println( "\"in quotes\"" ); displays "in quotes"</pre>
\b	Backspace

# Printing Using escape Sequences

- Example using \n and \"

```
System.out.print("\n Characters in quotes are called a \"String\"\nA\nB\nC");
System.out.print("\nThe \\n allows you to insert\nline breaks");
System.out.println("**");
```

Characters in quotes are called a "String"

A  
B  
C

The \n allows you to insert  
line breaks\*\*

println adds a line break  
at end, print does not

# Notes on Using char

- 16-bit unsigned Unicode values
  - Values stored as an integer from 0 - 65535
  - Examples: `½` `IV` `ix` `⊕`
- Characters are still numeric values
  - Able to participate in integer expressions
- When we used `\n` for a new line, that is an example of a `char`
  - The `char` datatype can represent escape sequences

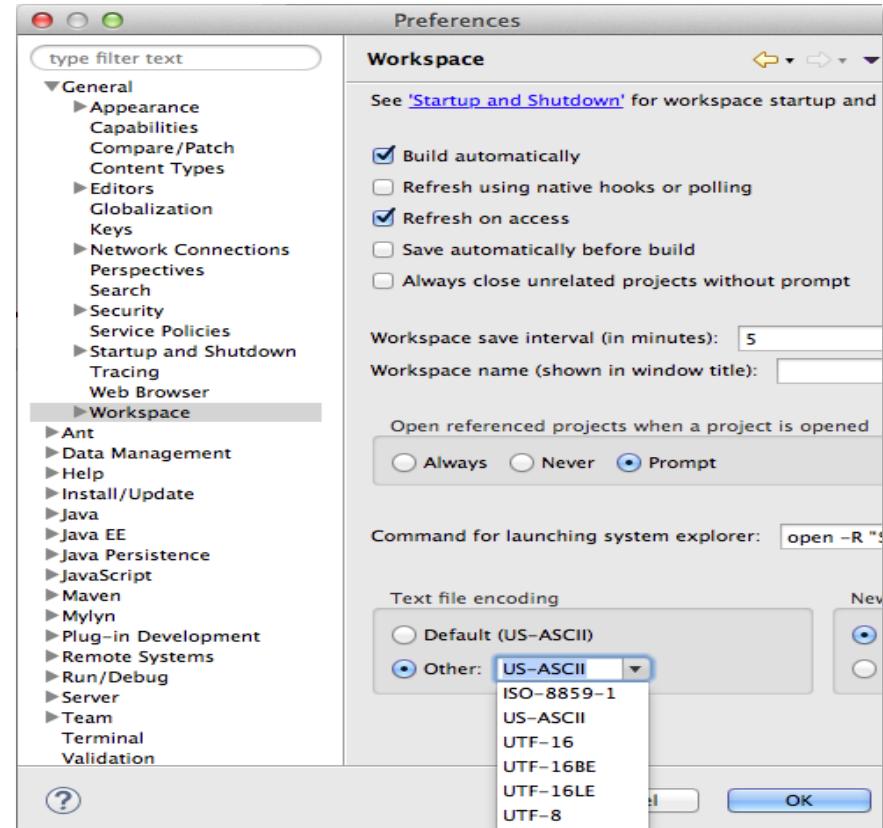
# Working with Unicode

- By default, some editors work with ASCII and not Unicode
- This will print out: ? but by Unicode, it should be Ω

```
char uniChar = '\u03A9';
System.out.println(uniChar);
```

# Unicode on Eclipse

- Can turn on UTF-8 character encoding



# Appendix Concepts

---

Basic Program Structure

---

Variables and Primitives

---

Blocks and Control Statements

---

Arrays

---

Char and Unicode

---

## **Java Compiler, VM, and Garbage Collector**

---

Additional Reading

---

# Compiling and Executing at the Command Line

- A program can be compiled from the command line

- Using the Java compiler `javac`

```
javac Driver.java
```

- If compilation is successful, output is class file named `Driver.class` in current folder
    - Same name as source file but with `.class` extension

- A program can be executed from the command line

- Using the command `java` to start the Java VM to interpret compiled code

- To execute the program in `Driver.class`

```
java Driver
```

- It will look for and execute static method `main()` in named class in the current folder
  - Eclipse typically stores `.class` files in the `bin/` or `build/` folder within your project
  - Can be deleted and rebuilt by re-compiling code if needed

# References to Objects Keep Them “Alive”

- Variables that references objects
  - Created using the keyword `new`
  - Assigning a reference to point to an existing object
  - Passing an object reference into a method
- As long as a reference is pointing to an object, it will be kept in memory

# Garbage Collector

- Memory is freed up in Java by the garbage collector
  - A separate thread runs looking at objects in memory
  - If no reference points to the object, that memory can be released or freed up
- Java uses a generational garbage collections system
  - Most objects have a short life span (weak generational hypothesis)
  - The garbage collector can take advantage of this

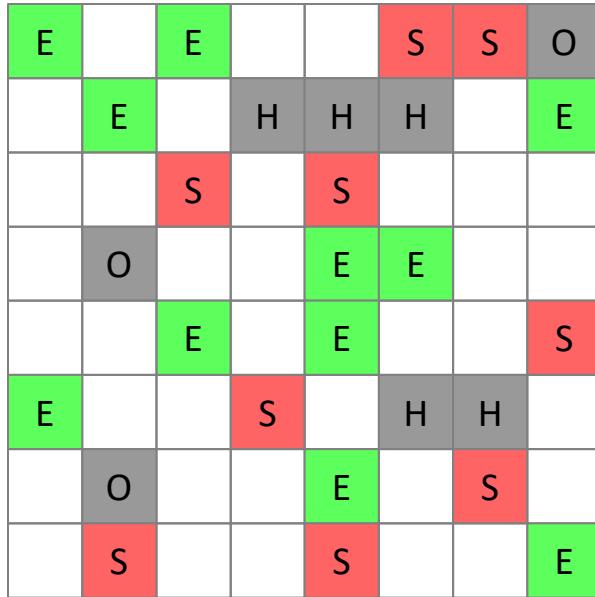
# Running the Garbage Collector

- It is possible to run the garbage collector in code
- Check out the Javadoc for the Runtime gc() method
  - And the System gc() method
- Making either of these calls is a suggestion to the virtual machine
  - It may be a good time to “clean up the garbage”
- This is never a good idea
  - The virtual machine has a much better idea of when to run the garbage collector
  - There is a very good chance that this will not run the garbage collector
    - Or possibly degrade the performance of your program
- Strong recommendation: let the runtime manage garbage collection

# "Stop the World" Events

- Some garbage collector activities are called "stop the world" (STW) events
  - All currently running threads are paused when they reach a suitable point
    - It may take time for all threads to come to a stop
  - The STW activity occurs
  - All threads are re-started
    - There may be a heavy CPU load as all threads re-start simultaneously
- Efforts to improve consistency of Java performance focus on reducing STW events

# Garbage First (G1) Collector



- Default in Java 9
  - Officially supported since Java 7u4
  - Earlier versions can enable by JVM option  
  `-XX:+UseG1GC`
- Java heap divided into fixed regions of variable purpose
  - Eden (new objects allocated here)
  - Survivor
  - Old
  - Humongous (object > half region size)
- Eden and Survivor are collectively known as the young generation
- Driven by a target latency  
(`-XX:MaxGCPauseMillis=<n>`, default 200ms)

