CS220 — Software Development II

J Maraist

November 26, 2017

Outline

Contents

1 Introduction 2

2 JUnit and test-driven development 4

3 Classes, objects, inheritance and notation 13

4 Arrays 25
   4.1 An ADT for resizable lists ................................................. 34

5 Interfaces 46

6 Basic generic types 46

7 Abstract classes 52

8 Recursion 60

9 Exceptions 80

10 Generic methods 92

11 Searching and sorting 92
   11.1 Linear search ...................................................................... 93
   11.2 Binary search ..................................................................... 94
   11.3 Comparing algorithms .......................................................... 96
   11.4 Bubble sort .......................................................................... 98
   11.5 Selection sort ...................................................................... 100
   11.6 Merge sort .......................................................................... 101
   11.7 Quicksort ............................................................................ 104

12 Linked lists 106
   12.1 Singly-linked lists ................................................................. 106
   12.2 Nested classes for hiding information .............................. 114
   12.3 Double-linked lists ............................................................... 116
   12.4 Equality on reference types ................................................... 121
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-jmarais/](https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmarais/)
  - 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
  - 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
  - Documentation at [junit.org/junit4/javadoc/latest/org/junit/Test.html](http://junit.org/junit4/javadoc/latest/org/junit/Test.html)

- 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
JUnit Assert methods


<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertEqual(String message, double expected, double actual)</code></td>
<td>Asserts that two doubles are equal within a positive delta.</td>
</tr>
<tr>
<td><code>assertEqual(String message, float expected, float actual)</code></td>
<td>Asserts that two floats are equal within a positive delta.</td>
</tr>
<tr>
<td><code>assertEqual(String message, long expected, long actual)</code></td>
<td>Asserts that two longs are equal.</td>
</tr>
<tr>
<td><code>assertEqual(String message, Object[] expecteds, Object[] actuals)</code></td>
<td>Deprecated, use <code>assertArrayEquals</code></td>
</tr>
<tr>
<td><code>assertEqual(String message, Object expected, Object actual)</code></td>
<td>Asserts that two objects are equal.</td>
</tr>
</tbody>
</table>

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
Test annotation arguments

From [junit.org/junit4/javadoc/latest/org/junit/Test.html](https://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`

```java
@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()
            .getBasis();

    System.out.println("Only one member of the basis");
    System.out.println(basis.length);

    System.out.println("Member of basis is \\
        + StringBuilderTests.SINGLE
        + \\
        ,
    System.out.println(basis[0]);
}

• 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; }
}
```

• 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;
```

• 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>
<tbody>
<tr>
<td>Car</td>
<td>- String makeModel - int mileage</td>
<td>«constructor» + Car (String, int)</td>
</tr>
</tbody>
</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?

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

What does the plus sign mean?

- 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:

  // 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)

```
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 `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

```
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
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
public class Vehicle {
    // data and methods
}
public class Car extends Vehicle {
    // more data and methods
}
public class Truck extends Vehicle {
    // more data and methods
}

Class diagrams for inheritance

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

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ constructor + Car(String, int)</td>
</tr>
</tbody>
</table>

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**
  
  ```java
  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

  ```java
  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.

```java
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
Truck has an extra instance variable

Constructor runs `super()` to set common variables, then stores capacity itself

Has some additional unique methods

**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

```
      Object
       ▲
     Vehicle
      ▲
     Car       Truck
```

**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

```
      Object
       ▲
     Vehicle
      ▲
     Car       Truck
```

**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();
}

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

```java
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);
```

```java
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
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

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

![Array Diagram]

**Basic syntax**

• Declare an array (does not allocate memory):

```java
final dataType[] arrayName;
```

  – Alternative syntax

```java
final dataType arrayName[];
```

• Allocate memory for a previously declared array:

```java
arrayName = new dataType[numberOfElements];
```

  Size cannot be negative
• Store and retrieve values in array:

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

• Access the length of an array:

```java
arrayName.length
```

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

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

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

- Declare an array (does not allocate memory):
  ```java
double[] numbers;  /* Alternate: */ double numbers[];
```

- Allocate memory for a previously declared array:
  ```java
numbers = new double[10];
```

- Store and retrieve values in array:
  ```java
numbers[3] = 7.5;
System.out.print(numbers[3]);
```

- Access the length of an array:
  ```java
numbers.length
```

- One-liners for declaration, allocation, and initialization:
  ```java
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:
  ```java
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
  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
class Main {
    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
class Main {
    public static void main(String[] args) {
        int[] myArr = {1, 2, 3};
        modifyArray(myArr);
        System.out.println(myArr[2]);
    }

    public static void modifyArray(int[] arr) {
        arr[0] = 7;
        arr = new int[3];
        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:
int[][] arr2d = new int[3][5];
arr2d[1][2] = 4;

• Accessing dimensions:

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

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

// ...

ratings[0][3] = 5;

Multi-dimensional arrays

... and on to higher dimensions

• A one-dimensional array
int[] arr1d = new int[6];
arr1d[3] = 7;

• A two-dimensional array

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

• A three-dimensional array

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

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:

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

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?

```java
// 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)

### 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.

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.

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:

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:

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

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

class ListPrinter {
    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 to 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

<table>
<thead>
<tr>
<th>MyList</th>
</tr>
</thead>
<tbody>
<tr>
<td>«constructor» +</td>
</tr>
<tr>
<td>MyList () «update»</td>
</tr>
<tr>
<td>+ void add(String) + void add(String, int) [0.5em]</td>
</tr>
<tr>
<td>«query» + String get(int)</td>
</tr>
<tr>
<td>+ int size()</td>
</tr>
</tbody>
</table>

Implementation

<table>
<thead>
<tr>
<th>MyList</th>
</tr>
</thead>
<tbody>
<tr>
<td>«constructor» +</td>
</tr>
<tr>
<td>MyList () «update»</td>
</tr>
<tr>
<td>+ void add(String) + void add(String, int) [0.5em]</td>
</tr>
<tr>
<td>«query» + String get(int)</td>
</tr>
<tr>
<td>+ int size()</td>
</tr>
</tbody>
</table>

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

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

Making it pass

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

custom 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();
    assertEquals(0, ml.size());
    ml.get(0);
}
```

Is there extra work for multiple appends?

The test

We can add to the appendingStuff test:

```
 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

```
@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);
    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));
}
```

```java
@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?

```java
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?

```java
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

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

public int size() {
    return used;
}

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

The add method

```java
public void add(final String item,
    final int index) {
    final String[] oldStrings = strings;
    final int oldSize = size();
    if (oldSize == capacity()) {
        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;
    used += 1;
}
```

Allocate more than one slot at a time

- How many slots should we allocate at a time?

```java
private static final int INCREMENT=10;
private String[] strings = new String[INCREMENT];
```
- Note no magic numbers!

- In add, change the reallocation line:

  ```java
  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));
}
```
Making the test pass

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

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:

  ```java
  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

```java
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:

```java
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
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”.)

