1 Introduction

The Software Design sequence

• Design algorithms
  – High-level problem-solving skills

• Implement algorithms as programs
Java - A modern programming language
– Organize data and instructions
– In both algorithms and programs, we must use low-level and precise logic
  * No ambiguity allowed
– Debugging and testing

• Understand what programs will do
• Communicate technical information about your programs
• Learn how to operate as a technical professional

Software Design I

The six things a program can do
1. Get input
2. Give output
3. Do arithmetic
4. Update a stored value
5. Test a condition, and select an alternative
6. Repeat a group of actions

Four ways Java will help you organize your work
1. Grouping related data together
2. Defining sequences of operations
3. Associating data with operations relevant to the particular data
4. Naming these groups, sequences and associations for easy and repeated use

Software Design II

• A deeper look at inheritance and object-oriented design
• Exceptions
• Recursion
• Linear (list) data structures
• Describing and tracing the effects of programs
• Programming with files and directories
• Multi-dimensional arrays
• Debugging and testing
  – *Knowing what "correct" means, and how to tell if your code meets that standard*
  – This class will have a strong focus on *test-driven development*

• Skills-based class

• Many things we learn will build on what we’ve already studied
  – Including all of Software Design I

• Practice is essential
  – Expect to work on CS220 *every day*

**Assignments**
Each assignment is...

• Posted to the course website

• Submitted through AutoLab
  – [https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmaraist/](https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmaraist/)
  – Remember the campus VPN [vpn.uwlax.edu](http://vpn.uwlax.edu)

• You should have received an email with account information on Monday
  – Check your spam filter
  – If you registered on Friday or after, I may not have your info
  – *Email* by 2pm if you still need one, and I’ll make additional accounts this afternoon

• Tomorrow’s lab will step you through a first AutoLab submission

**Textbooks**
There is no required text for this class, but you may be happier with a reference of some sort and/or a source of practice exercises. Some options:

• The CS120 online book, *Programming in Java*, zyBooks
  – Can print sections/chapters as well
  – Subscribe:
    * Sign up at [zyBooks.com](http://zyBooks.com)
    * Enter zyBook code: *UWLAXCS220MaraistFall2017*
    * Click *Subscribe*

• *Java: A Beginner’s Guide*, Herbert Schildt, Oracle Press
  – Several faculty recommend this book as a second reference for CS120

• O’Reilly has been reputable for reference books
2 JUnit and test-driven development

JUnit

• For specifying and running functional tests in Java

• A separate test for every method
  – Use Java annotations to mark the test

• The JUnit executable finds test methods, runs them, reports the results

• Eclipse will alert you to test failures

JUnit example test subject

From the JUnit wiki:

• A simple class

  ```java
  public class Calculator {
      public int evaluate(final String expression) {
          int sum = 0;
          for (String summand: expression.split("\\+"))
              sum += Integer.valueOf(summand);
          return sum;
      }
  }
  ```

JUnit example test class

• Test Calculator with

  ```java
  import static org.junit.Assert.assertEquals;
  import org.junit.Test;
  public class CalculatorTest {
      @Test public void evaluatesExpression() {
          final Calculator calculator = new Calculator();
          final int sum = calculator.evaluate("1+2+3");
          assertEquals(6, sum);
      }
  }
  ```

• The @Test annotation: how JUnit finds tests

• Method takes no parameters

• The class holding the tests has a zero-argument constructor
- Which is implicitly present if we give no constructor at all

**Assertions**

- Methods provided by JUnit for asserting things which should be true
- Parameter order: description, expected value, then actual value
  * Always give the description!
- For real-valued tests, an additional *tolerance* parameter `delta`
- Pointer equality vs. `equals`

**JUnit Assert JavaDoc**


```java
import static org.junit.Assert.*;
...
assertEquals(...);
```

**JUnit Assert methods**


| static void | `assertEquals(String message, double expected, double actual, double delta)` | Asserts that two doubles are equal to within a positive delta. |
| static void | `assertEquals(String message, float expected, float actual, float delta)` | Asserts that two floats are equal to within a positive delta. |
| static void | `assertEquals(String message, long expected, long actual)` | Asserts that two longs are equal. |
| static void | `assertEquals(String message, Object[] expecteds, Object[] actuals)` | Deprecated. use `assertArrayEquals` |
| static void | `assertEquals(String message, Object expected, Object actual)` | Asserts that two objects are equal. |
JUnit and exceptions

Can also specify tests which we expect to fail

- The Java API says that `ArrayList.get(0)` will throw an `IndexOutOfBoundsException` when the list is empty
  
  - We verify this behavior with:
    
    ```java
    @Test(expected = IndexOutOfBoundsException.class)
    public void empty() {
      new ArrayList<Object>().get(0);
    }
    ```
  

ArrayList.get exceptions

```
public E get(int index)
Returns the element at the specified position in this list.

Specified by:
get in interface List<E>
Specified by:
get in class AbstractList<E>
Parameters:
index - index of the element to return
Returns:
the element at the specified position in this list
Throws:
IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())
```

Test annotation arguments
From [junit.org/junit4/javadoc/latest/org/junit/Test.html](http://junit.org/junit4/javadoc/latest/org/junit/Test.html)
JUnit/AutoLab does not replace your other debugging skills!

• Use the main routine to dig deeper into your code

Trouble in StringCheckerTests

@Test public void testGetBasisSingle() {
    final String[] basis =
    buildSingle().getBasis();
    assertEquals(
        "Only one member of the basis", 
        1, basis.length);
    assertTrue(
        "Member of basis is \\
        + SINGLE + \\
        + "\", 
        basis[0].equals(SINGLE));
}

• Normally would not want StringBuilder to refer to StringBuilderTests
  – But this is just temporary
  – Will clear main, use again for the next problem

Investigate with StringChecker.main

public static void main() {
    final String[] basis =
    StringbuilderTests.buildSingle()
Shows us exactly what is happen in the test
  – Can add further print statements to methods
  – Just remove them/comment them out when the problem is fixed

Test-driven development
For class project we will adopt the discipline of test-driven development (TDD)
  • When coding, repeat the following steps:
    1. Write a (failing) test case
    2. Get it to compile
    3. Get it to pass
    4. Simplify and remove duplication
  • Once we have a test passing, we make sure that it keeps on passing

TDD Example
  • Working on a financial application
  • Need a class to make decisions as to whether certain commodities should be traded
    – A very mathematical question, so we’ll need to support various operations
  • Specifically, we need to calculate the first statistical moment about a particular point
  – Don’t have a method for it yet
  – But our stats experts gave us a simple example for this test:

```java
@Test public void testFirstMoment() { final
  InstrumentCalculator calc = new InstrumentCalculator();
  calc.addElement(1.0); calc.addElement(2.0); assertEquals(
    "First moment about 2.0 for {1.0,2.0} within tolerance",
    -0.5, calc.firstMomentAbout(2.0), TOLERANCE); }
```
Making testFirstMoment compile

- This won’t even compile right now
  - We might already have a class InstrumentCalculator, but we haven’t written firstMomentAbout yet!
  - For this example, let’s assume we do have the class, with a method getElements()
- So we add a stub for the method, without trying to implement it
  - We could make it return an absurd value
    ```java
    public class InstrumentCalculator {
        // ... keeping what’s already here
    
        public double firstMomentAbout(final double point) {
            return Double.NaN;
        }
    }
    
    - In our projects, we’ll usually throw an exception
      ```java
      public double firstMomentAbout(final double point) {
          throw new UnsupportedOperationException();
      }
      ```
- Whichever sort of stub we make, the test will now compile
  - But it will not yet pass

Making testFirstMoment pass

- The algorithm for calculating the first moment is standard — we look it up and implement it
  ```java
  public double firstMomentAbout(final double point) {
      double numerator = 0.0;
      for(final double element : getElements()) {
          numerator += element - point;
      }
      return numerator / elements.size();
  }
  ```
- Very often, a single test will correspond to much less code, or to only a small part of an algorithm

Write another failing test case

- The code we just added makes one test pass, but it’s not hard to conceive of cases which will fail
- There’s a division in the algorithm: are we safe against division by zero?
  - And what should happen when we call firstMomentAbout() with an empty data set?
  - Let’s say that the specification calls for an InvalidBasisException
- Write another test for this case!
  ```java
  @Test(expected = InvalidBasisException.class) public void testEmptyFirstMoment() {
      new InstrumentCalculator().firstMomentAbout(0.0);
      fail("Expected InvalidBasisException");
  }
  ```
  - fail is another JUnit method, like the assert methods, but never succeeding
Making `testEmptyFirstMoment` compile

What do we need to do to make `testEmptyFirstMoment` compile?

- If `InvalidBasisException` is not already part of `InstrumentCalculator`'s package, we must create it
- Otherwise it compiles

Making `testEmptyFirstMoment` pass

- We’ll need to throw an `InvalidBasisException` when there are zero elements
- So a revised `firstMomentAbout`:

```java
public double firstMomentAbout(final double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

    double numerator = 0.0;
    for (final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

- Now the test passes!

Write a failing test case

- Our next task is to write a routine for the second statistical moment about a point.
- So we write a test for this case. Our stats experts again gave us a simple example:

```java
@Test public void testSecondMoment() {
    final InstrumentCalculator calc = new InstrumentCalculator();
    calc.addElement(1.0);
    calc.addElement(2.0);
    assertEquals("Second moment about 2.0 for {1.0,2.0} within tolerance", 0.5,
        calc.secondMomentAbout(2.0), TOLERANCE);
}
```

Making `testSecondMoment` compile

- The problem is again that we do not define the method we are now testing
- And again we make it compile by adding a vacuous definition of the method. We’ll just copy `firstMomentAbout` and change the name:

```java
public double secondMomentAbout(final double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

    double numerator = 0.0;
    for (final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}
```
**Making testSecondMoment pass**

- Unsurprisingly, the code for the first moment does not satisfy the second moment’s test!
- But the algorithm for the second moment is very similar, and we only need to make one change: from

  ```java
  numerator += element - point;
  ```

  to

  ```java
  numerator += Math.pow(element - point, 2.0);
  ```

- And now it passes!

**Remove duplication**

- This time around there’s definitely duplication — we have two methods that are almost completely identical!
- In fact, the algorithm for any of the statistical moments has only the same variation that we see here
- So the best way to remove this duplication is with a more general private method `nthMomentAbout` which the others call

  ```java
  public double nthMomentAbout(final double point, final double n)
  { if (getElements().isEmpty()) { throw new
     InvalidBasisException(); }  
     double numerator = 0.0; for(final double element :
    getElements()) { numerator += Math.pow(element - point, n); }  
    return numerator / elements.size(); }
  ```

  ```java
  public double firstMomentAbout(final double point) { return
    nthMomentAbout(point, 1.0); }
  ```