# Young Generation Garbage Collection

- New objects are allocated in Eden space
- When Eden space is full, surviving objects are evacuated to survivor space
  - A new region is allocated from unallocated regions
  - Multiple Eden regions may be compacted into a single survivor region
  - This is a short STW event
- In the same collection event, objects in the survivor space may be promoted to old
  - If they have existed for a certain number of young generation collections

# Mixed Generation Garbage Collection

- When the Java heap passes a threshold, a mixed generation collection is started
  - Can be set `-XX:InitiatingHeapOccupancyPercent=<NN>`, default 45
- To find objects that are referenced, the garbage collector starts with “external roots”
  - Variables on the stack
  - Static variables in any loaded class
- The process has several steps:
  - Initial marking: finds all the roots. Piggybacked on a young generation GC. STW.
  - Root region scanning: finds old generation references from young.
  - Concurrent marking: finds reachable (live) objects across the entire heap.
  - Remark: completes marking by updating with changes since marking started. STW.
  - Cleanup: work out which regions are least-used.
  - Evacuation: moves live objects.
- Can suffer “to-space overflow” if application allocates objects faster than can be recovered

# Real Concurrent GC

## ■ Oracle JRockit Real Time and IBM Websphere Real Time

- Latency driven, throughput suffers
- STW pauses of a few ms to a few tens of ms
- Websphere RT constrains allocation rate
- JRockit constrains heap size
- Occasionally FullGC

## ■ Azul Zing

- Fully concurrent at all phases
- Sub-millisecond STW
- No constraints on heap size or allocation rate

# Garbage Collection Resources

- Java Garbage Collection Basics:
  - <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>
- Getting Started with the G1 Garbage Collector
  - <http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>
- Part 1: Introduction to the G1 Garbage Collector
  - <https://www.redhat.com/en/blog/part-1-introduction-g1-garbage-collector>
- G1 does not perform well with small heap size. Other GC algorithms:
  - Serial GC (-XX:+UseSerialGC) – only really useful in single threaded applications
  - Parallel GC (-XX:+UseParallelGC) – default prior to Java 9
  - Parallel Old GC (-XX:+UseParallelOldGC) – good throughput, ideal for batch
  - Concurrent Mark Sweep (CMS) (-XX:+UseConcMarkSweepGC) – low latency

# Measuring GC

## ■ Useful JVM options

- `-verbose:gc (&-Xloggc:<filename>)`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`
- `-XX:+PrintTenuringDistribution`
- `-XX:+PrintGCApplicationConcurrentTime`
- `-XX:+PrintGCApplicationStoppedTime`
- `-XX:+PrintAdaptiveSizePolicy`

## ■ Useful tools

- JVisualVM
- jHiccup
- Chewiebug GCViewer

# Appendix Concepts

---

Basic Program Structure

---

Variables and Primitives

---

Blocks and Control Statements

---

Arrays

---

Char and Unicode

---

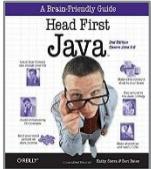
Java Compiler, VM, and Garbage Collector

---

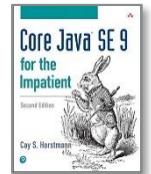
**Additional Reading**

---

# Additional Reading



- *Head First Java*, Sierra & Bates
  - Excellent and approachable introduction, but last updated in 2005



- *Core Java SE 9 for the Impatient*, Horstmann
  - Engaging, condensed version of the author's classic Core Java



- *Java: A Beginner's Guide*, Schildt
  - 7<sup>th</sup> Edition is fully updated to Java 9 with downloadable supplement for Java 10



- *Effective Java*, Bloch
  - Definitive guide to Java best practice by the author of the Java collection classes
  - 3<sup>rd</sup> Edition covers Java 9

# Fidelity LEAP

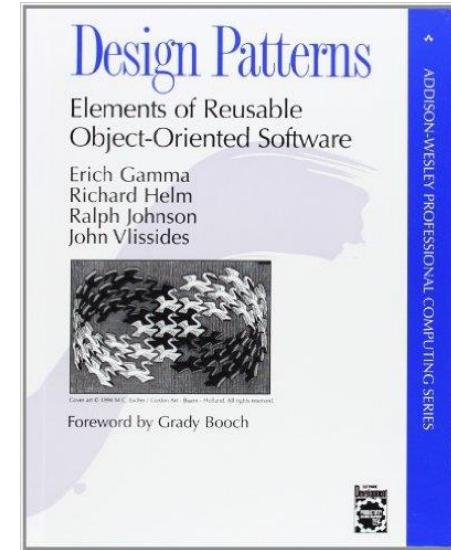
Technology Immersion Program

## Object-Oriented Analysis and Design

## Appendix B: Gang of Four Design Patterns

# Gang Of Four Design Patterns

- These design patterns are from the original Design Patterns book
  - Design Patterns Elements of Reusable Object-Oriented Software***
  - Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides



# Creational Patterns

## ■ **Abstract Factory**

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

## ■ **Builder**

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.

## ■ **Factory Method**

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

## ■ **Prototype**

- Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype.

## ■ **Singleton**

- Ensure a class has only one instance, and provide a global point of access to it.

# Structural Patterns

## ■ Adapter

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

## ■ Bridge

- Decouple an abstraction from its implementation so that the two can vary independently.

## ■ Composite

- Compose objects into tree structure to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

## ■ Decorator

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

## ■ Façade

- Provide a unified interface to a set of interfaces in a system. Façade defines a higher-level interface that makes the subsystem easier to use.

## ■ Proxy

- Provide a surrogate or placeholder for another object to control access to it.

# Behavioral Patterns

## ■ **Chain of Responsibility**

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

## ■ **Command**

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

## ■ **Observer**

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## ■ **Strategy**

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

## ■ **Template Method**

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

# Other Patterns

## ■ Data Access Object

- Provide an interface and an implementation for database operations.

## ■ Model View Controller

- Separate the functionality of an application into three areas. The Controller responds to user requests and obtains data from the Model. The View presents the data to the user and presents the user interface.

## ■ Factory

- Create a Factory class that will create an object that implements a specified interface and return a reference to that object in response to client requests.

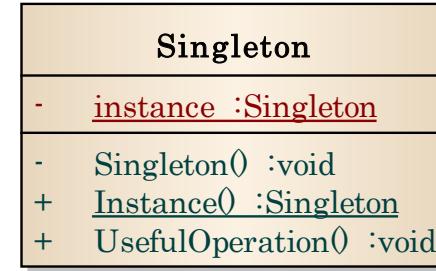
# Singleton

- Name: Singleton
- Problem: How to guarantee that only one instance of a class is created
- Solution: Create a Singleton class that provides a static method for clients to obtain access to the unique instance of the Singleton class
- Consequences: The Singleton class will declare its constructor with private accessibility
  - The unique instance can be created eagerly or lazily (on demand)
  - It would be possible to make a limited number of instances available, if that was required, while still maintaining control over the instance creation