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

More info at 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

```java
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

    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

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

The Iterator interface

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

Making MyList iterable
First, of course, a test

@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());
}

50
A class for the iterator

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

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:

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

Another look at Comparable

• Specify the type to which it is valid to compare
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

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:
Shape s = new Shape();

gives us:

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

public class Rectangle extends Shape {
  public Rectangle(int x, int y, int w, int h) {
    super();
    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
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;
    }
}
```
- Abstract classes are indicated with *slanted text* in class diagrams

**Adding an abstract method**

```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 String getInfo();
}
```
Subclasses must implement `getInfo`

The revised car lot

```java
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("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() {
        String capacityInfo = "Truck has a capacity of ", with "");
        return String.valueOf(capacity) + "");
    }
```

Extending abstract classes

```java
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("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() {
        String capacityInfo = "Truck has a capacity of ", with "");
        return String.valueOf(capacity) + "");
    }
```
• 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

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

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:

    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

    public class Car extend FourWheeledVehicle {
        // Same as before
    }

    public class Truck extend 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);
}

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 an 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 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;
    }
}
}
```

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td></td>
<td>&quot;Fin: 15&quot;</td>
</tr>
<tr>
<td>selfCall(5)</td>
<td></td>
<td>return 5+(10)</td>
</tr>
<tr>
<td>selfCall(4)</td>
<td></td>
<td>return 4+(6)</td>
</tr>
<tr>
<td>selfCall(3)</td>
<td></td>
<td>return 3+(3)</td>
</tr>
<tr>
<td>selfCall(2)</td>
<td></td>
<td>return 2+(1)</td>
</tr>
<tr>
<td>selfCall(1)</td>
<td></td>
<td>return 1+(0)</td>
</tr>
<tr>
<td>selfCall(0)</td>
<td></td>
<td>return 0</td>
</tr>
</tbody>
</table>
```

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 64;
    }
```
return 0; // A
} else if (b == 1) {
    return a; // B
} else {
    return a + mystery(a, b-1); // C
}

For the call to mystery(3, 4)

1. When we call mystery(3, 4)
   • Reach line C
   • Calculate 3+mystery(3,3)
   • Call mystery(3,3)

2. When we call mystery(3,3)
   • Reach line C
   • Calculate 3+mystery(3,2)
   • Call mystery(3,2)

3. When we call mystery(3,2)
   • Reach line C
   • Calculate 3+mystery(3,1)
   • Call mystery(3,2)

4. When we call mystery(3,1)
   • Reach line B
   • Return a, which is 3

5. Back to third stack frame
   • mystery(3,1) returned 3
   • So 3+mystery(3,1) gives 6

6. Back to second stack frame
   • mystery(3,1) returned 6
   • So 3+mystery(3,2) gives 9

7. Back to first stack frame
   • mystery(3,1) returned 6
   • So 3+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: \( n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 2 \cdot 1 \)
  - \( f(0) = 1 \)
  - \( f(n) = n \cdot f(n-1) \)

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

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

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);
    }
}

Test yourself: Fibonacci numbers

Write a recursive method that calculates the \(n\)th Fibonacci number, defined by:

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

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

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 \(n\)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

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 \times 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 $a \geq b$:

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)

```java
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 $m \geq n$ 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

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.setBackgroundColor(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.

```
  

```

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

```java
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

```java
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
• nats
• 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
Exception
    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) {

82
numerator = n;
denominator = d;
} else {
    final IllegalArgumentException
    error = new IllegalArgumentException
            ("Denominator is 0");
    throw error;
}

- IllegalArgumentException is also part of package java.lang

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;
```java
} 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

```plaintext
Exception
   
   RuntimeException

   Exception
   
   NullPointerException

   Exception
   
   IndexOutOfBoundsException

   ArrayIndexOutOfBoundsException
```

---

**Flowchart**

Exceptions code trace

```java
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`
  
  – ...
**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

![Diagram of Exception Hierarchy]

- **Object**
- **Throwable**
- **Error**
- **Exception**
- **Unchecked**
- **Checked**
- **RuntimeException**
- **Unchecked**
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
public 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 Searching and sorting

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?

11.1 Linear search

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
3a. If it does, then report success and stop
3b. 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

Output: need a return statement

Conditional: need if/else

Repetition: need a loop
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!

11.2 Binary search

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.

1. Start at the middle of the array
2. If the cell contains what you are looking for, report success and stop; otherwise either:
   3a. If cell value is too big, then look in the bottom half of the array
   3b. If cell value is too small, then look in the top half of the array
4. Repeat as needed
Binary search in Java

How might we implement binary search in Java?

```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

```java
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);
    }
}
```

Differences between linear and binary search

Binary search is more complicated than linear search — is this complexity worth it? How would we assess this?

Space  What are the memory requirements?
**Time** How long does it take to run?

The *worst-case complexity* of an algorithm is a measure of the amount of resources (time, space) the algorithm needs to deal with the worst possible input (the one that makes it do the most work).

For both search algorithms, the worst case we could have for running time is when the item being searched for is *not* in the array.

### 11.3 Comparing algorithms

**Measuring running time**

Having identified the worst case for these algorithms, how would we measure the running time they need? And how precise do we need to be?

- Wall clock time, CPU time
- Instruction count: Java instructions or machine-level instructions
- Number of basic units of work executed

Consider an algorithm \( A \) which operates on an input of size \( n \) (e.g., length of an array, number of lines in a file). Let \( f \) be the function that counts the number of basic units of work completed by the algorithm.

\[
f(n) = \begin{cases} 
2 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
3n - 1 & \text{otherwise}
\end{cases}
\]

**Big O notation**

Often we don’t need to worry about the minor details of an algorithm’s behavior — we just want the *big picture*.

*Big O notation* is a mathematical notation used to describe the limiting behavior of a function as its input tends towards infinity. For a function \( f(n) \), we say that \( f(n) \) is \( O(g(n)) \) for some function \( g \) if and only if there exists a positive constant \( M \) and value \( n_0 \) such that

\[
|f(n)| \leq M|g(n)| \text{ for all } n \geq n_0.
\]

This is written as \( f(n) = O(g(n)) \) or \( f(n) \in O(g(n)) \).

So for algorithm \( A \), we might have:

- \( f(n) = O(1) \) (constant)
- \( f(n) = O(n) \) (linear)
- \( f(n) = O(\log n) \) (logarithmic)
- \( f(n) = O(n^2) \) (quadratic)
**Running times for search algorithms**

The worst case for search is when the item being searched for is not in the array.

For *linear search*, the running time is $O(n)$.
- Looking at an element requires a constant amount of work
- Need to look at each element in the array

For *binary search*, the running time is $O(\log n)$.
- Finding the midpoint and inspecting the element requires a constant amount of work
- Need to halve the array about $\log_2 n$ times

**Comparing search algorithms**

Consider searching through an array of $2^{48}$ integers
- Each integer is 4 bytes, so total space is $2^{50}$ bytes, or one petabyte
- Around 2009, Google was processing 24 petabytes per day

Suppose it takes a nanosecond ($10^{-9}$ seconds) to process each entry
- *Linear search:*
  
  $2^{48}$ entries $\times \frac{10^{-9}}{\text{seconds}} \approx 2.81 \times 10^5$ seconds $\approx 3.25$ days

- *Binary search:*
  
  $\log_2(2^{48})$ entries $\times \frac{1}{\text{ns}} = 48$ ns $= 48$ billionths of a second

The difference between $O(n)$ and $O(\log n)$ can be dramatic!

**Sorting an array**

To use *binary search*, the array *must* to be sorted

There are *many* ways to do this.

But first, how can we tell if an array is sorted?
11.4 Bubble sort

Identifying an unsorted array

```java
public static boolean isSorted(int[] array) {
    // TODO: Implement this!
}
```

Bubble sort: overview

• Apply the scan-and-swap strategy to the array below:

```
2 1 3 6 0
```

• Compare the first pair of elements:

```
2 1 3 6 0
```

• Because the second element is smaller than the first, swap them.