  ```java
  public double secondMomentAbout(double point) { return
    nthMomentAbout(point, 2.0); }
  ```

- We already have tests in place, so we can be confident in this change

**It’s OK to duplicate!**

- This example seems draconian
  - Adding methods that we know are wrong!
  - Copying a method outright!
- But the point of TDD is that we are freed from worrying about more than one thing at a time
  - We might be setting up a test
– Or we might be writing code for a new feature, but *never at the same time as setting up a test*
– Or we might be refactoring away some horrible duplication, but *never at the same time as setting up a test or writing new code*
– *Do one thing at a time, and do it right*

• The tests we build up make later changes and additions much less risky

**How we’ll learn TDD**

• How does a *specification* become a *series of tests*?
  – Literally, we go sentence by sentence, translating the entire thing into a set of tests

• In the first two projects
  – I’ll give you the tests for each step
  – You’ll submit code making those tests — and only those tests — pass

• Over the semester, you’ll take responsibility for both the test and the primary code

**The various files**

• Three different roles, three different (sets of) classes and files
• All must compile together!

**Interface**

• Assignments usually specify one or more *Java interfaces*
  – Sec. 12.12 of the zyBook, or Ch. 8 of Schildt

• The methods I’ll expect your code to support
  – You should not edit the given interfaces!

**Implementation**

• Your job is to build these
• Just like what you’d write for CS120 projects

**Test classes**

• Makes sure that your interface methods perform as specified

• For early projects, given for each step
  – Run by AutoLab, available for you to run yourself
  – See the *Testing this assignment* paragraphs

• For later projects, you will write the tests yourself
3 Classes, objects, inheritance and notation

Real-life objects

This object is commonplace, and yet complicated

- It possesses some *state*
  - Including its current location, gear, current speed
- It has some *behaviors*
  - Like moving, accelerating, braking
- It *interacts with* other objects
  - Like the road, other cars, trees, people
- It is *made of* other objects
  - Like the engine, seats, tires, radio

Most of us can use it as a *black box*

- Don’t need to understand how it works
- Just need to know *what* we can do with it

Software objects

Definition

A *software object* is an entity in a program that possesses *state* (attributes) and *behavior* (actions).

- May interact with other objects
- May be composed of other objects
- Can be treated as a black box

Definition

*Object-oriented programming* (OOP) is a programming paradigm that uses the concept of *objects* to model entities. A program consists of interactions between some number of objects.
Object instances & classes
For our programs:

• We do want to be able to use many objects (which may be similar or different).
• We don’t want to have to rewrite lots of code.

In Java, we can achieve this by:

• Providing a template (blueprints) for all objects of the same type (group or kind) in a class, which specifies
  – attributes (data members), or properties of an object, and
  – behavior (actions) that an object can do

• Creating separate instances (objects) of that class to use in our programs

Class diagrams
Definition
A class diagram describes a class and how it can be used properly.

• Sketch of attributes and behaviors for objects of that type
• No details about how it works

<table>
<thead>
<tr>
<th>Class Name</th>
<th>attributes(instance variables)</th>
<th>operations(methods)</th>
</tr>
</thead>
</table>

Class diagrams - attributes

• The Car class has two simple attributes — used to store information about an individual car.

• Each different Car object (instance) in a program has its own attributes (instance variables).

• In Java, attributes correspond to fields
Class diagrams - private attributes

- Both of the Car class instance variables have private access visibility.
  - Information only accessible from within the class
- In Java, private attributes correspond to private fields
- How can such private variables be read or written?

What do the minus signs mean?

Class diagrams - instance methods

- **Update** methods (Mutators)
  - Change something about the state of an object
  - Often void methods: do an action but don’t give output
- **Query** methods (Accessors)
  - Tell us something about the state of the object
  - Methods which are not void: give information back

Class diagrams - public methods

- These methods all have public access
- Can be used outside of the class (from other code)
- Allows programmer to control how objects get modified and what information is revealed
Class diagrams - constructors

- Used to create an object
- Has the same name as the Class
- Has no (explicit) return type
- Almost always public (Why?)

The constructor here takes two parameters as input

Implementing the class

Creating and using objects in a program

- Creation:

  ```java
  // GENERAL
  final TYPE objectRef = new CONSTRUCTORNAME(params);
  
  // CONCRETE
  final Car myCivic = new Car("Honda Civic", 214118);
  ```

- Ask the object to perform an action:

  ```java
  // GENERAL
  objectRef.methodName(params);
  
  // CONCRETE
  myCivic.getMakeModel();
  myCivic.setMileage(myCivic.getMileage() + 1);
  ```
Object-oriented design

Now that we have our Car class, we can build more complex classes that use cars.

- Let’s create a CarLot that tracks cars (such as for a used car dealership)

<table>
<thead>
<tr>
<th>CarLot</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Car car1 - Car car2</td>
</tr>
<tr>
<td>«constructor» + CarLot()</td>
</tr>
<tr>
<td>&lt;query&gt; + void printCars()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>- String makeModel - int mileage</td>
</tr>
<tr>
<td>«constructor» + Car(String, int)</td>
</tr>
<tr>
<td>&lt;update&gt; + void setMakeModel(String)</td>
</tr>
<tr>
<td>+ void setMileage(int)</td>
</tr>
<tr>
<td>&lt;query&gt; + String getMakeModel()</td>
</tr>
<tr>
<td>+ int getMileage()</td>
</tr>
</tbody>
</table>

- The CarLot is an aggregate class, made up of other objects
- Relationships between classes can be signified on UML
  - Diamond arrow is a "used by" relationship

Running a Java program

A CarLot will create Car objects, but what creates the CarLot itself? The solution is to use static methods (and variables).

- Content marked static is independent of any object instance
- Usually associated with the class itself

```java
public class CarLot {
    private Car car1;
    private Car car2;

    public static void main(String[] args) {
        CarLot myCarLot = new CarLot();
    }

    public CarLot() {
        car1 = new Car("Honda Civic", 214118);
        ...
    }

    The main method
    In Java, the main method has special significance
    - Provides a point of entry for starting a program
      - Must be public and static
    - Any class can have a main method
    - Must have proper signature (including String array param)```
• In OO paradigm, `main` typically creates a top-level object and invokes a method which then takes over

```java
public class CarLot {
    ...
    public static void main(String[] args) {
        CarLot myCarLot = new CarLot();
        myCarLot.manage();
    }
    ...
    public void manage() {
        // Most of program functionality goes here
    }
}
```

**Adding more complexity**

Suppose the car dealership also sells commercial vehicles (trucks/vans) which have varying carrying capacities (e.g., 1/5/10 tons).

**One solution: Create a separate class**

<table>
<thead>
<tr>
<th>Car</th>
<th>Truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>- String makeModel - int mileage</td>
<td>- String makeModel - int mileage - int capacity</td>
</tr>
<tr>
<td>«constructor» + Car(String, int)[0.5em] «update» + void setMakeModel(String) + void setMileage(int)[0.5em] «query» + String getMakeModel() + int getMileage()</td>
<td>«constructor» + Truck(String, int, int)[0.5em] «update» + void setMakeModel(String) + void setMileage(int) + void setCapacity(int) «query» + String getMakeModel() + int getMileage() + int getCapacity()</td>
</tr>
</tbody>
</table>

18
Finding a better solution

Our first solution has some problems:

• Large amount of duplication
• Harder to write general-purpose code

Certain types of objects have things in common

• Cars/trucks/motorcycles
• Savings/checking/investment accounts

We should adjust our model to exploit these commonalities.

• Done via inheritance in Java

Inheritance

Inheritance is when one class (the subclass or child class) is based on another class (the superclass or parent class), which the child class extends or modifies in some way.

• Superclass (or parent) contains similarities
• Subclass (or child) extends the parent
  – Inherits methods and variables from the parent
  – Can add more methods and variables or modify existing ones

Allows us to make our code simpler and more useful!

Inheritance in UML and Java

Inheritance can be represented in UML with arrows from children to parents

• Each child is a more specific kind of parent object
• Called an is-a relationship

```java
public class Vehicle {
    // data and methods
}
public class Car extends Vehicle {
    // more data and methods
}
public class Truck extends Vehicle {
    // more data and methods
}
```
The **Vehicle** superclass

The # indicates *protected* access

- Like public: accessible within any child subclass (and further descendants)
- Like private: not accessible from other classes

Variables and methods in Vehicle are *inherited* by all descendants

Implementing the **Vehicle** class

Other than the protected variables and the change of names, the code is *identical* to the Car class.

- Make sure to include comments in your code!

The **Car** subclass

Every method and variable from Vehicle is *inherited* by Car

- *Except* for the constructor

A child’s constructor is responsible for calling the parent constructor.
• Called via super

    super();

• Must be the first action in the child constructor!
  – Call can be explicit or implicit (no arguments)

• Can also reference variables and methods in parent class

    super.methodName();
    super.variableIdentifier;

(Re-)Implementing the Car class

The Car constructor simply takes its inputs and "passes them up" to the Vehicle constructor via the super call. The Vehicle constructor is responsible for assigning the values to the instance variables.

public class Car extends Vehicle {
    /**
     * Constructor for Car
     * @param makeModel The make and model of the Car
     * @param mileage The miles on the car
     */
    public Car(String makeModel, int mileage) {
        super(makeModel, mileage);
    }
}

Implementing the Truck class

Inheritance structures

Objects can be part of an inheritance hierarchy, with multiple levels of ancestors and descendants.

• In Java, everything is descended from the Object class
Inheritance structures

Objects can be part of an inheritance hierarchy, with multiple levels of ancestors and descendants.

- In Java, everything is descended from the Object class

Type conformance

Every object conforms to the types of all its ancestors.

- In Java, everything conforms to the Object type

Conformance to interfaces

Interfaces specify methods without (normally) defining their bodies

- The names of the method
- The number of parameters each method has
- The type of each parameter
A class can implement one or more interfaces

- The class is then required to define a body for each interface method

Interfaces allow

- The description of what methods will be available to be separated from how they will be implemented
- Different implementations of the same method by different implementing classes

A vehicle interface

Every object conforms to both its ancestor classes and ancestor interfaces

```java
public interface VInterf {
    public String getMakeModel();
    public String drive();
}
```

```java
public class Vehicle implements VInterf {
    protected String makeModel;
    protected int mileage;
    // ...
}
```

**Polymorphism**

Polymorphism is the occurrence of something in several different forms.