# Singleton: Class Diagram

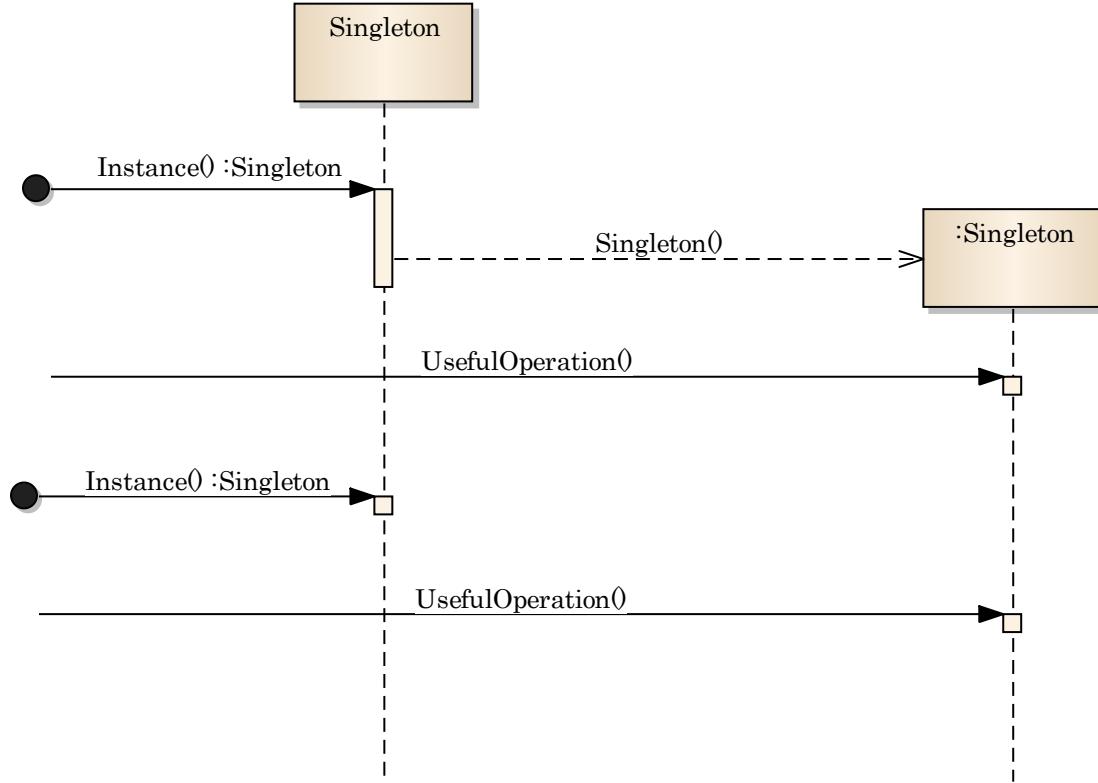
## The Singleton class

- Contains a static reference to the unique instance
- Contains a static method to obtain a reference to the unique instance
- Defines a private constructor to prevent uncontrolled object creation
- One or more public object scope methods for clients to invoke on the unique instance



# Singleton: Sequence Diagram

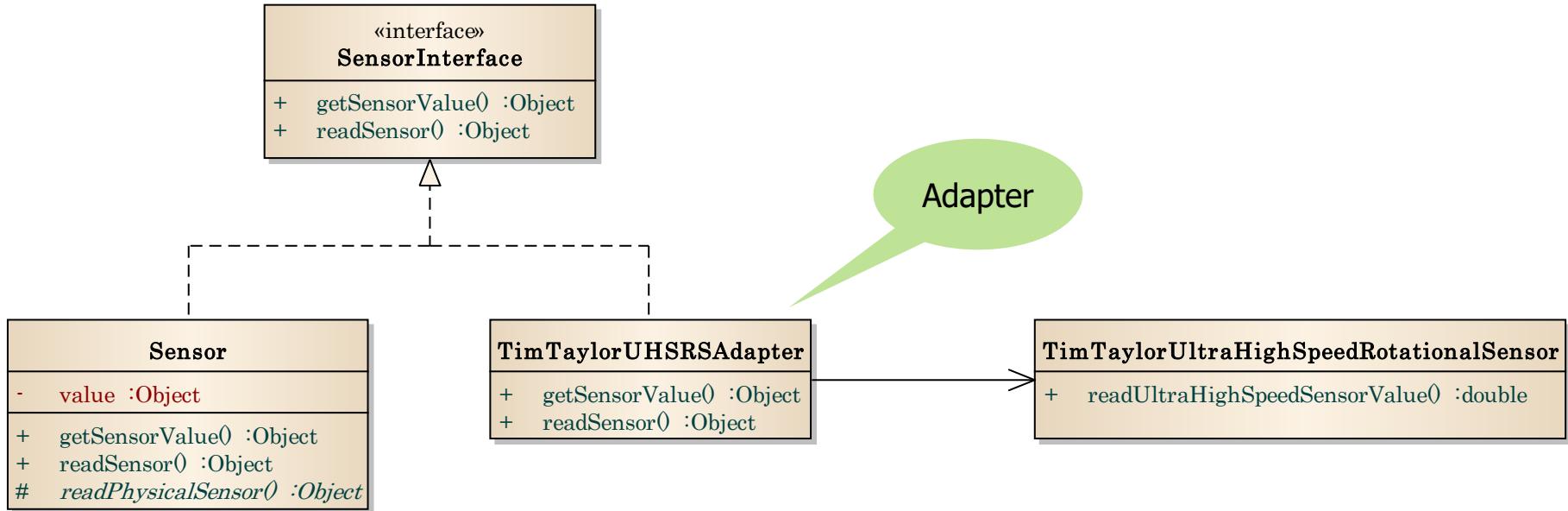
- This Singleton implementation uses lazy instantiation
  - Notice that the first request for the instance results in the unique instance being created
  - The second request simply returns the reference to the existing instance