```
1 2 3 6 0
```

Bubble sort: overview

• Repeat this process for subsequent pairs of elements:

```
1 2 3 6 0
```

```
1 2 3 6 0
```

```
1 2 3 0 6
```

• One scan through the array is not sufficient!

• We have moved the largest element to the rightmost slot
  – But after one pass, that’s all we can be sure of
Bubble sort: logical structure

1. Start at the beginning of the array
2. Check if first two elements are ordered correctly; if not, swap them
3. Repeat the process for subsequent pairs of elements
4. If no swaps were made, stop: the array is sorted
   - Otherwise, return to step 1 and repeat

Bubble sort: Java implementation

```java
public static void bubbleSort(int[] array) {
    boolean swapped;
    int numPasses = 0;
    do {
        swapped = false;
        for (int i=0; i<array.length-1-numPasses; ++i) {
            if (array[i] > array[i+1]) {
                int temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                swapped = true;
            }
        }
        ++numPasses;
    } while (swapped);
}
```

After each scan, the largest remaining element gets moved to the correct position, allowing us to stop the inner loop earlier each time

Bubble sort: complexity

The work done by `bubbleSort` is determined by how many times each loop executes

- Each pass through places the largest remaining item into its correct position, so at most \( n - 1 \) passes are required
- On the \( i^{th} \) pass, we have to look at \( n - i \) pairs of elements
- Looking at a pair of elements and swapping them if needed requires a constant amount of work

\[
\text{Total Work} = \sum_{i=1}^{n-1} (n - i) = (n - 1) + (n - 2) + \ldots + (n - (n - 1))
\]

\[
= \sum_{i=1}^{n-1} i = O(n^2)
\]
**Bubble sort: the best and worst cases**
In the *best case*, the array is already sorted

![Array in best case]

- `bubbleSort` requires one pass through the array to verify that no swaps are necessary: $O(n)$

In the *worst case*, the array is sorted in the reverse order

![Array in worst case]

- First pass moves the largest element to the end but leaves the remaining elements in the same relative ordering — $O(n^2)$ work in total

**11.5 Selection sort**

**Selection sort: overview**
Instead of scanning and swapping when we find an incorrect ordering, we could scan to find the smallest element, then move it to the beginning.

- Scan to find the smallest entry:

  ![Scan for smallest entry]

- Swap it into place:

  ![Swap into place]

- Repeat scan to find the next smallest entry and swap it into place:

  ![Repeat scan and swap]

**Selection sort: logical structure**

1. Start with the entire array marked "unsorted"
2. Scan through the unsorted portion to find the smallest element
3. Swap the smallest element with the element at the start of the unsorted portion; shrink the unsorted portion by one position
4. Repeat the process until there is no more unsorted portion
Selection sort: Java implementation

```java
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length-1; ++i) {
        int indexOfMin = i;
        for (int j = i + 1; j < array.length; ++j) {
            if (array[j] < array[indexOfMin]) {
                indexOfMin = j;
            }
        }
        if (i != indexOfMin) {
            int temp = array[i];
            array[i] = array[indexOfMin];
            array[indexOfMin] = temp;
        }
    }
}
```

Selection sort: complexity

The work done by `selectionSort` is determined by how many times each of the loops executes.

- We have $n$ iterations of the outer loop
- On the $i^{th}$ iteration, the inner loop executes $n-i$ times
- Comparing two elements requires a constant amount of work
- Swapping a pair of elements requires a constant amount of work

Worst-case running time is then $O(n^2)$

- Run time is the same regardless of whether or not the input is already sorted
- Same worst-case performance as Bubble Sort, but fewer swaps

Can we do better?

11.6 Merge sort

Merge sort: overview

Let’s try to apply the same idea we used for binary search to get better performance:

- A *divide and conquer* algorithm works by repeatedly breaking down a problem into smaller and smaller subproblems, until those subproblems become easy enough to be solved directly. The solutions to the subproblems then get pieced back together to provide a solution to the original problem.

- A sorting problem can be decomposed into smaller sorting problems
- Sorting a single element is an easy problem (base case)
- Subproblems can be recombined by merging their solutions together

Can we do better?
Merge sort: overview

Merge sort: merging two lists

- To merge two lists, start at the beginning of each one:

```
2 5 6 8
0 3 4 9
```

- Take the smaller element and place it in the new list:

```
2 5 6 8
0 3 4 9
```

Merge sort: merging two lists

- Advance the position counters:

```
2 5 6 8
3 4 9
```

- Take the smaller element and place it in the new list:

```
2 5 6 8
3 4 9
```

- Repeat this process:

```
5 8 6 2 9 4 3 0
```

- Until we get to the end:

```
0 2 3 4 5 6 8 9
```
Merge sort: Java implementation

The most intuitive way to implement merge sort is with recursion!

```java
public static int[] mergeSort(int[] array) {
    if (array.length > 1) { // Check stopping condition
        int mid = array.length / 2 - 1;
        // Split array contents into two smaller arrays
        int[] left = copyRange(array, 0, mid);
        int[] right = copyRange(array, mid+1, array.length-1);
        // Recursively sort the smaller arrays
        mergeSort(left);
        mergeSort(right);
        // Merge the sorted halves back together
        return merge(left, right);
    }
}
```

Merge sort: complexity

Given an array with \( n \) elements:

\[ O(n \log n) \text{ total work} \]

Merge sort: space complexity

Merge sort does \( O(n \log n) \) total work

- But it also allocates \( O(n \log n) \) total space
- The other algorithms sorted in place
• We can simplify pretty easily to $O(n)$ space — one spare buffer, and merge back-and-forth with the original space
• *But can we do better?*

### 11.7 Quicksort

**Quicksort**

• Also called partition-exchange sort
• Invented by Tony Hoare in 1959
• Refined over the years
• Quicksort is the default sorting algorithm in Java’s standard libraries
• But it was revised as recently as 2009 in Java 7

#### Quicksort: basic idea

1. Choose one element of the sequence, the *pivot*
2. Rearrange elements of the list so that:
   • Everything less than the pivot is to the left of the pivot
   • Everything greater than the pivot is to the right of the pivot
   • (Does not really matter what we do with equal values)
3. Recur on the values to the left and right of the pivot

• Sorts the array in place
• Choose?
  – The performance of quicksort depends crucially on the choice of the pivot
  – We’ll come back to this point later

#### Quicksort structure

```java
public static void quicksort(int[] array, int lo, int hi) {
    if (lo<hi) {
        final int p = partition(array,lo,hi);
        quicksort(array, lo, p-1);
        quicksort(array, p+1, hi);
    }
}
```

Delegate to *partition*

• Choosing the pivot
• Rearranging the array elements about the pivot
• Returning the index of the pivot
Quicksort partitioning

- For the pivot, choose element $hi$
- Loop maintains indices $i$ and $j$:
  - The pivot is bigger than entries from $lo$ to $i$ (inclusive)
    * Initially we have found no such entries
    * $i$ starts off as $lo-1$
  - Entries from $i+1$ to $j-1$ are bigger than the pivot
    * Initially we have found no such entries
    * $j$ starts off as $lo$
  - Entries from $j$ to $hi-1$ are to be arranged
    * The loop places entry $j$
- After the loop, we swap the pivot to between these regions
  - Check to see if needed

```java
public static int partition(int[] array, int lo, int hi) {
    final int pivot = array[hi];
    int i = lo-1;
    for(int j=lo; j<hi; j++) {
        if (array[j]<pivot) {
            i += 1;
            final int tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
        }
    }
    final int pivotPoint = i+1;
    if (pivot < array[pivotPoint]) {
        array[hi] = array[pivotPoint];
        array[pivotPoint] = pivot;
    }
    return pivotPoint;
}
```

How does Quicksort perform?
The for loop of partition visits every element of the (sub)list

- As with merge sort, the important question is how many times we do that

Some days, we are lucky
• If the pivot is near the middle of the range of values, we divide what we’re sorting about in half
• Then the analysis is as for merge sort: $O(n \log n)$

Some days, we are unlucky
• If the pivot is the highest or lowest value, we decrease the size of the unsorted area by one
• Then the analysis is as for selection sort: $O(n^2)$

Will we be lucky?

**Quicksort: the average case**
The worst case of QuickSort is that we are unlucky
• But in practice, this case is quite rare

QuickSort can be shown to have an *average* performance which really is $O(n \log n)$
• We can also push QuickSort towards $O(n \log n)$ performance by working harder on choosing the pivot
• Idea: take a larger constant amount of time to choose the pivot
• Or sometimes: take a non-constant time to choose the pivot for a greater average performance increase
• The current Java implementation
  – Uses an $O(n^2)$ for small arrays (below about 20)
  – Otherwise use a version of QuickSort with two pivots
  – Consistently runs faster in the average case than traditional QuickSorts — and Sun tested *heavily* before switching their implementation!

12 **Linked lists**

Linked lists

12.1 **Singly-linked lists**

**Self-referencing classes**
We have used to classes that are composed of primitives plus other, simpler types of data objects.