- When you declare a variable, you assign its type
- As your program executes, a polymorphic variable can appear to change type, based upon the object it is currently referencing

```
Vehicle myVehicle;
myVehicle = new Car("Honda Civic", 214118);
myVehicle = new Truck("Ford F-150", 0, 2);
```

- For each assignment, the compiler checks if the type of the assigned object conforms to that of the variable.
- At runtime, the actual type of the object being referenced determines how it behaves. (Virtual method invocation)
Using polymorphism

Once we have a set of sub-classes of a common class, we can do things like create an array of objects of different subclass types

- Just as with variables, the array stores references to objects, not the objects themselves

```java
Vehicle v1, v2, v3, v4;
v1 = new Car("Honda Civic", 214118);
v2 = new Car("Saturn S-Series", 163518);
v3 = new Truck("Ford F-150", 1234, 2);
v4 = new Truck("Mack Truck", 300, 20);
Vehicle[] lot = new Vehicle[4];
lot[0] = v1;
lot[1] = v2;
lot[2] = v3;
lot[3] = v4;
```

- Each Vehicle variable stores a reference to an object that conforms to Vehicle
- Generic Vehicle array contains references to two different types of objects

Using polymorphism

Polymorphism allows us to write methods that work with multiple types of objects

```java
public class CarLot {
    private Vehicle[] lot;
    public CarLot() {
        lot = new Vehicle[4];
        lot[0] = new Car("Honda Civic", 214118);
        // ...
        printVehicles();
    }
    private void printVehicles() {
        for (int i = 0; i < lot.length; ++i) {
            System.out.println(lot[i].getMakeModel());
        }
    }
}
```

- Since every object referenced in the array conforms to Vehicle, they will all have access to the inherited getMakeModel method

Using polymorphism

For an array of type Vehicle, we can only use the objects in that array in ways that are possible for Vehicle objects

```java
public class CarLot {
    private Vehicle[] lot;
```
public CarLot() {
    lot = new Vehicle[4];
    lot[0] = new Car("Honda Civic",
    214118);
    // ...
    printVehicles();
}

private void printVehicles() {
    for(int i=0; i<lot.length; ++i) {
        System.out.println
          (lot[i].getCapacity());
    }
}

• Error: Vehicle does not have a getCapacity method
  – Every Truck is necessarily a Vehicle, but
  – Not every Vehicle is necessarily a Truck

4 Arrays

Arrays

An array is a primitive data structure for storing multiple objects
• All elements of the array must have the same type
• The length of the array is fixed at its creation, and never changes
• Each position in the array stores a single element
• Each element is referenced by its index in the array

Basic syntax

• Declare an array (does not allocate memory):

  final dataType[] arrayName;
  
  – Alternative syntax
  
  final dataType arrayName[];
• Allocate memory for a previously declared array:

    arrayName = new dataType[numberOfElements];

    Size cannot be negative

• Store and retrieve values in array:

    arrayName[index] = expression;  // Store value at index
    arrayName[index];               // Retrieve value from index

• Access the length of an array:

    arrayName.length

    Not the same as the method call for String — str.length()

• One-liners for declaration, allocation, and initialization:

    final dataType[] arrayName = new dataType[ numberOfElements ];
    final dataType[] arrayName = { val1, val2, ..., valN };  // Initialize

Basic examples

• Declare an array (does not allocate memory):

    double[] numbers;    /* Alternate: */ double numbers[];

• Allocate memory for a previously declared array:

    numbers = new double[10];

• Store and retrieve values in array:

    numbers[3] = 7.5;
    System.out.print(numbers[3]);

• Access the length of an array:

    numbers.length

• One-liners for declaration, allocation, and initialization:

    double[] numbers = new double[10];
    double[] numbers = { 1.5, 4.5, 7.5, ..., 15.2 };
Arrays of primitive types

When using arrays, we need to ensure:

- Array variable is declared
- Memory is allocated for the array (using `new`)
- Contents of the array have been initialized

With primitive type:

```java
final int[] intArray = new int[5];
for(int i=0; i<intArray.length; ++i) {
    System.out.print(intArray[i] + " , ");
}
```

- Output:

```
0, 0, 0, 0, 0,
```

This works even though we skipped Step 3 – Java takes care of the initialization for us.

Simpler iteration

```
for loops make it easy to work with arrays
```

```java
final int[] intArray = new int[5];
for(final int j : intArray) {
    System.out.print(j + " , ");
}
```

Arrays of objects

We can also have arrays of complex type:

```java
final Car[] carArray = new Car[5];
for(int i=0; i<carArray.length; ++i) {
    System.out.println(carArray[i]);
}
```

- Output is

```
null
null
null
null
null
```

Why does this fail? Need to initialize array contents!

- Java doesn’t know how to initialize the objects
Example: all Civics

```java
final Car[] carArray = new Car[5];
// We need to initialize the objects in a sensible way
for(int i=0; i<carArray.length; ++i) {
    carArray[i] = new Car("Honda Civic", 1000 * i);
}
for(int i=0; i<carArray.length; ++i) {
    System.out.println(carArray[i]);
}
```

Arrays of objects

```java
final Person[] simpsons = new Person[3];
simpsons[0] = new Person("Homer", "D’oh!");
simpsons[1] = new Person("Flanders", "Okily Dokily!");
```

In an array of complex type (i.e., class), each element in the array stores a *reference* to an object of that class

- Does *not* store the object itself (just like a variable of complex type)
- We need to *instantiate* an object for each element of the array

Arrays and methods

```java
public static void main(String[] args) {
    int[] myArr = { 1, 2, 3 }; 
    squareArray(myArr); 
    System.out.println(myArr[2]); 
}

public static void squareArray(int[] arr) {
    for (int i = 0; i < arr.length; ++i) {
        arr[i] = arr[i] * arr[i];
    }
}
```

When an array is passed to a method, only its *reference* is passed (just like objects)

- Any modifications that the method does to the array contents persist after the method ends
Example
When an array is passed to a method, only its reference is passed (just like objects)

- The update to `arr` does not change what `myArr` references

```java
public static void main(String[] args) {
    int[] myArr = { 1, 2, 3 };    // myArr references { 1, 2, 3 }
    modifyArray(myArr);            // arr references { 7, 2, 9 }
    System.out.println(myArr[2]);  // 9
}

class Array1DExample {
    public static void modifyArray(int[] arr) {
        arr[0] = 7;
        arr = new int[3];            // arr references { 0, 0, 9 }
        arr[2] = 9;
    }
}
```

Multi-Dimensional Arrays
In Java, arrays can be extended to more than one dimension.

- A one-dimensional array:
  ```java
  int[] arr1d = new int[6];
  arr1d[3] = 7;
  ```

- A two-dimensional array:
  ```java
  int[][] arr2d = new int[3][5];
  arr2d[1][2] = 4;
  ```

- Accessing dimensions:
  ```java
  int[][] matrix = new int[7][10];
  int numRows = matrix.length; // Returns 7
  int numCols = matrix[0].length; // Returns 10
  ```

```
```
Using multi-dimensional arrays

Multi-dimensional arrays are useful for storing data that has *multiple indices*

• That is, "keys" to look it up

For example, storing movie reviews across users

```java
final int numPeople = 3;
final int numMovies = 5;
final int[][] ratings =
    new int[numPeople][numMovies];

// ...

ratings[0][3] = 5;
```

<table>
<thead>
<tr>
<th>reviewer (1st index)</th>
<th>movie (2nd index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 1</td>
<td>5</td>
</tr>
<tr>
<td>[1] 3</td>
<td>1</td>
</tr>
<tr>
<td>[2] 4</td>
<td>5</td>
</tr>
</tbody>
</table>

Multi-dimensional arrays

... and on to higher dimensions

• A one-dimensional array

```java
int[] arr1d = new int[6];
arr1d[3] = 7;
```

• A two-dimensional array

```java
int[][] arr2d = new int[3][5];
arr2d[1][2] = 4;
```

• A three-dimensional array

```java
int[][][] arr3d = new int[2][2][4];
arr3d[0][1][2] = 6;
```

First index is like the page number of a notebook

• And so on
Using multi-dimensional arrays

Another example: hourly temperatures for a weather station over 3 years

```java
int years = 3;
int days = 365;
int hours = 24;
double[][][] temps =
    new double[years][days][hours];
```

Storing temperature of $-1.2$ for Year 2 of 3, January 01, at 12 noon:

```java
temps[1][0][12] = -1.2;
```

Using multi-dimensional arrays

Just as a single `for` loop is useful for manipulating a one-dimensional array, nested `for` loops are useful for manipulating an $n$-dimensional array

- One loop per dimension

```java
int[][] arr2d = new int[20][15];
```

```java
for(int row=0; row<20; ++row) {
    for(int col=0; col<15; ++col) {
        final int n = arr2d[row][col];
        System.out.print(n + " ");
    }
    System.out.println();
}
```

- `row` loops over the first dimension
- `col` loops over the second dimension
- This code does work, but what is `wrong` with it?
Avoid "magic numbers" in code

Hard-coding values leads to fragile code

- Difficult to maintain, hard to debug

Arrays of arrays

A two-dimensional array is actually an array of arrays!

```
// Allocate space for 10 references to int[]
final int[][] arr2d = new int[10][];

// Allocate space for each "row"
for(int i=0; i<arr2d.length; ++i) {
    arr2d[i] = new int[5];
}
```

- `arr2d` is a variable that contains a reference to an array
  - `arr2d.length` gives size of this array
  - `arr2d[i]` gives element at position `i`

- `arr2d[i]` stores a reference to another array
  - `arr2d[i].length` gives size of this other array
  - `arr2d[i][j]` gives element at position `j` in this other array

Ragged arrays

What happens if we make this change to the array builder?

```
// Allocate space for 10 references to int[]
final int[][] arr2d = new int[10][];

for(int i=0; i<arr2d.length; ++i) {
    // Allocate space for each "row"
    arr2d[i] = new int[i+1];
}
```
Creates a *ragged* array (as opposed to a rectangular array)

![Ragged Array Diagram]

Creating multi-dimensional arrays

- Creating a rectangular two-dimensional array:

  ```java
  // Allocate all space for array at once
  int[][] matrix = new int[5][4]; // 5 rows and 4 columns

  // Shortcut initialization: 2d array with 2 rows and
  // 3 columns
  int[][] matrix = {{3, 5, 6}, {2, 4, 7}};
  ```

- Creating a ragged two-dimensional array:

  ```java
  // Allocate memory for each row separately
  int[][] matrix = new int[3][]; // 3 rows
  matrix[0] = new int[5]; // 5 columns in row 0
  matrix[1] = new int[3]; // 3 columns in row 1
  matrix[2] = new int[7]; // 7 columns in row 2

  // Shortcut init.: 2 rows with 2 and 4 cols, respectively
  int[][] matrix = {{2, 4}, {7, 3, 5, 6}};
  ```

Writing some code

Assuming that the matrix in the skeleton below is initialized, write the code necessary to multiply every entry by scalar.

```java
double[][] matrix;
// Assume matrix initialized here
double scalar = ...;

// Your code here...
```

Writing some more code

Write a public static non-void method named `matrixContains` that takes a 2-dimensional array of integers and an integer and returns `true` if the matrix contains that value, `false` otherwise.

```java
public static boolean matrixContains(int[][] matrix,
        int value) {
```
4.1 An ADT for resizable lists

Memory in Java

For primitive arrays in Java, we have the following requirement:

- The length of the array is fixed at creation (it never changes)

Why?

Consider the following:

```java
int[] arr1 = new int[3];
int[] arr2 = new int[4];
arr1.resize(5); // NOTE: Not an actual method for arrays!
```

A workaround?

This limitation can be problematic, as we may not know up front how many things we need to store.

- Allocate extra space for every array to reduce the likelihood of running out of room
- Create an overflow array to store additional items if/when first array is filled
- Create and use a larger array that can store old items plus new ones

```java
final int[] array = new int[5];
// .. intermediate work

// What should newSize be?
final int[] newArray = new int[newSize];
for(int i=0; i<array.length; ++i) {
    newArray[i] = array[i];
}
array = newArray;
```
Resizing an array?

Resizing an array is a common thing to want to do. Can we make a method for it?

```java
public static void resizeArray(int[] array, int newSize) {
    final int[] newArray = new int[newSize];
    for (int i = 0; i < array.length; ++i) {
        newArray[i] = array[i];
    }
    array = newArray;
}
```

But this won’t work

```java
int[] myArr = new int[20];
resizeArray(myArr, 30);
myArr[25] = 42; // Error here - index out of bounds
```

The method can change the contents of the array that `myArr` points to, but it is unable to change the memory location that `myArr` references

Resizing an array: Take 2

Resizing an array is a common thing we might want to do, so let’s make a method for it

```java
public static int[] resizeArray(int[] array, int newSize) {
    int[] newArray = new int[newSize];
    for (int i = 0; i < array.length; ++i) {
        newArray[i] = array[i];
    }
    return newArray;
}
```

Now we can do the following:

```java
final int[] myArr = new int[20];
myArr = resizeArray(myArr, 30);
myArr[25] = 42; // This works!
```

The method returns a reference to the new array, which can be used to update `myArr`

Resizing an array: time for Take 3?