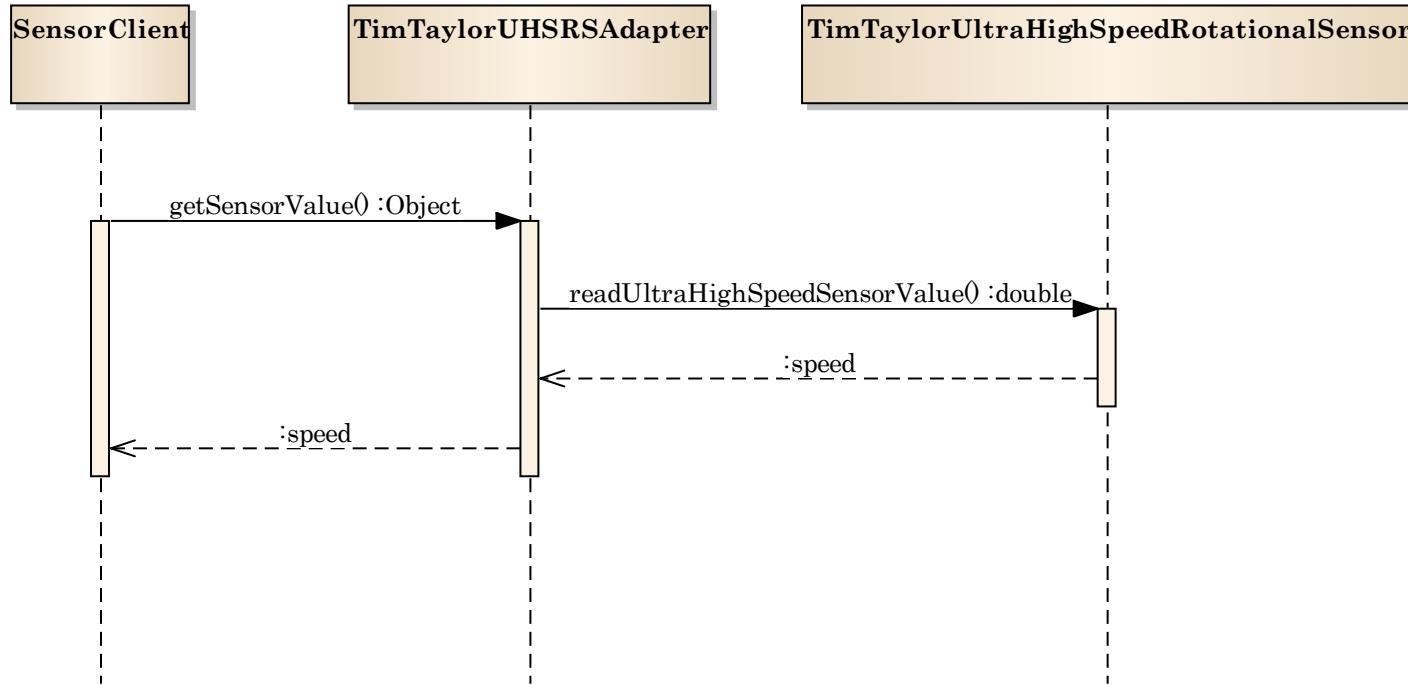
# Adapter

- Name: Adapter
- Problem: How to resolve incompatible interfaces
- Solution: Convert the original interface into a desired interface, by using an intermediate adapter object
- Consequences:
  - One adapter can adapt multiple interfaces
  - How much work does the Adapter do?
- Example:
  - The Home Heating system needs to work with a third-party sensor that provides an interface that is different than the interface defined by the Sensor base class
  - An Adapter for the third-party sensor will be the glue between the Home Heating system and the third-party sensor

# Adapter: Class Diagram



# Adapter: Sequence Diagram



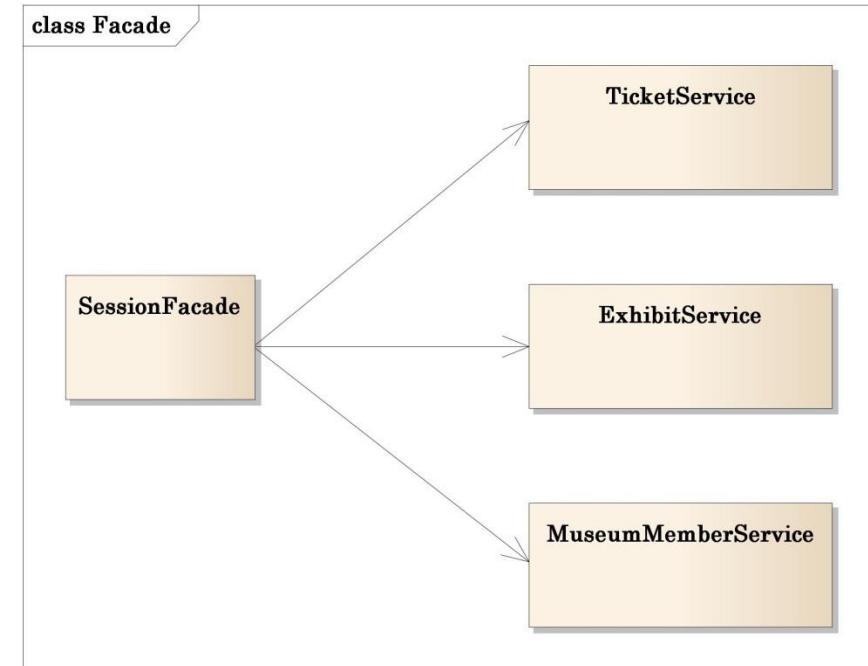
# Façade

- Name: Façade
- Problem: Provide access to a complex subsystem of many objects with many different methods that may be invoked
- Solution: Define a façade that provides a coarse-grained uniform interface to the subsystem for clients to use
- Consequences: Client-side programming is significantly simpler
  - One method invoked on the façade by the client may result in many methods invoked by the façade on the subsystem objects
  - For remote clients, this can dramatically reduce the number of remote calls from several calls to the subsystem to one call to the façade

# Session Façade Class Diagram

## ■ Session Façade in Java Museum

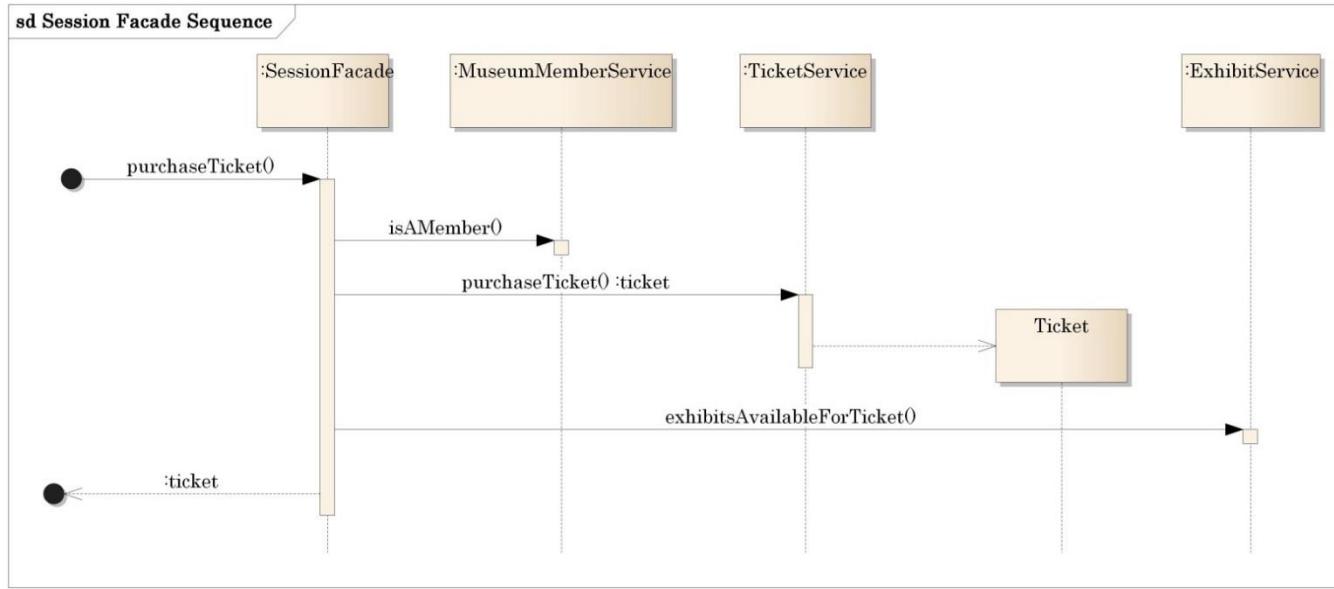
- Communicates with several business services
- Provides a coarse-grained interface for client applications
- Reduces the number of remote calls made by client applications