```java
public class Car {
    private int mileage;
    private String makeModel;
    // ...and so on...
}
```

What about classes that contain things of the *same* class?
public class NumberNode {
    private int num;
    private NumberNode next;
    // ...
}

Self-referencing classes
When we write a class, we are defining a sort of template

• Referencing other instances of the same class within the template is perfectly fine

• No object instances are created when the template is specified.

• The template specification simply declares variables that can be used to reference such objects — there is no immediate (and thus “infinite”) recursion

public class NumberNode {
    private int num;
    private NumberNode next;
    // ...
}

Linked data
Self-referencing classes allow us to link multiple instances of a class together

• These links can be structured to form a linked list of objects

A linked list is a linear collection of data elements, called nodes, which each store a data value and a pointer to the next node in the list.

Creating a list node: start with the data
Creating a list node: add link to next node

- Initially, the link is set to null (not pointing to anything)
- Methods allow us to manipulate both the data and the link after the node is created
Creating a simple, singly linked list

We can manually create a short linked list using these objects.

NumberNode nn1, nn2, nn3;
n1 = new NumberNode(1);
n2 = new NumberNode(2);
n3 = new NumberNode(3);
n1.setNext(nn2);
n2.setNext(nn3);

An alternate way to create the list

We do not need separate variables to reference each object

- Can use the getNext() method instead

NumberNode first;
first = new NumberNode(1);
first.setNext(new NumberNode(2));
first.getNext().setNext(new NumberNode(3));

A basic list structure

A basic list consists of nodes linked together
Creating lists of indefinite length

A linked list can be made arbitrarily long, simply by adding more objects onto the end of the list (bounded only by available memory)

- To create an empty list, allocate no nodes
- For the first element, we update the head pointer of the list record
- For subsequent elements, we keep track of the last node we created

public class NumberList {

//...
private void createList(int length) {
if (length < 1) {
    return;
}

head = new NumberNode(0);
NumberNode position = head;
for (int i=1; i<length; ++i) {
    NumberNode next = new NumberNode(0);
    position.setNext(next);
    position = next;
}
}
}

Traversing a linked list

A list can be traversed by starting at the head node and following the next links to the end, accessing data along the way.

- TODO: Implement the toString method

public String toString() {
    String listContents = "(";
    String sep = " ";

    NumberNode pos = head;
    while (pos != null) { // Loop until we reach the end
        listContents += sep + pos.getNum();
        sep = ", ";
        pos = pos.getNext(); // Advance to next node in list
    }

    return listContents + ")";
}
Determining the size of a list

A list can be traversed by starting at the head node and following the next links to the end, accessing data along the way.

- TODO: Implement the size method

```java
public int size() {
    int numNodes = 0;
    NumberNode pos = head;
    while (pos != null) {
        ++numNodes;
        pos = pos.getNext();
    }
    return numNodes;
}
```

This requires $O(n)$ time! Can we do better?

- All mutations to the chained nodes goes through NumberList
- So just add a size field to the NumberList class

The revised list class diagram

The list can keep track of its size internally.

```
NumberList
- int size - NumberNode
  head
  «constructor» +
  NumberList(int)[0.5em]
  «query» + int size() +
  int get(int) + String
toString()[0.5em]
  «update» + void set(int, int)

NumberNode
- int num - NumberNode
  next
  «constructor» +
  NumberNode(int)[0.5em]
  «query» + int
  getNum() + NumberNode
  getNext()[0.5em]
  «update» + void
  setNum(int) + void
  setNext(NumberNode)
```

Updating list data

A list can be traversed by starting at the head node and following the next links to the end, accessing data along the way.

- TODO: Implement the set method

```java
public void set(int index, int value) {
    // Check for invalid index first
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Invalid index");
    }
    NumberNode pos = head;
    for (int i = 0; i < index; ++i) {
```
pos = pos.getNext(); // Advance to correct position
pos.setNum(value); // Store the value at the position

This requires $O(n)$ time

- Not much we can do this time

Retrieving list data

A list can be traversed by starting at the head node and following the next links to the end, accessing data along the way.

- TODO: Implement the `get` method

```java
public int get(int index) {
    // Check for invalid index first
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Invalid index");
    }
    NumberNode pos = head;
    for (int i = 0; i < index; ++i) {
        pos = pos.getNext(); // Advance to correct position
    }
    return pos.getNum(); // Return the value at the position
}
```

Again we have $O(n)$ time

The node vs. the element

There is some duplicated code in `set` and `get`

- In both cases we need to access the $n^{th}$ node of the list representation
  - Although not necessarily the $n^{th}$ element of the list
- We can write a separate helper method with this reasoning

```java
private NumberNode getNode(int index) {
    // Check for invalid index first
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Invalid index");
    }
    NumberNode pos = head;
    for (int i = 0; i < index; ++i) {
        pos = pos.getNext(); // Advance to correct position
    }
    return pos;
}
```
– private because the NumberNode is an internal detail
– But still accessible from other methods

• Then get and set are much simpler

```java
public int get(int index) {
    final NumberNode pos = getNode(index);
    return pos.getNum(); // Return the value at the position
}
public void set(int index, int value) {
    final NumberNode pos = getNode(index);
    pos.setNum(value); // Store the value at the position
}
```

Adding an element to the list
In a linked list, we can insert a new element into the list simply by updating the next fields

```java
public void add(int index, int value) {
    if (index == 0) {
        head = new NumberNode(value, head);
    } else {
        final NumberNode pos = getNode(index-1);
        pos.setNext(new NumberNode(value, pos.getNext()));
    }
    size = size+1;
}
```

• So there’s an inconsistency here
  – We must treat an insertion to the front of the list differently

Sentinel nodes
We can avoid this inconsistency with an extra NumberNode at the beginning of the chain
• Called a sentinel node

• The sentinal holds no data, just points to the first actual node of the list

Setup with the sentinal
When setting up a list with a sentinal, we create the sentinal node along with the list wrapper

```java
public NumberList() {
    size=0;
    head = new NumberNode(-99, nil);
}
```
• –99 is not so much magical as it is irrelevant

**Fetching a node or the sentinel**
When we fetch a list node, we might also want the sentinel

```java
class NumberNode {
    private double value;
    public NumberNode(double value) {
        this.value = value;
    }
    public double getNum() {
        return value;
    }
    public void setNum(double value) {
        this.value = value;
    }
    public int getNext() {
        return 0;
    }
}
```

```java
private NumberNode getNodeOrSentinal(int index) {
    // Check for invalid index first
    if (index < -1 || index >= size) {
        throw new IndexOutOfBoundsException("Invalid index");
    }
    NumberNode pos = head;
    for (int i=0; i<=index; ++i) {
        pos = pos.getNext(); // Advance to correct position
    }
    return pos;
}
```

• –99 is not so much magical as it is irrelevant

**Adding an element with the sentinel**
In a linked list, we can insert a new element into the list simply by updating the next fields

```java
public void add(int index, int value) {
    final NumberNode prev = getNodeOrSentinal(index-1);
    prev.setNext(new NumberNode(value, prev.getNext()));
    size = size+1;
}
```

**Removing an element with the sentinel**
As with insertions, deletions just require updates to the next field

```java
public int remove(int index) {
    final NumberNode prev = getNodeOrSentinal(index-1),
    removed = prev.getNext();
    prev.setNext(removed.getNext()));
    return removed.getNum();
}
```

12.2 Nested classes for hiding information

**Hiding implementation details**
The user of a linked list should not have to worry about *how* the list is implemented

• They should only need to deal with the actual data in the list
• Can we hide the NumberNode class entirely?