The following scenario still poses a problem:

```java
public static void main(String[] args) {
    int[] array = new int[5];
    populateList(array);
    printList(array);
}
```

```java
public static void populateList(int[] array) {
    // work ..
    array = resizeArray(array, 10);
}
```
// more work...
)

public static void printList(int[] array) {
    for (int i = 0; i < array.length; ++i) {
        System.out.println(array[i]);
    }
}

A better solution
Add a layer of abstraction between the array contents and those who need to use it!

All problems in computer science can be solved by another level of indirection. — David Wheeler

Abstract data types
An abstract data type is a model that defines data types in terms of their behavior (what can be done with it).

What this means for the user:
• Do need to know what they can do with the data type
• Don’t need know how the data type is implemented

What does this remind you of?
Example: the String class in Java
• We manipulate String objects using public methods
• We don’t need to know the underlying representation of the characters to use it (though it can be helpful!)

Defining an abstract data type
Let’s consider our basic needs for a resizeable array:
• Add items (at end or at specific position)
• Access items
• Get number of items
We can abstract these needs as a list

```
MyList
```

```markdown
«constructor» +
MyList() «update»
+ void add(String) + void
add(String, int) «query» + String get(int)
+ int size()
```

Implementation

```
public class MyList {
    public MyList() { // ...
    }
    public void add(final String item) {
        // ...
    }
    public void add(final String item,
        final int index) {
        // ...
    }
    public String get(final int index) {
        // ...
    }
    public int size() { // ...
    }
}
```

First step — the empty list

The test

```java
@Test public void emptyList() {
    final MyList ml = new MyList();
    assertEquals(0, ml.size());
}
```
Making it pass

private String[]
    strings = new String[0];

public MyList() { }

public int size() {
    return strings.length;
}

One piece of actual stuff

The test

@Test public void
    appendingStuff() {
        final MyList
            ml = new MyList();
        ml.add("stuff");
        assertEquals
            (1, ml.size());
        assertEquals
            ("stuff",
            ml.get(0));
    }

Making the test pass

private String[]
    strings = new String[0];
public void add(final String item) {
    final int oldLen = size();
    final String[] newStrings
        = new String[1+oldLen];
    for(int i=0; i<oldLen; ++i) {
        newStrings[i] = strings[i];
    }
    newStrings[oldLen] = item;
    strings = newStrings;
}

Reading the empty list

In fact we could strengthen the first test a bit:

@Test(expected=ArrayIndexOutOfBoundsException.class)
public void emptyList() {
    final MyList ml = new MyList();
Is there extra work for multiple appends?

The test

We can add to the appendingStuff test:

```java
lst.add("thing2");
assertEquals("List at length 2",
    2, lst.size());
lst.add("thing3");
assertEquals("List at length 3",
    3, lst.size());
assertEquals("Get what you give #2",
    "thing2", lst.get(1));
assertEquals("Get what you give #3",
    "thing3", lst.get(2));
```

It works as is!

Our `add(String item)` method works just fine for these additional tests.

What about the other `add` method?

The test

We’ll start the same way we did with the appending `add`

```java
@Test public void insertingStuff() {
    final MyList lst = new MyList();
    lst.add("stuff", 0);
    assertEquals(1, lst.size());
    assertEquals("stuff", lst.get(0));
}
```

What about the other `add` method?

But what’s really important is that we can insert from either side

```java
@Test public void insertingLeft() {
    final MyList lst = new MyList();
    lst.add("thing1", 0);
    ```
```java
assertEquals(1, lst.size());
assertEquals("thing1",
    lst.get(0));

lst.add("thing2", 0);
assertEquals(2, lst.size());
assertEquals("thing1",
    lst.get(1));
assertEquals("thing2",
    lst.get(0));
}

@Test public void insertingRight() {
    final MyList
        lst = new MyList();
lst.add("thing1", 0);
    assertEquals(1, lst.size());
    assertEquals("thing1",
        lst.get(0));

lst.add("thing2", 1);
    assertEquals(2, lst.size());
    assertEquals("thing1",
        lst.get(0));
    assertEquals("thing2",
        lst.get(1));
}

The inserting add

• Remember the old array
• Set up the new array
• Old elements before index have the same position
• Old elements after index shift right
• The new element’s insertion point
• Tests pass!
• Tests for more than two elements?

public void add(final String item,
    final int index) {
    final int oldSize = size();
    final String[] oldStrings = strings;
    strings = new String[1+oldSize];

```
for(int i=0; i<index; i++) {
    strings[i] = oldStrings[i];
}

for(int i=oldSize; i>index; i--) {
    strings[i] = oldStrings[i-1];
}

strings[index] = item;

A simplification
Do we really need two separate implementations for the two add methods?

• Can one method just use the other?
  • add(String item, int index) is more general
  • So can we replace the other add’s body with a call to the more general method?

public void add(final String item) {
    add(item, size());
}

• We have tests already in place
  – And they tell us this simplification is correct!

Stepping back
What’s good about this class

• We now have a resizing array list
  – Much simpler than the earlier approaches
  – Details of reallocations nicely hidden away

• Others can use it easily as well

What’s bad about this class

• Doesn’t support removal
• Reallocates array on every addition
• Can only store String objects

Capacity vs. use
We can take care of two concerns with the same idea

• The size of the underlying array, and
• The number of spaces of the array actually in use
do not have to be the same!
Separating capacity and use

Fields

private String[] strings
    = new String[0];
private int used=0;

public int size() {
    return used;
}

protected int capacity() {
    return strings.length;
}

The add method

public void add(final String item,
    final int index) {
    final String[] oldStrings = strings;
    final int oldSize = size();
    if (oldSize == capacity()) {
        strings = new String[oldSize+INCREMENT];
        for(int i=0; i<index; i++) {
            strings[i] = oldStrings[i];
        }
    }
    for(int i=oldSize; i>index; i--) {
        strings[i] = oldStrings[i-1];
    }
    strings[index] = item;
    used += 1;
}

Allocate more than one slot at a time

• How many slots should we allocate at a time?

    private static final int INCREMENT=10;
    private String[] strings = new String[INCREMENT];

        – Note no magic numbers!

• In add, change the reallocation line:

    strings = new String[oldSize+INCREMENT];

• We check our tests to make sure this change is good and...
They fail

- We no longer get an error from reading the empty list!
- That slot actually does exist now
- So we must explicitly check for an out-of-bounds reference

```java
public String get(final int index) {
    if (index >= size()) {
        throw new ArrayIndexOutOfBoundsException();
    }
    return strings[index];
}
```

- Now the tests pass

- Although we might white-box test by adding test cases that actually force reallocations

Removal isn’t that hard anymore

The test

```java
@Test public void remove(int index) {
    final MyList lst = new MyList();
    lst.add("thing1");
    lst.add("thing2");
    lst.add("thing3");
    lst.remove(1);
    assertEquals(2, lst.size());
    assertEquals("thing1", lst.get(0));
    assertEquals("thing3", lst.get(1));
}
```

Removal isn’t that hard anymore

The test Three tests

Beginning, middle, and end of the list

```java
@Test public void removeFirst() {
    final MyList lst = new MyList();
    lst.add("thing1");
    lst.add("thing2");
    lst.add("thing3");
    lst.remove(0);
    assertEquals(2, lst.size());
    assertEquals("thing2", lst.get(0));
    assertEquals("thing3", lst.get(1));
}
```

```java
@Test public void removeMiddle() {
    final MyList lst = new MyList();
    lst.add("thing1");
    lst.add("thing2");
    lst.add("thing3");
    lst.remove(1);
    assertEquals(2, lst.size());
    assertEquals("thing1", lst.get(0));
    assertEquals("thing3", lst.get(1));
}
```

```java
@Test public void removeLast() {
    final MyList lst = new MyList();
    lst.add("thing1");
}
```
Making the test pass

```java
public void remove(final int index) {
    if (index >= size()) {
        throw new ArrayIndexOutOfBoundsException();
    }
    for(int i=index; i<size()-1; i++) {
        strings[i] = strings[i+1];
    }
    used -= 1;
}
```

- Why is the bound `size()-1` and not `size()`?
- And the tests compile and pass again

**Lists in Java**

In Java:

- The idea of a list and its operations is specified in the `List` interface
- Concrete implementations of the list abstract data type are provided by several classes, in particular:
  - `ArrayList`
  - `LinkedList`

  - Both interface and implementations are generic to allow for arbitrary types of objects to be stored

```java
ArrayList<TYPE> myList = new ArrayList<TYPE>(); // Fill in TYPE
List<TYPE> myList = new ArrayList<TYPE>(); // Fill in TYPE
ArrayList<String> myList1 = new ArrayList<String>();
List<String> myList2 = new ArrayList<String>();
```

**ArrayList in Java**

The `List` interface and classes support several basic operations:

```java
ArrayList<String> list = new ArrayList<String>();
list.add("Hello"); // contents: {"Hello"}
list.add("World!"); // contents: {"Hello", "World!"}
```
list.add(1, "Blue"); // contents: {"Hello", "Blue", "World!"}

list.contains("Blue"); // returns true
list.set(1, "Green"); // contents: {"Hello", "Green", "World!"}

String temp = list.get(1); // returns "Green"
int curSize = list.size(); // returns 3

list.clear(); // contents: {}
curSize = list.size(); // returns 0

- Along with selective remove operations
  - But the interface does not show us the internal details like (what we called) capacity

**Primitive and reference types**

The ArrayList class only supports reference types.

- This does not work:

```
ArrayList<int> list = new ArrayList<int>(); // INCORRECT!
```

Solution: another layer of abstraction!

- Integer class provides a wrapper for int
- Double class provides a wrapper for double
- And so on
- All standard in java.lang

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(42));
Integer first = list.get(0);
```

**Autoboxing and unboxing**

Having to create objects for each int we add to the list is cumbersome. We’d like to be able to do the following:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(42);
int first = list.get(0);
```

In fact, we can do just that!

- **Autoboxing** is the process by which a primitive type is automatically converted to its corresponding wrapper object.
- **Unboxing** is the process by which a wrapper object is automatically converted back to its primitive type.

See also [docs.oracle.com/javase/tutorial/java/data/autoboxing.html](docs.oracle.com/javase/tutorial/java/data/autoboxing.html)
5 Interfaces

Interfaces

An interface is a contract that specifies what something can do. The Java language already contains a large number of interfaces that can be used to add functionality to code.