# Session Façade Sequence Diagram

## Session Façade in Java Museum

- Provides a coarse-grained interface
- Reduces the number of remote calls

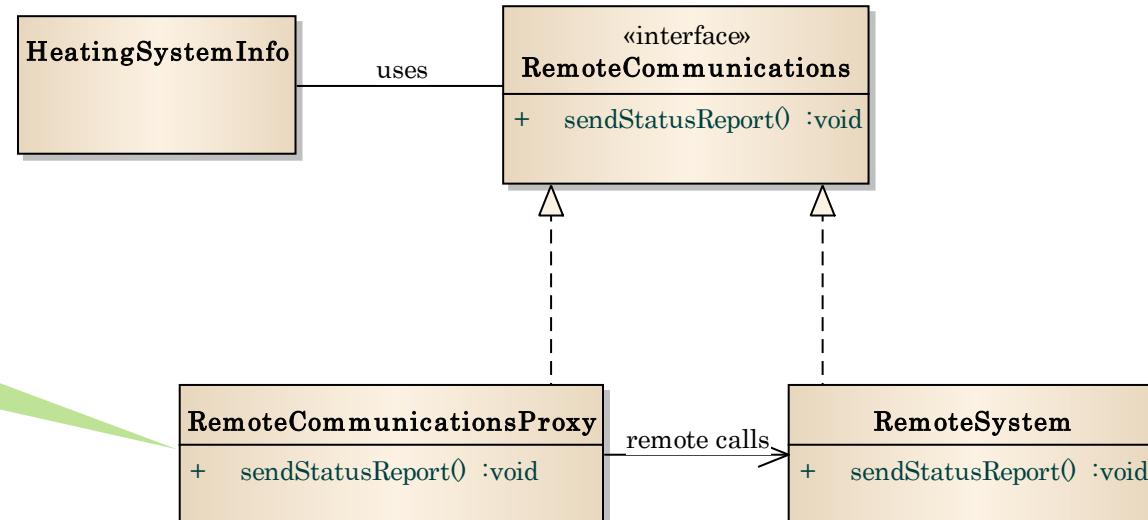


# Proxy

- Name: Proxy
- Problem: How to utilize a representative of another object to provide additional functionality such as security, communications, or performance
- Solution: Define a proxy class that implements the same interface as the original object. The proxy will define the additional functionality. The client will communicate with the proxy.
- Consequences:
  - Proxy has an opportunity to define pre- and post-processing
  - Proxy can determine if it will delegate the call to the original object
  - The functioning of the Proxy is transparent to the client

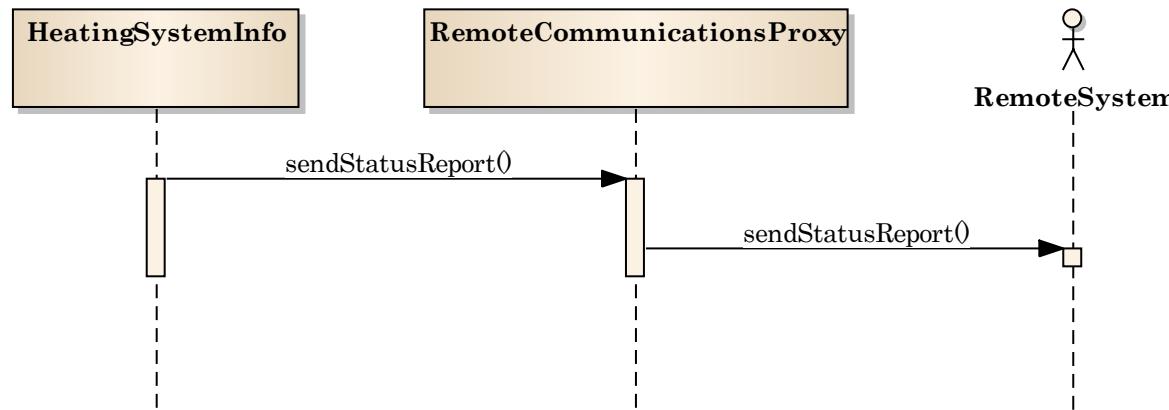
# Proxy: Class Diagram

- The Home Heating System must send a status report to remote destinations
  - The remote communications systems have provided an interface that they implement
- To keep the low-level communications details out of the application code, a proxy that manages the remote communications will be used



# Proxy: Sequence Diagram

- The HeatingSystemInfo uses the RemoteCommunications interface
- The RemoteCommunicationsProxy implements the interface provided by the RemoteSystem

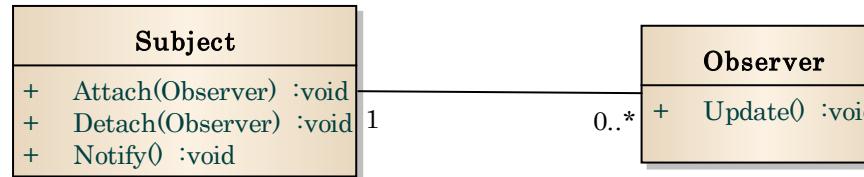


# Observer

- Name: Observer, Publish-Subscribe
- Problem: One or more objects need to be informed when another object is modified
- Solution: Define a one-to-many dependency relationship between the Subject and the Observers that depend on the state of the Subject. When the Subject state changes, automatically inform the Observers of the state change.
- Consequences: The Observers and the Subject can vary independently. It is easy to add or remove Observers.
  - There is minimal coupling between the Subject and its Observers
  - Changes to the Subject are automatically broadcast to all Observers
  - A simple change to the Subject may cause a cascade of updates to many Observers

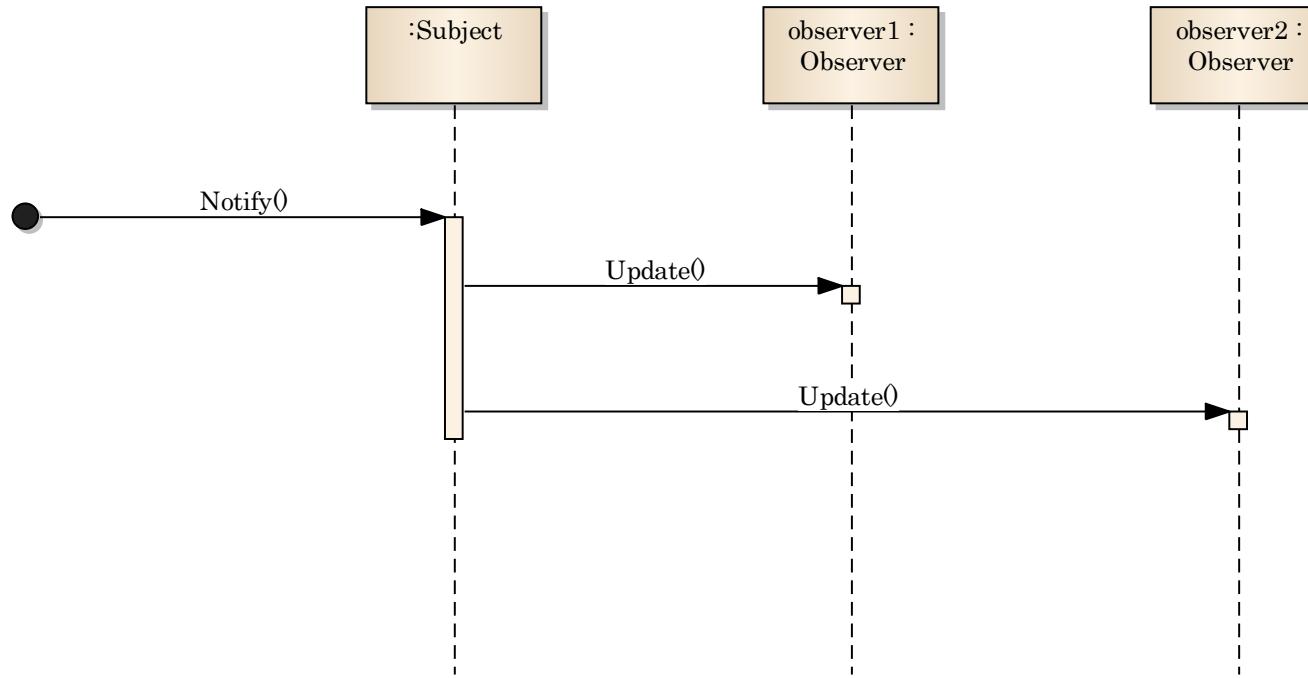
# Observer Class Diagram

- Observer pattern class diagram
  - The Subject has a one-to-many relationship with Observers