```
```
Nested classes

A nested class is a class defined within another class

- There are two kinds: static nested classes and inner classes

```java
class OuterClass {
    //...
    private static class StaticNestedClass {
        //...
    }
    private class InnerClass {
        //...
    }
}
```

- Useful for organizing code and keeping related classes together
- Hides unnecessary details from other classes
- Makes testing harder
  - Can no longer create `NumberNode` objects by themselves for test purposes

Inner classes

- Associated with one instance of the outer class
- Can access all methods and data of its instance the outer class
  - Regardless of their accessibility (public or private)
- Can be marked private and completely hidden from other classes

```java
class NumberList {
    private NumberNode head, tail;
    private int size;
    //...
    private class NumberNode {
        private int num;
    }
}
```
private NumberNode next, prev;

//...
}
}

12.3 Double-linked lists

Types of linked list

We have been discussing singly linked lists

- Linked lists where each node contains a single link to another node in the list

There are also doubly linked lists

- Linked lists where each node contains two links, one to the subsequent node and one to the preceding node

Sentinel nodes in doubly linked lists

Many implementations of doubly linked lists will use two sentinel nodes: one at the head, and one at the tail

Creating a doubly linked list
Initializing a doubly linked list

```java
class NumberList {  
    private int size;  
    private NumberNode head;  
    private NumberNode tail;  

    public NumberList() {  
        clear();  
    }  

    public void clear() {  
        // TODO: Write me  
    }  

    //...
}
```

### NumberNode

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>int</td>
<td>Integer value</td>
</tr>
<tr>
<td>next</td>
<td>NumberNode</td>
<td>Next node in the list</td>
</tr>
<tr>
<td>prev</td>
<td>NumberNode</td>
<td>Previous node in the list</td>
</tr>
</tbody>
</table>

```java
+ NumberNode(int, NumberNode, NumberNode)
```

Initializing a doubly linked list

```java
class NumberList {  
    private int size;  
    private NumberNode head;  
    private NumberNode tail;  

    public NumberList() {  
        clear();  
    }  

    public void clear() {  
        head = new NumberNode(0, null, null);  
        tail = new NumberNode(0, head, null);  
        head.next = tail;  
    }
```
size = 0;
}

// ...
}

```
NumberNode
- int num - NumberNode next -
  NumberNode prev
+ NumberNode(int, NumberNode, NumberNode)
```

### Accessing linked list elements

How would we implement the `get` method?

```java
public int get(int idx) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    return getNode(idx).data;
}
```

- User-accessible method that allows access to data stored in the list
- Internally, it finds the node at the specified position via the `getNode` helper method and then returns its data
- The `getNode` method is completely hidden from the user

### Accessing linked list nodes

What’s the most efficient way to implement `getNode`?

```java
private NumberNode getNode(int idx) {
    // Allow retrieving the tail sentinel
    if (idx < 0 || idx > size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    NumberNode node;
    if (idx < size / 2) { // Scan from front
        node = head.next;
        for(int i=0; i<idx; ++i) {
            node = node.next;
        }
    } else { // Scan from back
```
node = tail;
for(int i=size; i>idx; --i) {
    node = node.prev;
}
return node;

Adding elements to a linked list

The user has two methods for adding elements:

- **add(int)** takes a value to be added at the end of the list

```java
NumberList list = new NumberList(); // ()
list.add(10); // (10)
list.add(20); // (10, 20)
list.add(30); // (10, 20, 30)
```

- **add(int, int)** takes an index and a value, and should insert the value into the list at the given index

```java
NumberList list = new NumberList(); // ()
list.add(0, 10); // (10)
list.add(0, 20); // (20, 10)
list.add(1, 30); // (20, 30, 10)
list.add(3, 40); // (20, 30, 10, 40)
list.add(3, 50); // (20, 30, 10, 50, 40)
list.add(1, 60); // (20, 60, 30, 10, 50, 40)
```

Implementing element addition

Next: the two `add` methods

```java
public void add(int value) {
    add(size, value); // Insert before node at index ‘size’
}

public void add(int idx, int value) {
    // Find the node itself, and insert a new node before it!
    addBefore(getNode(idx), value);
}
```

- We need to create a new `NumberNode` which contains the given value, and then position it correctly within the list by updating links.

Implementing node addition

How do we implement the `addBefore` method?
private void addBefore(NumberNode pos, int value) {
    NumberNode newNode = new NumberNode(value, pos.prev, pos);
    pos.prev = newNode;
    newNode.prev.next = newNode;
    ++size;
}

Implementing node removal

How do we implement the remove method for NumberNodes?

private int remove(NumberNode pos) {
    pos.next.prev = pos.prev;
    pos.prev.next = pos.next;
    pos.next = null;
    pos.prev = null;
    --size;
    return pos.data;
}

Implementing element removal

With the previous methods as helpers, the remove method for elements is straightforward

public int remove(int idx) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    return remove(getNode(idx));
}

• These methods are an exception to the rule that we should separate query methods and mutator methods
  – Returning what we remove is a common pattern for deletions from data structures
  – But the exceptions prove the rule!

Implementing update methods

We can again use getNode for the set method

public int set(final int idx, final int value) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    NumberNode pos = getNode(idx);
    int oldVal = pos.data;
    pos.data = value;
    return oldVal;
}
Implementing the query methods

How do we implement the `indexOf` method?

```java
public int indexOf(final int value) {
    int idx = 0;
    NumberNode pos = head.next;
    while (pos != tail) {
        if (pos.data == value) {
            return idx;
        }
        ++idx;
        pos = pos.next;
    }
    return -1;
}
```

A `for` loop for list nodes

We can also use a `for` loop to iterate through a doubly-linked list

- Start at one sentinel node and proceed until we find the other

```java
public int indexOfAlt(int value) {
    int idx = 0;
    for(NumberNode pos=head.next; pos!=tail; pos=pos.next) {
        if (pos.data == value) {
            return idx;
        }
        ++idx;
    }
    return -1;
}
```

Can we do better?

- In terms of \(O(n)\), no
- But we can reduce the number of actions for each node

### 12.4 Equality on reference types

**Identity equality**

In Java, we have two forms of equality for reference types: identity equality and content equality. Two objects have the same identity if and only if they are both the same object, and the same address in memory.