```java
public interface Comparable {
    public int compareTo(Object obj);
}
```

- Methods for sorting, maintaining sorted structures

```java
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

- `ActionEvent` is representation of an event as an object
- The Java Runtime Environment creates a particular event object when the corresponding event occurs

6 Basic generic types

Generic data structures

It would be inconvenient to rewrite `MyList` every time we want to use a new type of data

- Goal: Write our list code in such a way so that it works with *any complex data type*

One possible solution: use `Object` for the list element type

- Simple and it works for any complex type
- Used in early versions of Java (1.0-1.4)
- Requires casting

```java
public class MyList {
    private Object[] items;
    public void add(final String item) {
        // ...
    }
}
```

// Use like this
final String fifth = (String)(lst.get(4));

- Tedious
- Can introduce errors in code
  - Which we do not discover until *runtime*
Searching for a better solution
Consider how we write and use a method:

- We specify formal parameters that represent *inputs* to the method
- We write the method so that it works regardless of what those inputs actually are
- When we call (invoke, use) the method, we pass in actual values (arguments) for it to process.

We would like to be able to do something similar for a class:

- Specify parameters that represent type "inputs" to the class
- Write the class so that it works regardless of what those type inputs actually are
- Provide actual types (arguments) for the type "inputs" when we use the class (at variable declarations and object instantiations)

Java generics
Generic types provide a way to do this by using the concept of a *type variable*

- Added to Java 5 in 2004
- Allows for classes and methods to be written for any complex type

Specifically, generics allow non-primitive types to be *type parameters* when defining classes, interfaces, and methods

A *generic class* is a class that is defined with one or more *type parameters* (type variables). (A class that takes “inputs”.)

```java
public class MyGenericClass<T1, T2, ..., Tn> { /* .. */ }
```

More info at [https://docs.oracle.com/javase/tutorial/java/generics/types.html](https://docs.oracle.com/javase/tutorial/java/generics/types.html)

An analogy with methods

- A method specifies input values via formal parameters

  ```java
  public static void someMethod(int var1, String var2) { /* .. */ }
  ```

  - `var1` and `var2` are formal parameters for `someMethod`
  - Can be used anywhere in the method itself where an `int` or a `String` would be used

- A generic class specifies type values via type parameters

  ```java
  public class MyGenericClass<TypeVar1, TypeVar2> { /* .. */ }
  ```

  - `TypeVar1` and `TypeVar2` are type parameters for the class
  - Can be used anywhere in the class itself where a `type` would be used
A simple example

A regular class

```java
public class Box {
    private Object data;
    public Box(Object d) {
        data = d;
    }
    public void set(Object d) {
        data = d;
    }
    public Object get() {
        return data;
    }
}
```

- Defines a type called Box

A generic class

```java
public class Box<SomeType> {
    private SomeType data;
    public Box(SomeType d) {
        data = d;
    }
    public void set(SomeType d) {
        data = d;
    }
    public SomeType get() {
        return data;
    }
}
```

- Defines a generic type which requires an input in order to be used

Generic type invocation

- Method invocation passes arguments (values) to a method

```java
public static void someMethod(int var1, String var2) { /* ... */ }
public static void main(String[] args) {
    someMethod(42, "Hello, World!");
}
```

- Method is run using 42 and "Hello, World!" for var1 and var2

- Generic type invocation passes type arguments to a generic class
public static void main(String[] args) {
   // Create parameterized type
   final Box<String> wordContainer;
   // Instantiate
   wordContainer = new Box<String>("Hello, World!");
   // Another type and instance
   final Box<Vehicle> carBox = new Box<Vehicle>( /* ... */ );
}

- Creates the parameterized types `Box<String>` and `Box<Vehicle>`
- Instantiates objects of these parameterized types

Empty angle brackets

- In Java 5 and 6, instantiating a parameterized class requires that the type argument appear twice
  
  ```java
  final Box<Car> shippingContainer = new Box<Car>(new Car("DeLorean", 1980));
  ```

- From Java 7 this was simplified
  
  - Only the first use of the type argument is necessary
    ```java
    final Box<Car> shippingContainer = new Box<>(new Car("DeLorean", 1980));
    ```
  - The empty angle brackets `<>` are sometimes referred to as the diamond operator
  - Both styles work in recent versions of Java
  - The second style is preferred
  - Must use the first if dealing with legacy code

Interfaces with type parameters

- Interfaces can also take type parameters
- You may have encountered the method `iterator`
- Gives an object which lets us see the elements of an array or list one at a time
  ```java
  final String[] myStrings;
  // Setup for myStrings omitted
  
  final Iterator<String> iter = myStrings.iterator();
  while (iter.hasNext()) {
      System.out.println(iter.next());
  }
  ```
Two generic interfaces

Under the hood, there are two generic interfaces behind this mechanism

The `Iterable` interface

```java
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

The `Iterator` interface

```java
public interface Iterator<T> {
    public T next();
    public boolean hasNext();
}
```

Making `MyList` iterable

First, of course, a test

```java
@Test public void iterator() {
    final MyList lst = new MyList();
    lst.add("thing1");
    lst.add("thing2");

    final Iterator<String> iter = lst.iterator();
    assertEquals("First element iterated first", "thing1", iter.next());
    assertEquals("Second element iterated next", "thing2", iter.next());
    assertFalse("Nothing else was in list", iter.hasNext());
}
```

A class for the iterator

```java
class MyListIterator implements Iterator<String> {
    private int index=0;
    private final int limit;
    private final String[] strings;
    public MyListIterator(int limit, String[] strings) {
        this.limit = limit;
        this.strings = strings;
    }
    public String next() {
        return strings[index++];
    }
    public boolean hasNext() {
        return index<limit;
    }
}
```
Putting it all together

```java
public class MyList
    implements Iterable<String> {

    // Rest unchanged

    public Iterator<String> iterator() {
        return new MyListIterator(used, strings);
    }
}
```

• And the test passes

**Short-form for loop**

• The `Iterable` class is behind the short-form `for` loop

• Any object of a class implementing `Iterable` can be used in these loops:

```java
for (final String s : myList) {
    System.out.println(s);
}
```

**Another look at `Comparable`**

`Comparable` is actually a generic interface

• Specify the type to which it is valid to compare

```java
public interface Comparable<T> {
    public int compareTo(T o);
}
```

• So we can compare an `Integer` to an `Integer`, but not to a `String`

7 Abstract classes

**Abstract classes**

An abstract class is a class that cannot be instantiated

• Used for creating a pattern for other classes

• Behavior can be partially specified, requiring subclasses to provide any missing details

What details might be missing? Method implementations

• An abstract method is a method that is declared but does not have an implementation

• An abstract class can have zero or more abstract methods
  – Might not have missing details
Abstract classes in Java

An abstract class or method takes the abstract keyword

```java
public abstract class Shape
    extends JComponent {
    public Shape() {
        super();
    }

    public abstract double getArea();

    public String toString() {
        return "@ (" + getX() + ",
                        + getY() + ")";
    }
}
```

- Has an abstract method without implementation details
- Has regular concrete methods with implementation details
- Still has a constructor

But abstract classes cannot be instantiated

- Compiling this code:
  ```java
  Shape s = new Shape();
  ```
  gives us:
  ```java
  Shape is abstract; cannot be instantiated
  ```

Extending abstract classes

To be used, an abstract class must be extended

Subclasses of an abstract class:
- Inherit all implemented methods of the parent
- Must either
  - Implement all abstract methods of the parent, or
  - Be declared abstract as well

```java
public class Rectangle extends Shape {
    public Rectangle(int x, int y, int w, int h) {
        super();
    }
```
```java
setBounds(x, y, w, h);
}
public double getArea() {
    return getWidth() * getHeight();
}
public String toString() {
    return "Rectangle " + super.toString();
}
}
```

- Usable constructor
- Parent's abstract method implemented
- Overriding other methods is optional

Another example: the car lot

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td># String makeModel # int mileage</td>
<td>«constructor» + Car(String, int)</td>
</tr>
<tr>
<td>«constructor» + Vehicle(String, int)[0.5em] «update» + void setMakeModel(String) + void setMileage(int)[0.5em] «query» + String getMakeModel() + int getMileage()</td>
<td></td>
</tr>
</tbody>
</table>

Making Vehicle abstract

```java
public abstract class Vehicle {
    protected String makeModel;
    protected int mileage;

    public Vehicle(String s, int m) {
        this.makeModel = s;
        this.mileage = m;
    }
    public void setMakeModel(String s) {
        makeModel = s;
    }
    public void setMileage(int m) {
        mileage = m;
    }
```
public String getMakeModel() {
    return makeModel;
}

public int getMileage() {
    return mileage;
}

public abstract class Vehicle {
    protected String makeModel;
    protected int mileage;

    public Vehicle(String s, int m) {
        this.makeModel = s;
        this.mileage = m;
    }

    public void setMakeModel(String s) {
        makeModel = s;
    }

    public void setMileage(int m) {
        mileage = m;
    }

    public String getMakeModel() {
        return makeModel;
    }

    public int getMileage() {
        return mileage;
    }

    public abstract String getInfo();
}

• Abstract classes are indicated with slanted text in class diagrams

Adding an abstract method

public abstract class Vehicle {
    protected String makeModel;
    protected int mileage;

    public Vehicle(String s, int m) {
        this.makeModel = s;
        this.mileage = m;
    }

    public void setMakeModel(String s) {
        makeModel = s;
    }

    public void setMileage(int m) {
        mileage = m;
    }

    public String getMakeModel() {
        return makeModel;
    }

    public int getMileage() {
        return mileage;
    }

    public abstract String getInfo();
}
- Subclasses must implement `getInfo`

The revised car lot

```
public class Car extends Vehicle {
    public Car(String s, int m) {
        super(s, m);
    }
    public String getInfo() {
        final StringBuilder info = new StringBuilder();
        info.append("This Car is a ");
        info.append(makeModel);
        info.append(", with ");
        info.append(mileage);
        info.append(" miles.");
        return info.toString();
    }
}
```

```
public class Truck extends Vehicle {
    private int capacity;
    public Truck(String s, int m, int c) {
        super(s, m);
        capacity = c;
    }
    // ...
    public String getInfo() {
```
• Each child class extends the same abstract parent class
• Each child class provides its own implementation of the parent’s abstract methods

Another extension
Let’s extend Vehicle to keep track of the number of wheels

```java
public abstract class Vehicle {
    // All the old stuff
    public abstract String getInfo();
    public abstract int countWheels();
}
```

We could add a method to both Car and Truck

```java
public int countWheels() { return 4; }
```

But it’s never a good sign when we have to copy code

An abstract extension
Not every vehicle has four wheels, but there are (at least) two subclasses which do

• These two subclasses can share a parent:

```java
public abstract class FourWheeledVehicle extends Vehicle {
    public int countWheels() { return 4; }
}
```

  – Must still be abstract, since FourWheeledVehicle does not define a body for getInfo

• Truck And Car can extend the new parent instead of Vehicle

```java
public class Car extends FourWheeledVehicle {
    // Same as before
}
```

```java
public class Truck extends FourWheeledVehicle {
    // Same as before
}
```

  – They inherit an abstract getInfo as before, and each provides a body for it
  – They inherit a concrete countWheels, since FourWheeledVehicle gives it a body
Abstract classes in arrays