# Observer Sequence Diagram

- The Subject automatically updates all Observers when Notify is invoked

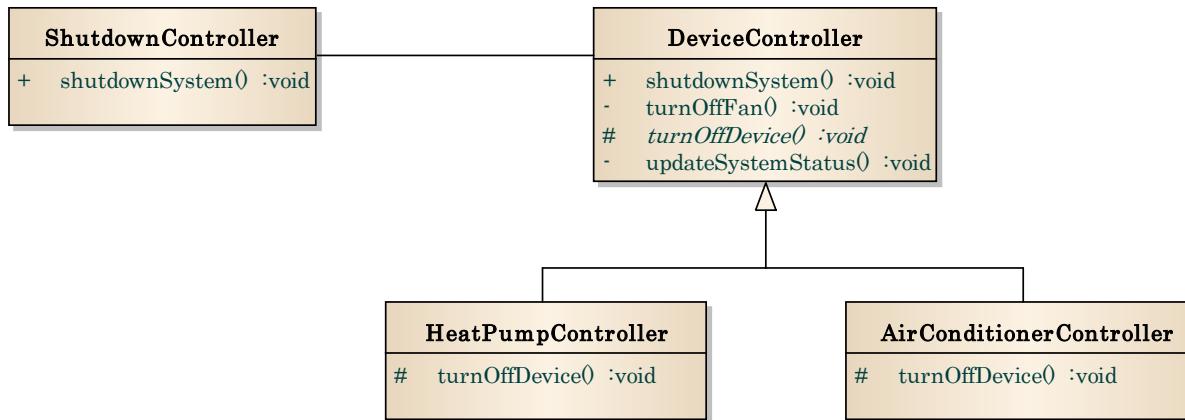


# Strategy

- Name: Strategy
- Problem: How to design a collection of related algorithms while allowing the particular strategy to vary independently of, and transparently to, the client?
- Solution: Declare a common interface for all supported algorithms. Introduce a Context object that has a reference to the selected Strategy. The client has a reference to the Context.
- Consequences:
  - The selected Strategy is hidden from the client
  - The Strategy can change dynamically without affecting the client
  - An abstract base class can replace the Strategy interface to define common Strategy elements
  - Some Strategies may not need information provided by the Context

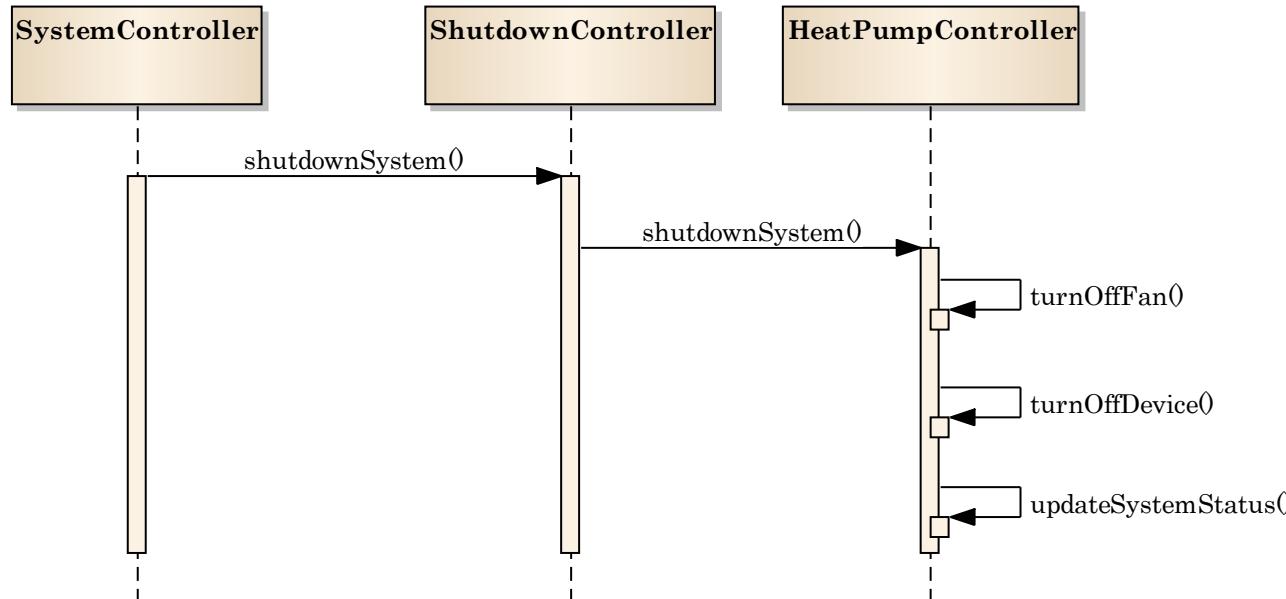
# Strategy: Class Diagram

- When using the Template pattern, the SystemController has a direct link to the selected DeviceController
  - If a different DeviceController is chosen, the SystemController will have to be updated
- To make it possible to change from HeatPumpController to AirConditioningController without having to update the SystemController, we can use the Strategy pattern



# Strategy: Sequence Diagram

- The SystemController now only depends upon the ShutdownController
  - The ShutdownController can change the Strategy without the SystemController knowing or caring which Strategy is being used