```java
Object obj1 = new Object();
Object obj2 = obj1;
if (obj1 == obj2) {
    System.out.println
```
if (obj1.equals(obj2)) {
    System.out.println("Same content");
}

Output:
Same object
Same content

Content equality
Two objects have the same content if and only if they have the same state (as defined by the programmer)

Person p1 = new Person("Joe Smith");
Person p2 = new Person("Joe Smith");
if (p1 == p2) {
    System.out.println("Same object");
}
if (p1.equals(p2)) {
    System.out.println("Same content");
}

Program output:
Same content

A while loop for list nodes
Let’s go back to indexOf

• Could you think of a way to make the loop quicker?
• Hint: how can we avoid testing for the tail sentinel at every node?
• Remember that the sentinels are regular NumberNode instances
  – They have a data field
– We just choose to disregard it

• So we can put whatever we want in the sentinel’s data field

– For example, we could put value there

– Now we know that we will find value in the NumberNode chain

– We need to check just once that we are not at the tail sentinel

```java
public int indexOfAlt(int value) {
    int idx = 0;
    tail.data = value;
    NumberNode pos = head.next;
    while (pos.data != value) {
        pos = pos.next;
        ++idx;
    }
    if (pos == tail) {
        return -1;
    } else {
        return idx;
    }
}
```

Creating a list of vehicles

Let’s modify the NumberList so that it can store Vehicle objects instead.

• NumberNode becomes VNode

• Type for data: int becomes Vehicle

• == becomes .equals

All classes extend Object, and so they inherit an equals method:

```java
object1.equals(object2)
```

• By default, equals is defined as ==, which returns true if and only if the two objects occupy the same location in memory

Defining content equality

By default content equality is defined as identity equality:

```java
public class Object {
    //...
    public boolean equals(Object other) {
        return (this == other);
    }
}
```

To modify this behavior, we need to override the equals method
But remember, `equals` is defined to be

```java
public boolean equals(Object other)
```

- We can compare any two objects for equality

• Usually, we first check the type of the other object
  
  - Usually, if it differs we can just return `false` right away
    ```java
    if (!(other instanceof Vehicle)) {
      return false;
    }
    ```
  
  • If the types do match, we **cast** the argument

    ```java
    final Vehicle v = (Vehicle)other;
    ```

    - Tell Java to treat `other` as a `Vehicle`
    - Java will check that the actual runtime type matches this assertion
    - But check our program has already checked, the cast is safe
    - Before Java generics, programs had many more casts

**Defining content equality — putting it all together**

Once we cast the argument, we can access its fields, and reason about the objects’ equality

```java
public abstract class Vehicle {
  public boolean equals(final Object other) {
    if (!(other instanceof Vehicle)) {  // should always print true
      return false;
    }
    final Vehicle v = (Vehicle)other;
    return getMakeModel().equals(v.getMakeModel()) && (getMileage() == v.getMileage());
  }
}
```

**Expectations for an `equals` method**

The `equals` method should implement an equivalence relation on non-null object references:

• It should be reflexive:

  ```java
  // Should always print true
  System.out.println(obj.equals(obj));
  ```

• It should be symmetric:
// Should always print true
if (objA.equals(objB)) {
    System.out.println(objB.equals(objA));
} else {
    System.out.println(!objB.equals(objA));
}

• It should be transitive

// Should always print true, or nothing
if (objA.equals(objB) && objB.equals(objC)) {
    System.out.println(objA.equals(objC));
}

• There is no way for the compiler to enforce these expectations!
  – But if we fail to meet these conditions, our programs can misbehave in mysterious (and very hard to debug!) ways
  – More details on the Javadoc page for Object

Implementing the equals method
The implementation of equals provided in Vehicle compares Vehicle instances by make/model and mileage, and is an equivalence relation

• All subclasses (Car, Truck, Van) inherit this implementation.

• Given that, what is the output of the code below?

final Truck
    t1 = new Truck("Ford F-150", 0, 5),
    t2 = new Truck("Ford F-150", 0, 5);
System.out.println(t1.equals(t2));
t2.setCapacity(7);
System.out.println(t1.equals(t2));

  – Both statements print true!

• Capacity of Truck instances is ignored
  – We never told Java that it should be considered
  – But easy to fix

The equals method for Truck instances
Truck instances should be compared by make/model, mileage and capacity

public class Truck extends Vehicle {
    //...

    // Overrides equals method from Vehicle
public boolean equals(Object other) {
    if (!(other instanceof Truck)) {
        return false;
    }
    Truck t = (Truck) other;
    return (getMakeModel().equals(t.getMakeModel()) &&
            getCapacity() == t.getCapacity() &&
            getMileage() == t.getMileage());
}

Avoiding duplication

Some of the comparisons are already encoded in the Vehicle.equals method

- We can avoid repeating those comparisons using super

public class Truck extends Vehicle {
    //...

    public boolean equals(Object other) {
        if (!(other instanceof Truck)) {
            return false;
        }
        final Truck t = (Truck) other;
        return (this.capacity == t.capacity)
                && super.equals(other);
    }
}

Checking the implementation

With the equals method for Truck instances, this code behaves as expected:

final Truck
    t1 = new Truck("Ford F-150", 0, 5),
    t2 = new Truck("Ford F-150", 0, 5);
System.out.println(t1.equals(t2)); // true
System.out.println(t2.setCapacity(7));
System.out.println(t1.equals(t2)); // false

But what about this case?

final Truck t = new Truck("Ford F-150", 0, 5);
final Car c = new Car("Ford F-150", 0);
System.out.println(t.equals(c)); // [A]
System.out.println(c.equals(t)); // [B]

- Line [A] prints true, as we would hope
- But line [B] prints false!
- We have broken the symmetric property of equals
  - (Which means transitivity is also broken)
Implementing equals across subclasses
In general, it is very difficult to retain the symmetric and transitive properties for equals across subclasses.

• A solution: Have each subclass provide its own equals method

```java
public class Car extends Vehicle {
    //...

    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        Car c = (Car) other; // Cast Obj. ref to Car ref
        return (getMakeModel().equals(c.getMakeModel()) &&
                getMileage() == c.getMileage());
    }
}
```

• We run into the same issues if we have subclasses of Car, Truck, or Van.

Checking the actual class
The class of an object is available at runtime

• Sometimes, this can help us avoid problems when comparing across subclass

• Object also defines the method getClass

  – Returns the actual, runtime type of the object
  – Expressed as an object of type java.lang.Class

• So in Vehicle we can check the actual classes of this and other:

```java
public abstract class Vehicle {
    //...

    public boolean equals(Object other) {
        if (getClass() != other.getClass()) {
            return false;
        }
        final Vehicle v = (Vehicle)other;
        return getMakeModel().equals(v.getMakeModel()) && (getMileage() == v.getMileage());
    }
}
```
12.5 Comparables and sorted lists

Sorted lists
Suppose we want to keep our list of Vehicle instances sorted in some way.

- Need a way to compare two Vehicle instances and identify which comes first

We can implement the Comparable interface

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

The compareTo() method should:

- Return 0 if the invoking object and obj are "equal"
- Return a negative number if invoking object is "less than" obj
- Return a positive number if invoking object is "greater than" obj

Implementing the Comparable interface
As with equals, the programmer defines what equal-to, greater-than, and less-than mean

For example, we will:

- Sort Car instances first by make/model, then by mileage
- Sort Truck instances by make and model, then capacity, then mileage
- Sort Van instances by make and model, then number of passengers, then mileage

We can ensure that all subclasses provide an implementation of compareTo by having the Vehicle class implement the interface

```java
public abstract class Vehicle implements Comparable<Vehicle> {
    
    So Vehicle agrees to provide a method int compareTo(Vehicle v)
    
    But we do not define any such method in Vehicle!
    