```java
public void fillLot() {
    final Vehicle[] vehicles = new Vehicle[10];
    for (int i=0; i<vehicles.length; ++i) {
        if (i % 2 == 0) {
            vehicles[i] = new Car("Honda Civic", 0);
        } else {
            vehicles[i] = new Truck("Ford F-150", 0, 10);
        }
    }
    displayLot(vehicles);
}

private void displayLot(Vehicle[] vehicles) {
    for (int i = 0; i < vehicles.length; ++i) {
        System.out.println(i + ": " + vehicles[i].getInfo());
    }
}
```

• The base type of the array is the abstract type
  – Stores references to objects that conform to `Vehicle`
• Array is filled with references to objects whose actual type is concrete
  – Of course: only concrete types can be instantiated!
• Calling `getInfo()` works for all objects
  – Based on the actual object type, Java dispatches the version appropriate for each

Software design patterns

So far, you’ve learned new language features to solve new problems
• But we’re mostly at the end of the features of languages like Java
• Important keys for harder problems include
  – More sophisticated algorithms
  – Describing ideas which encompass several language structures and how they interact

A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design


Today we’ll look at two examples of design patterns
• Template Method is a behavioral pattern
  – Making classes behave in a structured, predictable way

• Factory Method is a creational pattern
  – Creating classes in a structured, predictable way

**Template method**

*Template Method* is a structure for classes with related behavior

• Define an outline (skeleton) of an algorithm in a template class, and provide implementation details in subclasses

In Java, we apply *Template Method* by

1. Creating as abstract class to provide a sketch or outline of a class, and then

2. Filling in the details using subclasses

**Template Method example**

```java
public abstract class Game {
    public abstract void play() {
        initialize();
        while (!gameIsOver()) {
            takeATurn();
        }
    }
}

// The template methods
protected abstract void initialize();
protected abstract boolean gameIsOver();
protected abstract void takeATurn();
}
```

Particular game classes will provide the details for each step!

**Factory Method pattern**

*Factory Method* is a creational pattern

• Instead of using constructors directly within code, define an interface for object creation
  – Interface has *factory methods* which returns a new instance
  – Factory methods may take argument just as constructors may take arguments

```java
public interface VehicleFactory {
    public Vehicle build(int mileage);
}
```
public class VolvoFactory extends VehicleFactory {
    public Vehicle build(final int mileage) {
        return new Car("Volvo", mileage);
    }
}

public class FordTruckFactory extends VehicleFactory {
    public Vehicle build(final int mileage) {
        return new Truck("Ford F-650", mileage, 3);
    }
}

8 Recursion

Simple looping structures
To add repetition to code, we generally use basic loops

- Loops can run indefinitely or for fixed intervals
- All of the loop occurs within the context of a single method

```java
int sum = 0;
int count = 1;
while (count < 6) {
    sum = sum + count;
    System.out.println(count + ": " + sum);
    ++count;
}
```

Simple method calls
Methods can call one another (caller/callee relationship)

- Improves the logical/functional structure in code
- Each method call transfers control to a new method
- When a method returns, control is transferred back to the original method
// Simple printing method
public static void printGreater(int x, int y) {
    if (isGreaterThan(x, y)) {
        System.out.println(x + " is greater than "+ y);
    } else {
        System.out.println(y + " is greater than " + x);
    }
}

// Returns true if x is greater than y, and false otherwise
public static boolean isGreaterThan(int x, int y) {
    return (x > y);
}

**Executing the printGreater method**

// Simple printing method
public static void printGreater(int x, int y) {
    if (isGreaterThan(x, y)) {
        System.out.println(x + " is greater than " + y);
    } else {
        System.out.println(y + " is greater than " + x);
    }
}

// Returns true if x is greater than y, and false otherwise
public static boolean isGreaterThan(int x, int y) {
    return (x > y);
}

Four steps:

1. When called (e.g., from main), printGreater takes control of program execution
2. It calls isGreaterThan, which transfers control to that method
3. isGreaterThan returns the boolean it has computed
4. printGreater continues until done then control returns to its caller

**Keeping track of method calls**

Whenever a method is called, the computer needs to keep track of where it is in the calling method

The run-time stack, or call stack, is a linear data structure that maintains information about the active subroutines (methods) in a program

- An active subroutine is one that has been called but has not yet finished execution
- When a method is called, information about that method is pushed onto the call stack
- When a method finishes, it is popped from the stack
The call stack

// Simple printing method
public static void printGreater(int x, int y) {  
    if (isGreaterThan(x, y)) {  
        System.out.println(x + " is greater than " + y);  
    } else {  
        System.out.println(y + " is greater than " + x);  
    }  
}  
// Returns true if x is greater than y, and false otherwise
public static boolean isGreaterThan(int x, int y) {  
    return (x > y);  
}

1. Start in main
2. Call printGreater
3. Call isGreaterThan
4. Finish isGreaterThan
5. Finish printGreater

Another example
What happens when we call start()?

public static void start() {  
    int x = selfCall(5);  
    System.out.println("Fin: " + x);  
}  
public static int selfCall(int x) {  
    System.out.println("Pre: " + x);  
    return selfCall(x);  
}

- Like printGreater, selfCall calls a method inside itself
- Unlike printGreater, selfCall calls itself

Recursion is when a thing is defined in terms of itself
- In programming, it occurs when a method calls itself
The call stack
Let’s look at the call stack when `start()` is called

```java
public static void start() {
    int x = selfCall(5);
    System.out.println("Fin: "+x);
}
public static int selfCall(int x) {
    System.out.println("Pre: "+x);
    return selfCall(x);
}
```

- Start in `main`
- Call `start`
- Call `selfCall`
- Call `selfCall`
- Call `selfCall`
- ...

Components of recursion
For recursion to work properly, we need:

**Recurrence condition** Cases where the code will call itself to generate repetition

**Base case(s)** Some point that we are guaranteed to reach, where the recurrence will stop and the method will *not* call itself anymore

Just as for a loop!

- Loops need a continuation condition
- Which eventually becomes false, and makes us exit the loop
Fixing the recursive method

A corrected version of selfCall:

```java
public static void start() {
    int x = selfCall(5);
    System.out.println("Fin: " + x);
}

public static int selfCall(int x) {
    System.out.println("X: " + x);
    if (x <= 0) {
        System.out.println("Base case!");
        return 0;
    } else {
        int r = x + selfCall(x - 1);
        System.out.println("R: " + r);
        return r;
    }
}
```

Main

```
"Fin: 15"
```

Test yourself

Trace the following recursive method calls to determine output:

```java
public static void start() {
    System.out.println("A = " + mystery(2,2));
    System.out.println("B = " + mystery(3,4));
    System.out.println("C = " + mystery(10,7));
}

public static int mystery(int a, int b) {
    if (b == 0) {
        return 0;
    } else {
        return a + mystery(a - 1, b - 1);
    }
```

Main

```
"A = 6"
"B = 22"
"C = 1000000"
```

Test yourself

Trace the following recursive method calls to determine output:
return 0;         // A
} else if (b == 1) {
    return a;      // B
} else {
    return a + mystery(a, b-1); // C
}

For the call to \texttt{mystery(3, 4)}

1. When we call \texttt{mystery(3, 4)}
   - Reach line \texttt{C}
   - Calculate $3 + \text{mystery}(3, 3)$
   - Call \texttt{mystery}(3, 3)

2. When we call \texttt{mystery}(3, 3)
   - Reach line \texttt{C}
   - Calculate $3 + \text{mystery}(3, 2)$
   - Call \texttt{mystery}(3, 2)

3. When we call \texttt{mystery}(3, 2)
   - Reach line \texttt{C}
   - Calculate $3 + \text{mystery}(3, 1)$
   - Call \texttt{mystery}(3, 2)

4. When we call \texttt{mystery}(3, 1)
   - Reach line \texttt{B}
   - Return \texttt{a}, which is 3

5. Back to third stack frame
   - \texttt{mystery}(3, 1) returned 3
   - So $3 + \text{mystery}(3, 1)$ gives 6

6. Back to second stack frame
   - \texttt{mystery}(3, 1) returned 6
   - So $3 + \text{mystery}(3, 2)$ gives 9

7. Back to first stack frame
   - \texttt{mystery}(3, 1) returned 6
   - So $3 + \text{mystery}(3, 1)$ gives 12
How do we use recursion?

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— Niklaus Wirth, *Algorithms + Data Structures = Programs*

Many mathematical series are defined by recursive recurrence relations

- Factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdots \cdot 2 \cdot 1$
  - $f(0) = 1$
  - $f(n) = n \cdot f(n-1)$

- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$
  - $f(0) = 0$
  - $f(1) = 1$
  - $f(n) = f(n-1) + f(n-2)$

Test yourself

Write two methods to calculate $n!$ given a positive integer $n$, one using loops and one using recursion.

With a loop

```java
public static int factorialLooped(int n) {
    int product = 1;
    for (int i=1; i<=n; ++i) {
        product *= i;
    }
    return product;
}
```

With recursion

```java
public static int factorialRec(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorialRec(n-1);  
    }
}
```
Tracing the recursive factorial method

```java
public static void start() {
    int res = factorial(4);
    System.out.println("4! = " + res);
}

public static int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

```
main()
<table>
<thead>
<tr>
<th>start()</th>
</tr>
</thead>
</table>
| factorial(4) return 4*(6)
| factorial(3) return 3*(2)
| factorial(2) return 2*(1)
| factorial(1) return 1
```

Test yourself: Fibonacci numbers

Write a recursive method that calculates the nth Fibonacci number, defined by:

- \( f(0) = 0 \)
- \( f(1) = 1 \)
- \( f(n) = f(n-1) + f(n-2) \)

Then compute \( f(2) \) through \( f(5) \)

```java
public static int fibonacci(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Fibonacci numbers: comparing implementations

In the recursive implementation for computing the nth Fibonacci number with \( n = 4 \), how many recursive calls are made?

**Calls we’d make**

\( fib(5) \) calls first \( fib(3) \), then \( fib(4) \)
• fib(3) calls first fib(1), then fib(2)
  – fib(1) is a base case
  – fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case

• fib(4) calls first fib(2), then fib(3)
  – fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case
  – fib(3) calls first fib(1), then fib(2)
    * fib(1) is a base case
    * fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case

That’s 15 calls

• Lots of repeated work!
• At least in this form, Fibonacci is less suited to implementation by recursion

Fibonacci numbers: running that code
In the recursive implementation for computing the $n$th Fibonacci number with $n = 4$, how many recursive calls are made?

Calls we’d make
fib(5) calls first fib(3), then fib(4)

• fib(3) calls first fib(1), then fib(2)
  – fib(1) is a base case
  – fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case

• fib(4) calls first fib(2), then fib(3)
  – fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case
  – fib(3) calls first fib(1), then fib(2)
    * fib(1) is a base case
    * fib(2) calls first fib(0), then fib(1)
    * fib(0) is a base case
    * fib(1) is a base case
* fib(1) is a base case

That’s 15 calls

- Lots of repeated work!
- Is recursion a bad approach for Fibonacci?
  - Or is this way of structuring recursion bad?

**Fibonacci with a loop**

```java
public static long fib(final int num) {
    long fib=0;
    long next=1;
    for(int n=0; n<num; n++) {
        final long newNext=fib+next;
        fib = next;
        next = newNext;
    }
    return thisFib;
}
```

- The loop runs (about) num times, so we should be able to find a way that recurs num times
- Note how we use next and fib
  - Both calculated on each pass through the loop
  - Both preserved from one pass through the loop to the next
- How can we provide both from one recursive call to the next?
  - By passing both as parameters!

**Recursion with accumulating parameters**

- Instead of calculating the result in a method body after the return of a recursive call,
- Calculate the result in the arguments of the call