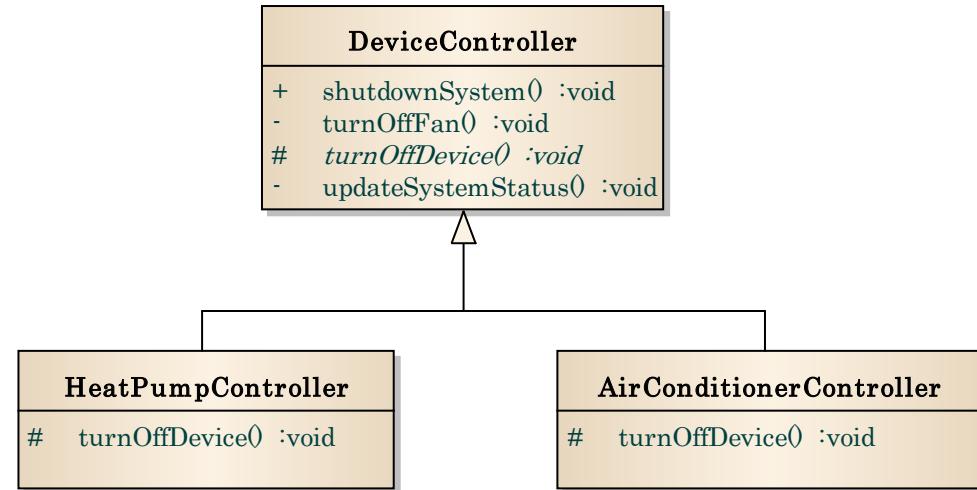
# Template

- Name: Template, Hollywood Pattern
  - “Don’t call me, I’ll call you.”<sup>1</sup>
- Problem: Several operations contain the same steps of an algorithm, specifying only a few unique steps of their own. How to ensure the common steps are always performed, and to simplify the development of the (potentially many) operations.
- Solution: Define the skeleton of the algorithm in a base class method. Include abstract methods for the steps to be defined by the derived classes. Derived classes only need to define the abstract methods.
- Consequences:
  - Derived class is considerably simpler since it only defines the abstract methods
  - Algorithm is always correctly applied, since it is defined only once in the base class
  - Derived class objects are directly used by the client

1. Comment often heard by aspiring actors in Hollywood

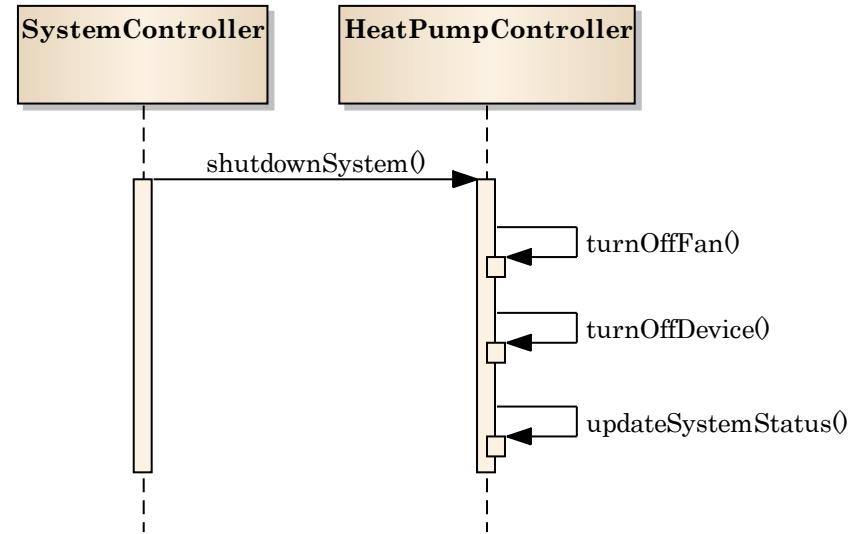
# Template: Class Diagram

- The Home Heating system now needs to support both heating and air conditioning. The process of turning off the system involves similar steps whether the system is heating or cooling.
  - However, there will be some differences in the process of turning off heating and turning off cooling
  - We can use a Template to define the common steps once in the base class
  - The subclasses can define the steps specific to each situation in the `turnOffDevice()` method



# Template: Sequence Diagram

- The HeatPumpController defines only the `turnOffDevice()` method
- All other methods are inherited from the `DeviceController` base class
- The `SystemController` calls the public `shutdownSystem()` method
  - Defined in the base class



# Data Access Object

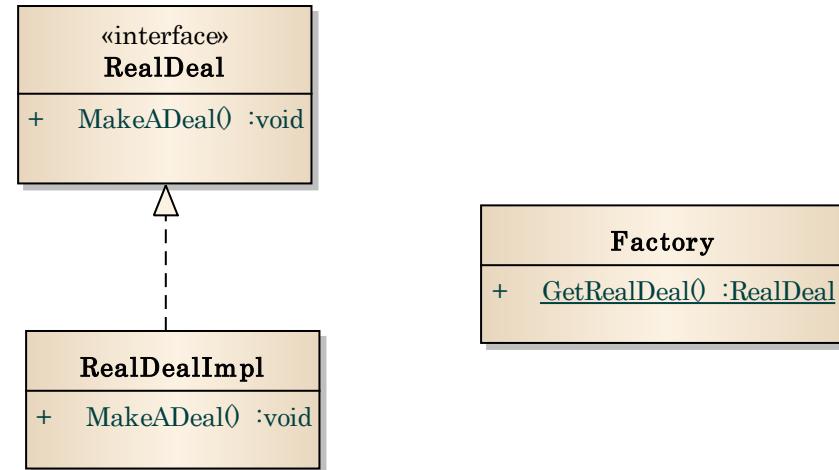
- Name: Data Access Object (DAO)
  - Not to be confused with DOA
- Problem: There are several to many places in your application that need to communicate with a data source such as a relational database
- Solution: Encapsulate the data source communication code in one object—the Data Access Object. Other parts of the program can call on the DAO to communicate with the data source
- Consequences:
  - The only part of the program that needs to know the data source communication details is the DAO
  - The rest of the program is insulated from any data source specific details

# Factory

- Name: Factory
- Problem: How to obtain a reference to an object that implements an interface without programming to the implementing class. How to encapsulate object creation.
- Solution: Use a Factory class that will create the object that implements the interface and return a reference to that object in response to client requests
- Consequences: The client can program to the interface without knowing what the actual implementation class is
  - The Factory can be written to obtain the class name from configuration information
  - The implementation class can be changed by either updating the Factory or the configuration information. The client will not be aware of any such change.

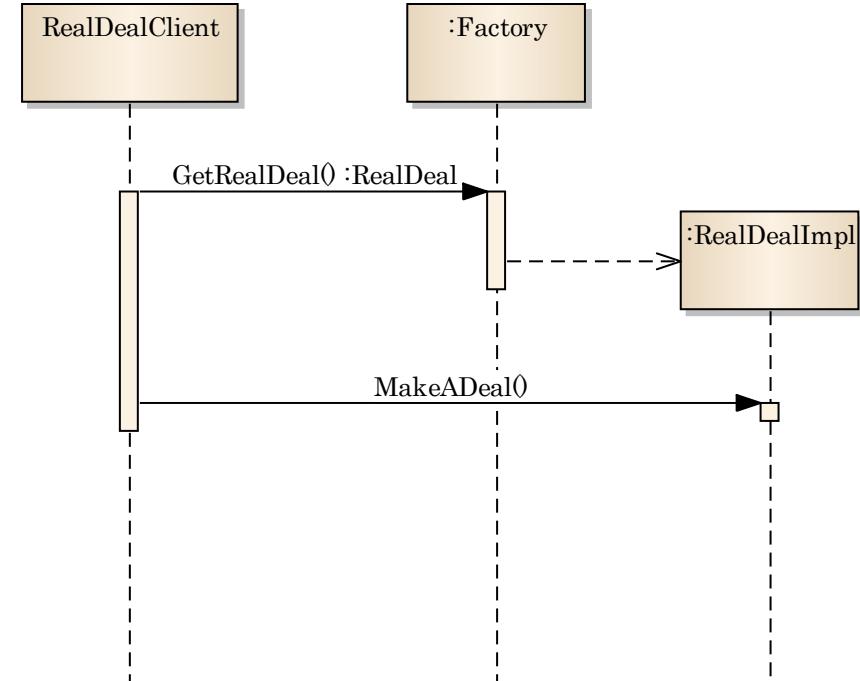
# Factory: Class Diagram

- The Factory can create a RealDeal object
  - Provides a method for clients to use
- The client can program to the interface