    It’s OK because Vehicle is abstract
    
    – We are placing an obligation on the concrete children of Vehicle
```
A first try at a `compareTo` method for `Car`

```java
public class Car extends Vehicle {
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            // Not a car, so unclear how to order
            return 0;
        }

        final Car c = (Car)other;

        // Compare makeModel first
        if (!makeModel.equals(c.makeModel)) {
            return makeModel.compareTo(c.makeModel);
        }

        // If same makeModel, compare by mileage
        return getMileage() - c.getMileage();
    }
}
```

**Requirements for the `compareTo` method**

The `compareTo` method should provide a total ordering on the objects of each class that implements it

- Sign should flip when reversing caller and argument
  - If `(x.compareTo(y) < 0)` then `(y.compareTo(x) > 0)`
  - If `(x.compareTo(y) > 0)` then `(y.compareTo(x) < 0)`

- It should be transitive
  - If `(x.compareTo(y) < 0 && y.compareTo(z) < 0)` then `(x.compareTo(z) < 0)`

- Sign should be consistent for equal objects

  If `x.compareTo(y)` returns 0 then either:
  - `(x.compareTo(z) > 0 && y.compareTo(z) > 0), or`
  - `(x.compareTo(z) < 0 && y.compareTo(z) < 0)`

See also the [Comparator Javadoc page](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)

**Consistency between `compareTo` and `equals`**

It is also strongly recommended that that `compareTo` and `equals` be consistent

```
(x.compareTo(y) == 0) == (x.equals(y))
```

That is: `compareTo` should say two objects are equal if and only if `equals` says that they are equal

So we should always have one of the scenarios below
• But it is not enforced by the compiler
• Up to the programmer to stay consistent

Both equal

obj1.equals(obj2) == true
obj1.compareTo(obj2) == 0

Both not equal

obj1.equals(obj2) == false
obj1.compareTo(obj2) != 0

An inconsistent compareTo and equals

public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        Car c = (Car) other;
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            return 0;
        }
        Car c = (Car) other;
        if (!this.makeModel.equals(c.makeModel)) {
            return this.makeModel.compareTo(c.makeModel);
        }
        return getMileage() - c.getMileage();
    }
}

• So is a Car instance equal to instances of other classes, or not?

Are these compareTo and equals consistent?

public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        final Car c = (Car) other;
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }
}
public int compareTo(Vehicle other) {
    if (!(other instanceof Car)) {
        return -1;
    }
    final Car c = (Car)other;
    if (!this.makeModel.equals(c.makeModel)) {
        return this.makeModel.compareTo(c.makeModel);
    }
    return getMileage() - c.getMileage();
}

• We have ordered Car instances before non-cars
• Is this viable?

More trouble with related classes

The fix on the previous slide

if (!(other instanceof Car)) {
    return -1;
}

works to ensure consistency when comparing Car instances, but also introduces some odd behavior if adopted as is for Truck

public class Truck extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            return -1;
        }
        // ...
    }
}

// And executing statements
Car c = new Car("Honda Civic", 214118);
Truck t = new Truck("Ford F-150", 0, 5);
if (c.compareTo(t) < 0) { System.out.println("c less than t"); }
if (t.compareTo(c) < 0) { System.out.println("t less than c"); }

This will print:
c less than t
t less than c
Comparison across related classes
We need to also enforce an ordering across classes

```java
public class Car extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Car)) {
            return -1; // Cars always come first
        }
        // ...
    }
}
```

```java
public class Van extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Van)) {
            return 1; // Vans always come last
        }
        // ...
    }
}
```

Comparison across related classes
We need to also enforce an ordering across classes

```java
public class Truck extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            if (other instanceof Car) {
                return 1; // Trucks come after Cars
            } else if (other instanceof Van) {
                return -1; // Trucks come before Vans
            } else {
                return -1; // Trucks before anything else
            }
        }
        // ...
    }
}
```

- Requires subclasses to know about each other
- Not particularly sustainable for a larger number of classes
- Burdensome when adding new subclasses of Vehicle

Sorted lists
Once we can compare our objects, we can ensure that our own lists are *sorted* by changing how objects get added.

**Implementation of sorted insert**

Sorted insertion is relatively simple, as long as the list is kept sorted

```java
public void add(final Vehicle v) {
    VNode pos = head.next;
    while (pos != tail && v.compareTo(pos.data) >= 0) {
        pos = pos.next;
    }
    addBefore(pos, v);
}
```

Prevent users from messing up the ordering

```java
public void add(final int idx, final Vehicle v) {
    // Ignore index specified by user
    add(v);
}
```

**Maintaining unique items**

Sorted lists can be used to maintain a unique set of items

- The list will hold no duplicates

```java
public void add(Vehicle v) {
    VNode pos = head.next;
    while (pos != tail && v.compareTo(pos.data) > 0) {
        pos = pos.next;
    }
    // v.compareTo(pos.data) <= 0, so check equality
    if (pos != tail && v.equals(pos.data)) {
        return false; // Don’t add if already stored
    }
    addBefore(pos, v);
}
```
12.6 List iterators

**The VehicleList class**

Let’s turn back to the unsorted list

We can:

- Create a list
- Add to list
- Remove from list
- Modify an element
- Find an element
- Get length

What else might we want to do?

- **Loop** over it!

```
VehicleList

«constructor» + VehicleList()
«query» + int size() + boolean isEmpty() + Vehicle get(int) + int indexOf(int) + boolean contains(Vehicle) + String toString()
«update» + void clear() + void add(Vehicle) + void add(int, Vehicle) + Vehicle set(int, Vehicle) + Vehicle remove(int)
```

**Looping through a list**

If we use the same loop as for array-based lists, we lose

```
for (int i=0; i<list.size(); ++i) {
    System.out.println(list.get(i));
}
```

Each call to `get` is $O(n)$, so the whole traversal would be $O(n^2)$ — not acceptable

Internally, a list can loop through itself because it has **direct access** to `head`, `tail`, and the data nodes

- These details are **private**, so other classes cannot access them

**Solution:** The list can provide other classes with an **iterator**

- An iterator is an object that enables a programmer to traverse a container, without needing to know the underlying structure of the container itself
public interface Iterable<T> {
    public Iterator<T> iterator();
}

public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}

Using a list iterator
How does list iteration work?

• Internal iteration in the VehicleList class

    public void display() {
        VNode pos = head.next;
        while (pos != tail) {
            System.out.println(pos.data);
            pos = pos.next;
        }
    }

• External iteration by an outside class

    VehicleList list = new VehicleList();
    // ... add items to list ...

    final Iterator<Vehicle> iter = list.iterator();
    // Just like Scanner: get next and advance
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }

Step 1: Make the list Iterable
A container must indicate that it can be iterated over by implementing the Iterable interface

public class VehicleList implements Iterable<Vehicle> {
    // ...
    public Iterator<Vehicle> iterator() {
        // TODO: We’ll fill this in later
        throw new UnsupportedOperationException();
    }
}

• The iterator method must return an object that can do the things an iterator should be able to do
  – Namely, an object that implements the Iterator interface
• We can specify a template for such an object as an inner class
Step 2: Make an Iterator

The class of iterator objects is often defined as a private inner class of the container itself.

```java
public class VehicleList implements Iterable<Vehicle> {
    // ...
    public Iterator<Vehicle> iterator() {
        return new VIterator();
    }

    private class VIterator implements Iterator<Vehicle> {
        // The VIterator class defines a template for iterators over a VehicleList
        // The Iterable container will provide an Iterator object when asked for one

        private VNode pos;
        public VIterator() {
            pos = head;
        }

        public boolean hasNext() {
            return (pos.next != tail);
        }

        public Vehicle next() {
            if (!hasNext()) {
                throw new NoSuchElementException("...");
            }
            return pos.next;
        }

        public void remove() {
            if (!hasNext()) {
                throw new NoSuchElementException("...");
            }
            pos = pos.next;
        }
    }
}
```

Building the VIterator class

In order to iterate through a VehicleList, a VIterator object needs to know where it is in the list at any point in time.