```java
static long fibHelper(int n, long fib, long next) {
    if (n<1) {
        return fib;
    } else {
        return fibHelper(n-1, next, fib+next);
    }
}
```
public static long fib(int n) {
    return fibHelper(n, 0, 1);
}

Test yourself: Mersenne numbers
Write and trace a recursive method to calculates a Mersenne number. Mersenne numbers are defined using the recurrence:

- \( f(1) = 1 \)
- \( f(n) = 2 \cdot f(n-1) + 1 \)

public static void start() {
    System.out.println("A = " + mersenne(2));
    System.out.println("B = " + mersenne(3));
    System.out.println("C = " + mersenne(4));
}

public static int mersenne(int n) {
    if (n == 1) {
        return 1;
    } else {
        return 2 * mersenne(n-1) + 1;
    }
}

A = 3
B = 7
C = 15

Greatest common divisor
The greatest common divisor (GCD) of two integers \( a \) and \( b \) is the largest positive integer that is a divisor of both \( a \) and \( b \) (i.e., it evenly divides them)

Several algorithms for computing the GCD:

- Brute force (starting at the smaller integer and counting down)
- Euclid’s algorithm
- Dijkstra’s GCD algorithm
Euclid’s algorithm
To find GCD(a,b) where a≥b:
1. If a=0, then GCD(a,b)=b
2. If b=0, then GCD(a,b)=a
3. Find q,r such that a=bq+r
4. Find GCD(b,r)
   • Since GCD(a,b)=GCD(b,r)
(Try 192, 78)

public static int gcd(final int a,
   final int b) {
if (a==0) return b;
if (b==0) return a;
final int q=a/b;
final int r=a%b;
return gcd(b,r);
}

Dijkstra’s algorithm
Motivation: quotient and remainder are comparatively slow operations
• So an algorithm based on addition/subtraction could be better
• If m%d and n%d are both zero
   – Then (m-n)%d is also zero
• So GCD(m,n) for m≥n is the same as GCD(m-n,n)
• To find GCD(a,b)
   1. If a=b, then GCD(a,b)=a
   2. If a>b, then GCD(a,b)=GCD(a-b,b)
   3. Else GCD(a,b)=GCD(a,b-a)
   (Try 192, 78 again)

Thinking recursively
Recursion is a very useful approach to use when the problem being solved contains similar subproblems that can be solved individually and then pieced back into a final solution.
The Sierpinski carpet fractal

1. Start with a square drawing region
2. Divide the square into 9 sub-squares
3. Color the center sub-square
4. Repeat the process for each of the remaining 8 sub-squares
Logical code structure for drawing the Sierpinski carpet fractal

Some GUI classes the department kicks around

1. Start with a square with side length \( n \)
2. Divide it into 9 squares with side length \( \frac{n}{3} \)
   - Must keep track of square size
3. Color in the center square
4. Repeat process for the other 8 squares
   - Must know the location of the square
5. Stop when the squares get small enough
   - Must know when to stop

Download from the course homepage

First a bit of overhead

**Window** A class wrapping up JFrame as a window
- add, remove for any JComponent

**Shape** An abstract class extending JComponent for shapes we might display in a Window

**Rectangle, Square, Oval** Concrete subclasses of Shape.
- Constructor parameters determine size and position
- Inherit JComponent methods like setBackground
Code for the Sierpinski carpet

private void drawFractal(int x, int y, int size, int depth) {
    if (depth <= 0) {
        return;
    }
    int subSize = size / 3;
    // Draw center square

    // Draw top row

    // Draw middle row

    // Draw bottom row
}

Code for the Sierpinski carpet

private void drawFractal(int x, int y, int size, int depth) {
    if (depth <= 0) {
        return;
    }
    int subSize = size / 3;
    // Draw center square
    Square s = new Square(x+subSize, y+subSize, subSize, subSize);
    s.setBackground(Color.YELLOW);
    window.add(s);
    // Draw top row

    // Draw middle row

    // Draw bottom row
}
Code for the Sierpinski carpet

```java
private void drawFractal(int x, int y, int size, int depth) {
    if (depth <= 0) {
        return;
    }
    int subSize = size / 3;
    // Draw center square
    Square s = new Square(x+subSize, y+subSize, subSize, subSize);
    s.setBackground(Color.YELLOW);
    window.add(s);
    // Draw top row
    drawFractal((x + 0*subSize) (y + 0*subSize), subSize, depth-1);
    drawFractal((x + 1*subsize), (y + 0*subSize), subSize, depth-1);
    drawFractal((x + 2*subSize), (y + 0*subSize), subSize, depth-1);
    // Draw middle row
    drawFractal((x + 0*subSize), (y + 1*subSize), subSize, depth-1);
    drawFractal((x + 2*subSize), (y + 1*subSize), subSize, depth-1);
    // Draw bottom row
    drawFractal((x + 0*subSize), (y + 2*subSize), subSize, depth-1);
    drawFractal((x + 1*subsize), (y + 2*subSize), subSize, depth-1);
    drawFractal((x + 2*subSize), (y + 2*subSize), subSize, depth-1);
}
```

Towers of Hanoi

Given a set of six disks of varying sizes, stacked on poles:

- Move the disks from the left pole to the right pole, while ensuring that only one disk is moved at a time and a large disk is never placed on top of a smaller one.

![Towers of Hanoi diagram]

Solving the Towers of Hanoi puzzle

The start of a strategy:

- Move the smallest five disks from the left pole to the center pole
- Move the largest disk from the left pole to the right pole
- Move the smallest five disks from the center pole to the right pole
Identifying subproblems

Until we move the smallest five disks to the center pole, we do not need to move the bottom one at all (nor can we).

- We can pretend the largest disk doesn’t exist, giving us a new smaller version of the same problem!
- Now we just have to figure out how to move the smallest five disks from the left pole to the center pole...

Solving the subproblem

To solve the subproblem, the same reasoning applies:

- Move the smallest four disks from the left pole to the right pole
- Move the second largest disk from the left pole to the center pole
- Move the smallest four disks from the right pole to the center pole

A simple recursive solution

```java
public static void moveDisks(final int numDisks,
                              final Pole source,
                              final Pole dest,
                              final Pole temp) {
```
if (numDisks > 1) {
    moveDisks(numDisks-1, source, temp, dest);
}

// Base case: move disk from source pole to dest pole
moveDisk(source, dest);

if (numDisks > 1) {
    moveDisks(numDisks-1, temp, dest, source);
}

Recursion and iteration
Every looping structure can be replaced with recursion and every recursive solution can be replaced with looping

• The basic while loop:

    while (condition) {
        loop body code;
        progress statement;
    }

• Recursive method:

    private void methodName() {
        if (condition) {
            loop body code;
            progress statement;
            methodName(); // recur (loop again)
        }
    }

Recursion or iteration?
Recursion and loops may be interchangeable, but sometimes a problem makes more sense with one or the other.

public static long
    factRec(final int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factRec(n-1);
    }
}

public static long
    factLoop(final int n) {
long product = 1;
for (int i=1; i<=n; ++i) {
    product *= i;
}
return product;
}

Recursive implementations can incur an overhead at runtime

- Space for keeping track of method calls and local variables
- Time for executing each method
- Many compilers can optimize tail calls — when the argument of the return statement is just the recursive call — to be just as efficient as loops

public static int
    fact(final int n) {
        return helper(n, 1);
    }
public static int
    helper(final int n,
        final long prod) {
        if (n<2) {
            return prod;
        } else {
            return helper(n-1, n*prod);
        }
    }

- For very small problems with an easy iterative solution, favor the loop

**Mutual recursion**

In some cases, recursion can be done indirectly and may not be immediately obvious.

- In mutual recursion, methods call each other

    public boolean isEven(int n) {
        if (n == 0) {
            return true;
        } else {
            return isOdd(n-1);
        }
    }

    public boolean isOdd(int n) {
        if (n == 0) {
            return false;
        } else {
            return isEven(n-1);
        }
    }
As usual, must make sure to move towards a base case!

Understanding mutual recursion can be more subtle

– But sometimes it’s the easiest solution

Another problem: Generating anagrams

Given a single word, generate all anagrams of the word

For example: "ants"

• ants
• atns
• nats
• ntas
• tans
• tsna
• tnas
• anst
• ntsa
• tasn
• tnsa
• asnt
• astn
• nsat
• nsta
• tsan
• atsn
• nast
• sant
• satn
• snat
• snta
• stan
• stna
An anagrams solution

// Method for others to call
public static void printAnagrams(String word) {
    helper("", word);
}

// Helper method for us to use which gets passed additional info
private static void helper(String prefix,
        String remChars) {
    if (remChars.length() <= 1) {
        System.out.println(prefix + remChars);
    } else {
        // Pick each letter from remChars as the "starting point"
        for (int i=0; i<remChars.length(); ++i) {
            final char c = remChars.charAt(i);
            final String remLeft = remChars.substring(0, i),
                remRight = remChars.substring(i + 1);
            helper(prefix+c, remLeft+remRight);
        }
    }
}

9 Exceptions

Errors in programming
    There are two basic forms of programming error:

Syntax errors
    Due to a violation of the syntax of the programming language
    - Incorrectly typed code, misspellings, wrong punctuation
    - Results in a compile-time error: code cannot be run

Logic errors
    Arise from syntactically correct code that can compile and run but does not work as expected
    - May result in a run-time error
    - JVM catches errors when the code executes
        - But not all errors will be caught!
Run-time errors in Java

When the JVM detects an error, it throws an exception

Example 1

String str = null;
str.toLowerCase();

Throws:

Exception in thread
"main" java.lang.
NullPointerException
   at Driver.main
      (Driver.java:5)

Example 2

int j = 0;
int k = 25/j;

Throws:

Exception in thread
"main" java.lang.
ArithmeticException
   at Driver.main
      (Driver.java:8)

Example 3

double[] arr = new double[3];
arr[3] = 29.4;

Throws:

Exception in thread
"main" java.lang.
ArrayIndexOutOfBoundsException
   at Driver.main
      (Driver.java:11)

Exception error messages

When the JVM detects an error, it throws an exception

• Running
public class Driver {
    public static void main(String[] args) {
        final String str = null;
        str.toLowerCase();
    }
}
gives:

Exception in thread "main" java.lang.NullPointerException
    at Driver.main(Driver.java:4)

- The exception is identified by type: NullPointerException
- The message has a stack trace of all active code (methods, classes, line numbers)
  * Sometimes called a traceback
  * Includes the name of the method executing when the exception occurs

A longer stack trace

Exception in thread "main" java.lang.NullPointerException
    at Thing.doSomething(Thing.java:9)
    at Thing.<init>(Thing.java:5)
    at Driver.main(Driver.java:3)

What does this stack trace tell us?

- Method doSomething() of class Thing in file Thing.java contains error at line 9
- doSomething() had been called by the Thing() constructor (<init>) at line 5 of file Thing.java
- And the Thing() constructor had been called from line 5 of the main() method of class Driver, file Driver.java

Throwing exceptions

- Exceptions can be generated using a throw statement
  
  throw EXPRESSION;

- Exceptions are just a particular kind of Java object
  
  - They all have superclass java.lang.Exception

- For example:

  public class SimpleFraction {
    private int numerator, denominator;

    public SimpleFraction(int n, int d) {
        if (d != 0) {

  81
What happens to the exceptions?

- When an exception is thrown, the current block of code terminates immediately
- If the current block does not have code to handle the exception, then the JVM moves to the enclosing block of code
  - And then the next one
  - And so on
  - When the top-level of a method terminates, we return to the calling point of that method
- Each enclosing block or method terminates, one at a time, until either
  1. The top level of the program is reached (usually the main method)
     - At which point the program terminates and displays a run-time error message to the user
  2. Or a block of code that can catch and handle the exception is reached

Handling exceptions
Exceptions are thrown up the call stack until they are caught with a try/catch/finally block
try {
    tryInstructionBody;
} catch (ExceptionClass1 parameterName) {
    exceptionHandlerBody1;
} catch (ExceptionClass2 parameterName) {
    exceptionHandlerBody2;
} finally {
    finallyBody;
}

- One or more catch blocks
- The finally block is optional

Semantics of the try statement

try {
    String str = null;
    System.out.print(str.trim());
} catch (ArithmeticException e) {
    System.out.println("Math Error");
} catch (NullPointerException e) {
    System.out.println("No String");
} finally {
    System.out.println("Finished");
}
System.out.println("More code here");

- Execution begins by running the body of try
- If an exception occurs, it is handled by the appropriate matching catch clause (if one exists)
- If a finally clause is included, then it will always execute after the try and any catch clauses
- str is declared but not initialized, so trim causes an exception
- Exception is checked, but not of type ArithmeticException, so this catch block is skipped
- Exception checked again; it is a NullPointerException, so this catch block executes
- Lastly, the finally block is run, printing out the message

Make sure the Scanner is closed

final Scanner scanner;
final int x;
try {
    scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    x = scanner.nextInt();
} catch (InputMismatchException e) {
    x = 0;
Exceptions and methods

The code that actually causes an exception does not always have to be directly inside a try block itself.

- Exception-causing code may be within another method that is called by the current one.

Types of exceptions

When deciding what a try/catch block can and cannot catch, the JVM checks whether the actual exception conforms to the catch block’s named type.

- There are many more kinds of exceptions than those shown below.