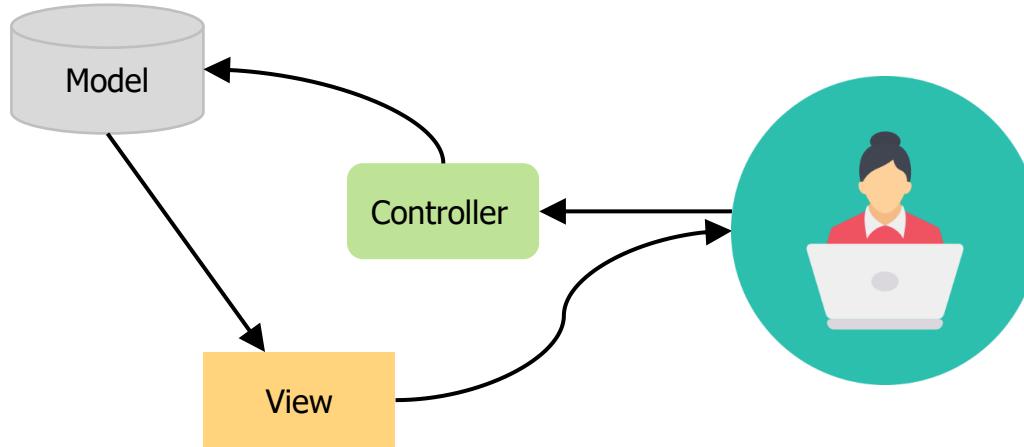


# Factory: Sequence Diagram

- The client requests a RealDeal object from the Factory
  - The Factory creates a RealDealImpl object that implements the RealDeal interface
  - The client programs to the interface
  
- The Factory decides when to create the RealDealImpl object
  - And how many to create
    - Only one for all clients to share
    - One for each client



# Model View Controller Pattern



# Model View Controller Pattern (continued)

## ■ Model View Controller (MVC) Pattern

- Originated with Smalltalk
- Divides application into three types of objects

## ■ Model

- Describes the data in the application
- Resides in the Business layer

## ■ View

- Presents the data to the user
- Interacts with the user
- Resides in the Presentation layer

## ■ Controller

- Controls the response to a user request
- Calls on objects in the Model
- Forwards to the View to display the data

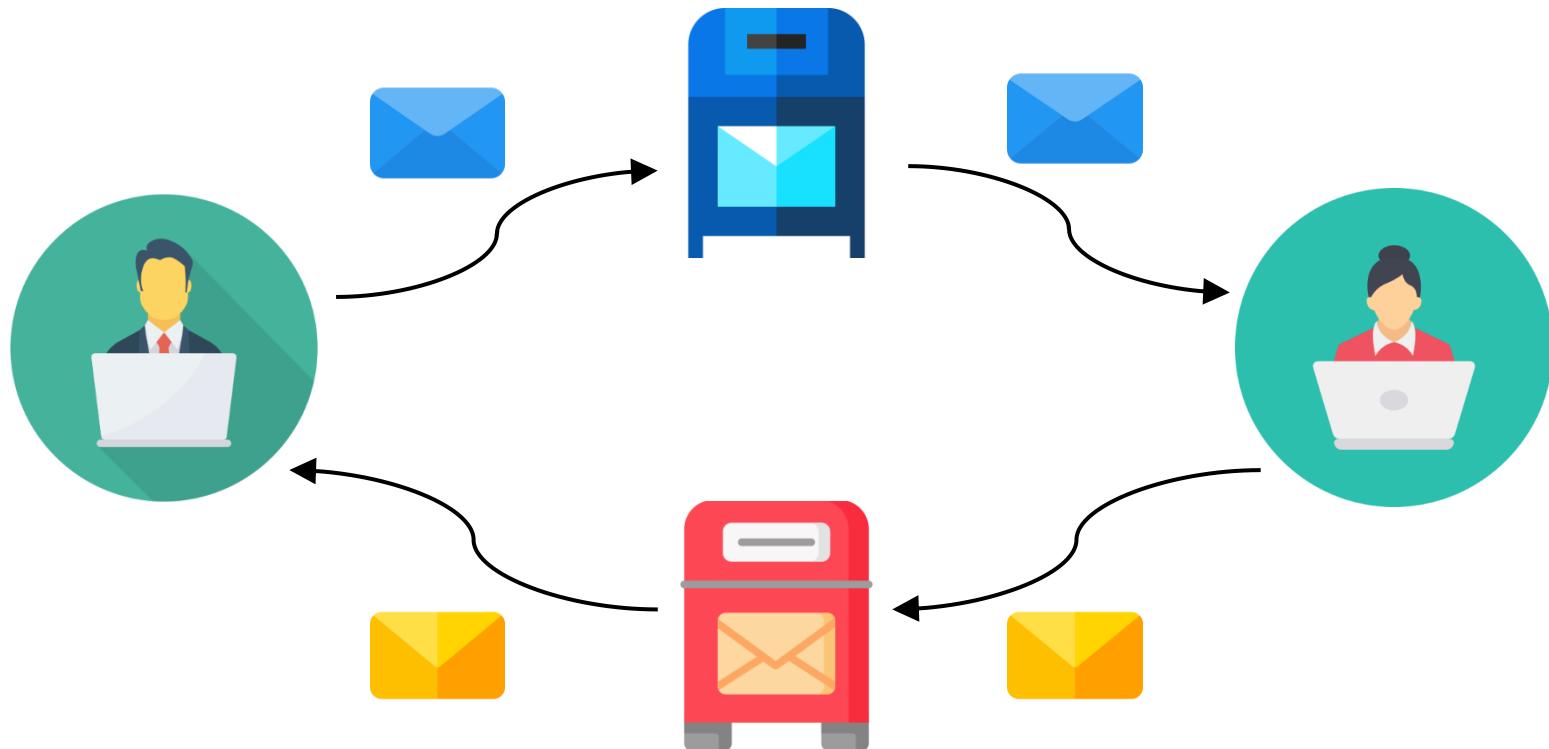
# Model-View Separation Principle

## ■ Two main parts:

1. The Model should not know about the View
  - The Model could be used with different types of Views
2. The View should not contain business rules
  - They should focus on presenting data
  - Business rules are useful in many applications

## ■ Question: What is the advantage of this principle? – Are there disadvantages when using this principle?

# Message-Based Architecture



# Message-Based Architecture (continued)

- Sender and receiver are disconnected
  - Communicate via messages
  - Sent to a message queue
- Receiver can send a response
  - By creating a new message
  - And sending to a response queue
  - The original sender can receive the response
- Good for intermittent connectivity
  - Store and forward
- Good for dissimilar clients
  - Java and .NET

# Design Pattern Resources

- [Design Patterns, Gamma, Helm, Johnson, Vlissides](#)
- [Writing Effective Use Cases, Alistair Cockburn](#)
- [UML Distilled, Martin Fowler](#)
- [Refactoring, Martin Fowler](#)
- [Refactoring to Patterns, Joshua Kerievsky](#)
- Martin Fowler (*Thoughtworks*)
  - <http://martinfowler.com/>
- [Ajax Design Patterns, Michael Mehemoff](#)