```
) finally {
    scanner.close(); // Close scanner regardless
}
```

```
Exceptions and methods

The code that actually causes an exception does not always have to be directly inside a try block itself.

- Exception-causing code may be within another method that is called by the current one.

Types of exceptions

When deciding what a try/catch block can and cannot catch, the JVM checks whether the actual exception conforms to the catch block’s named type.

- There are many more kinds of exceptions than those shown below.
```
Exceptions code trace

public void topTry() {
    try {
        System.out.println("try: Before methodA()");
        methodA();
        System.out.println("try: After methodA()");
    } catch (Exception e) {
        System.out.println("Handled by topTry!");
    }
    System.out.println("try: Finished");
}

public void methodA() {
    try {
        System.out.println("A: Before methodB()");
        methodB();
        System.out.println("A: After methodB()");
    } catch (NullPointerException e) {
        System.out.println("Handled by methodA!");
    } finally {
        System.out.println("A: Finally");
    }
    System.out.println("A: Finished");
}

public void methodB() {
    System.out.println("B: Throwing an exception");
    throw new ArithmeticException();
    System.out.println("B: Threw an exception");
}

Re-throwing from a catch block

catch blocks do not have to completely resolve an exception

• Caught exceptions can be re-thrown by handlers

• finally clause will still execute after re-throwing

try {
    String str = null;
    System.out.print(str.trim());
} catch (ArithmeticException e) {
    System.out.println("Math Error");
} catch (NullPointerException e) {
    System.out.println("No String");
    throw e;
} finally {
    System.out.println("Finished");
}
System.out.println("Never reached");
• `str` is declared but not initialized, so `trim` causes an exception

• Exception does not conform to `ArithmeticException`
  – So first `catch` block is skipped

• Exception does conform to `NullPointerException`
  – So message is printed
  – And then the exception is re-thrown

• Although we cancel execution of the handler(s) once the exception is re-thrown, still execute the `finally` block and print its message

• The last `println` never executes

**Type hierarchy of exceptions**

• Error
  – `OutOfMemoryError`
  – `StackOverflowError`

• Exception
  – `RuntimeException`
    * `ArithmeticException`
    * `ClassCastException`
    * `IllegalArgumentException`
    * `IndexOutOfBoundsException`
    * `NullPointerException`
    * `SecurityException`
  – `IOException`
  – ...

![Class hierarchy diagram]

Object

```
  Throwable
    Error
    Exception
```

Subclasses
**Handler order**

What gets printed by the following code?

```java
try {
    String str = null;
    String lower = str.toLowerCase();
} catch (Exception e) {
    System.out.println("Generic exception");
} catch (RuntimeException e) {
    System.out.println("Runtime exception");
} catch (NullPointerException e) {
    System.out.println("Null pointer");
}
```

The first handler matches:

**Generic exception**

- Always put the most specific exception types first!

**Checked and unchecked exceptions**

Some exceptions are **checked**, others are **unchecked**

- Checked exceptions must be explicitly addressed within the methods where they may be raised
  - Enforced by the compiler
  - What does it mean to *address* an exception? Is there some other way than catching it?

- Unchecked exceptions do not need to be explicitly handled in code
  - But will still cause runtime failure if they are not handled
Dealing with checked exceptions

Any code that might produce a checked exception must either:

• Catch it
  – Potential offending instructions placed inside a try block
  – Via a catch handler that matches the exception type

• Propagate it
  – Declare that this method can produce unhandled exceptions
  – Via a throws declaration

```java
public void writeToFile() throws IOException {
  // Code here that may produce an IOException
  // No try block is necessary
}
```

The throws declaration forces caller of the writeToFile to either catch the exception, or also propagate it

Catch or propagate

Catch

```java
public void methodA() {
  try {
    writeToFile();
  } catch (IOException e) {
    // Code to handle
    // the exception
  }
}
```

Propagate

```java
public void methodB() throws IOException {
  writeToFile();
}
```

Easy and wrong ways out

• A poor way to catch exceptions:

```java
public class MyClass {
  public static void main(String[] args) {
    try {
      // Exception-throwing code here
    } catch (Exception e) {
```
• A poor way to *propagate* exceptions:

```java
public class MyClass {
    public static void main(String[] args)
        throws Exception {
        // Exception-throwing code here
    }
}
```

**Printing the stack trace**
Sometimes we need to catch an exception but there is no graceful solution.

• *Print* the stack trace
• *Stop* the program

```java
try {
    ...
} catch (NullPointerException e) {
    System.out.println("Invoking null pointer handler...");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Invoking index handler...");
} catch (OutOfMemoryError e) {
    System.out.println("Invoking memory handler...");
} catch (Exception e) {
    // Unsure how to resolve...
    e.printStackTrace();
    System.exit(-1); // Stops the program
}
```

**Re-throwing from a catch block**
What gets printed by the following code?

```java
try {
    String str = null;
    System.out.print(str.trim());
} catch (NullPointerException e) {
    System.out.println("No String");
    throw e;
} catch (Exception e) {
    System.out.println("Generic exception");
} finally {
    System.out.println("Finished");
}
System.out.println("Reached?");
```
Re-thrown exceptions aren’t caught by the same try/catch block

No String
Finished

Nesting try/catch blocks

Code
What gets printed by the following code?

```java
public void test() {
    try {
        try {
            try {
                System.out.print("If at first you ");
                trying();
            } catch (ArithmeticException e) {
                System.out.print("do ");
            }
            finally {
                System.out.println("succeed");
            }
        } catch (NullPointerException e) {
            System.out.print("try ");
            throw e;
        }
        finally {
            System.out.print("...");
        }
    } catch (Exception e) {
        System.out.print("try ");
    }
    finally {
        System.out.print("again");
    }
    System.out.println("!");
}

public void trying() {
    System.out.print("don’t ");
    throw new NullPointerException();
}
```

Output
If at first you don’t succeed
try ... try again!

Creating custom exceptions
We can create our own exceptions in Java!

- Descendants of Exception are checked exceptions
• Descendants of RuntimeException are unchecked

```java
class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException(String msg) {
        super(msg);
    }
}
```

• Recommended practice:
  – Do not extend Error
  – Use existing exceptions where appropriate
  – Create checked exceptions in other circumstances

10 Generic methods

11 Search

**Searching for information**
Many applications involve finding pieces of information

• Finding a book in a library or name in an address book
• Finding movie show times & nearby locations
• Finding a path through a maze

• Finding the shortest drive from La Crosse to Las Vegas
• Finding a flight from La Crosse to London costing less than $1,200

**Simple searching**
Some of these types of searches are challenging, some are easier.

• Depends on the constraints of the search and the structure of the search space

An often critical factor in search is how our data is organized:
• Which *data structures* are we using?

• How can we access individual pieces of data?

A *data structure* is a particular way of organizing data in a computer (program) so that it can be used efficiently.

Example: How can be find a single piece of data in an array?

**Linear search**

Consider how to explain the search process step-by-step in English:

**Implementing linear search**

Take the English description and isolate the basic parts

1. Start at the beginning of the array

2. Check if the cell contains what you are looking for

3. If it does, then report success and stop

   • Otherwise, move on to the next cell and repeat, assuming you aren’t at the end of the array

4. If you get to the end of the array and haven’t found the item, report failure
Linear search in Java
After identifying the basic parts, translate into code:

```java
private static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; ++i) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Why return -1?
- If the search succeeds, the method returns the position of the target item within the array.
- If the search fails, the method returns a signal value of -1 to indicate to the caller that the target has not been found.
  - -1 is definitely not an index of the array!

Exploiting the structure of the data
Can we do better than linear search?
- If the input array is a random list of numbers, then probably not
  - The target number could be anywhere!
  - Example: Finding a particular word in a book
    It was a bright cold day in April, and the clocks were striking thirteen.
    Winston Smith, his chin nuzzled . . .
- If the input array is organized in some way, then maybe!
  - Example: Finding a particular word in a dictionary

```
dystopia: 1. an imaginary place where people lead dehumanized and often fearful lives
https://www.merriam-webster.com/dictionary/dystopia
```

Binary search

We can take advantage of sorted data to improve the search process.
Binary search in Java

How might we implement binary search in Java?

private static int binarySearch(int[] arr, int target) {
    int begin = 0;
    int end = arr.length - 1;

    while (begin <= end) {
        int mid = (begin + end) / 2; // Find the midpoint

        if (arr[mid] == target) { // Found it!
            return mid;
        } else if (arr[mid] < target) { // mid value too small
            begin = mid + 1;
        } else {/* arr[mid] > target */ // mid value too large
            end = mid - 1;
        }
    }
    return -1; // Failed search
}

Binary search: a recursive implementation

private static int binarySearch(int[] arr, int target) {
    return binSearchHelper(arr, target, 0, arr.length - 1);
}

private static int binSearchHelper(int[] arr, int target, int begin, int end) {
    if (begin > end) { // Base case #1
        return -1; // Failed search
    }
    int mid = (begin + end) / 2; // Find the midpoint

    if (arr[mid] == target) { // Found it! (base case #2)
        return mid;
    } else if (arr[mid] < target) { // mid value too small
        return binSearchHelper(arr, target, mid + 1, end);
    } else {/* arr[mid] > target */ // mid value too large
        return binSearchHelper(arr, target, begin, mid - 1);
    }
}