```java
public class VehicleList implements Iterable<Vehicle> {
    // ...
    private class VIterator implements Iterator<Vehicle> {
        private VNode pos;
        public VIterator() {
            pos = head;
        }

        public boolean hasNext() {
            return (pos.next != tail);
        }

        public Vehicle next() {
            if (!hasNext()) {
                throw new NoSuchElementException("...");
            }
            return pos.next;
        }

        public void remove() {
            if (!hasNext()) {
                throw new NoSuchElementException("...");
            }
            pos = pos.next;
        }
    }
}
```
pos = pos.next;
    return pos.data;
}
public void remove() {
    throw new UnsupportedOperationException();
}
}

- Track the current position as pos
- Starts at head sentinel
  - Keep pos before the node whose content would be returned for a call to next()
- We can tell if there is another element by whether pos.next is the tail sentinel
- At each call to next(), advance pos and return data
- We'll opt not to support remove

Using the iterator
The iterator allows external classes to loop over a list’s contents

```
final VehicleList list = new VehicleList();
/* add contents to list here */

int hondaCount = 0;

final Iterator<Vehicle> iter = list.iterator();
while (iter.hasNext()) {
    final Vehicle nextVehicle = iter.next();
    if (nextVehicle.getMakeModel().equals("Honda Civic")) {
        ++hondaCount;
    }
}
System.out.println
("There are " + hondaCount + " Honda Civics!");
```

The for-each loop
The for-each loop can be used with any Iterable container

- With our VehicleList

```
for (final Vehicle v : list) {
    System.out.println(v.getMakeModel());
}
```

Read this as "for each v in list"
• With arrays

```java
int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (final int x : arr) {
    System.out.println("Value: " + x);
}
```

12.7 A generic list class

Creating a generic list class

We can make `VehicleList` generic by adding a type parameter and revising the code (and tests) accordingly

• General class `GenericList` with type variable `SomeType`

• Any method dealing with the data replaces `Vehicle` with the `GenericList` class’s type parameter `SomeType`

• Any method that doesn’t use the data stays exactly the same

From `VehicleList`...

```java
public class VehicleList {
    public void add(int idx, Vehicle v) {
        addBefore(getNode(idx), v);
    }
    public void addBefore(Node p, Vehicle v) {
        final Node newNode = new Node(v, p.prev, p);
        newNode.prev.next = newNode;
        p.prev = newNode;
        ++size;
    }
    // ...
}
```

...to `GenericList`

```java
public class GenericList<SomeType> {
    public void add(int idx, SomeType e) {
        addBefore(getNode(idx), e);
    }
    public void addBefore(Node p, SomeType e) {
        final Node newNode = new Node(e, p.prev, p);
        newNode.prev.next = newNode;
        p.prev = newNode;
        ++size;
    }
    // ...
}
```
Using a generic list

When we instantiate a generic list, we must provide a type argument

```java
GenericList<Vehicle> lot = new GenericList<Vehicle>();
GenericList<Integer> numbers = new GenericList<>();
```

- In the second line Java can work out the argument type from other information — but that information must be present
- Adding an object to a generic list using `add` or `set` is checked by the compiler to ensure type compatibility
- Accessing an object from a generic list using `get` or `remove` carries the declared type of the list

Iterating for a generic list

The `Iterable` and `Iterator` interfaces share the type parameter of the list itself

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

public interface Iterator<T> {  
    public boolean hasNext();  
    public T next();  
    public void remove();
}

public class GenericList<SomeType>
    implements Iterable<SomeType> {  
    //...

    public Iterator<SomeType> iterator() {  
        return new GenericListIterator();
    }

    private class GenericListIterator  
        implements Iterator<SomeType> {  
        //...  
        public SomeType next() {  
            pos = pos.next;  
            return pos.data;
        }
    }
}
```

Generic sorted lists

Recall our sorted list class:
public class SortedVehicleList {
    // ...
    public void add(final Vehicle v) {
        VNode pos = head.next;
        while (pos != tail && v.compareTo(pos.data) >= 0) {
            pos = pos.next;
        }
        addBefore(pos, v);
    }
}

Can we make it generic in the same way?

public class SortedList<T> {
    // ...
    public void add(final T item) {
        Node pos = head.next;
        while (pos != tail && item.compareTo(pos.data) >= 0) {
            pos = pos.next;
        }
        addBefore(pos, item);
    }
}

• No! We cannot just assume that any class T implements Comparable

Type variable bounds
We can impose bounds on type variables

• A simple example first:

    public interface Plastic {
        public String color();
    }

• Then we can make a version of Box which holds only objects which conform to Plastic

    public class PlasticBox<T extends Plastic> {
        // ...just like Box...
    }

• Java will enforce this bound when we describe a particular PlasticBox

    public class FoodWrap implements Plastic {
        // ...
    }
    public class AluminumFoil {
        // *Not* Plastic
        // ...
    }

140
This declaration compiles:

```java
PlasticBox<FoodWrap> p;
```

This declaration is erroneous:

```java
PlasticBox<AluminumFoil> p;
```

**Bounds and type arguments**

Comparable is more complicated than Plastic

- Comparable requires a type argument
- What argument should it take?
  - In other words: what type of values do we want to compare to?
- When we think of comparison and total orders, we think about comparing similar things
  - We'd compare an Integer to an Integer
  - We'd compare a String to a String
  - We would *not* compare a String to an Integer
  - Remember the laws for these things!
- So the elements of a SortedList must be comparable to themselves
  - We could use the type variable again in the bound
    ```java
    public class SortedList<T extends Comparable<T>> { }
    ```
  - Is this good enough?

**With Vehicle**

We could now define a sorted list of Vehicle using the generic type

- We’ve already made Vehicle implement Comparable<Vehicle>
- So Vehicle satisfies the bound
- Progress!

**With Car**

What if we wanted a SortedList<Car>?

- We can certainly use compareTo on two Car instances
- So we should be able to make a sorted list of them
- But Car not satisfy the bound!
  - Doesn’t Car implement Comparable<Car>?
  - No — Car implements Comparable<Vehicle>!
- So Car does not fit the pattern T implements Comparable<T>
  - And a declaration
    ```java
    SortedList<Car> cars;
    ```
    will raise an error
Why would `Car` be reasonable?
`Car`'s `compareTo` method can be applied to more than just `Car` instances

- **SortedList** would be fine with `Car` implementing `Comparable<Car>`
- But when `Car` implements `Comparable<Vehicle>`, we want `SortedList` to understand that the method is still OK for any `Car`
  - This is true because `Car` is a subclass of `Vehicle`
  - This would be true for *any* class for which `Car` is a subclass
  - Or in other words, for any superclass of `Car`
- We want to declare `SortedList` so that its type argument `T`
  - Can be compared to any superclass of `T`
  - We don’t care exactly what it’s comparable to, so long as it is a superclass of `T`
- Java lets us make such a declaration:

  ```java
  public class SortedList<T extends Comparable<? super T>> {
      // So now, when we declare
      SortedList<Car> cars;
      Java will notice
      1. That the type argument `Car` implements `Comparable<Vehicle>`, and
      2. That `Vehicle` is in fact some superclass of the argument `Car`
      So we’ve satisfied the constraint!
  ```

From the Java standard library
At first these type bounds seem like a strange and exotic idea

But we find them in a number of places in the Java libraries
- In enumerated types

  ```java
  EnumMap<K extends Enum<K>,V>
  EnumSet<E extends Enum<E>>
  ```

- In classes for event handling

  ```java
  EventListenerProxy<T extends EventListener> An abstract wrapper clas
  ```

- In methods for collections classes like `SortedSet`

  ```java
  Comparator<? super E> comparator()
  Returns the comparator used to order
  ```

(Screenshots from SDK Javadoc pages)
12.8 Stacks and queues

Ordered data structures
Basic linked lists allow us to:

- Collect a number of objects
- Access them via `get` and iterators
- Add and remove items

For an unsorted list, there are no constraints on these operations

But sometimes, constraints on data insertion and access are desirable

- Netflix queue
- Amazon wishlist
- Email inbox
- Browser history
- Edit history in a word processor

Stacks
Recall:

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

A stack is an abstract data type that serves as a collection of elements with two principal operations:

- `Push` adds an item to the top of the stack
- `Pop` removes an item from the top of the stack
- Elements inserted into a stack come out in the reverse order
- A stack is a last-in-first-out (LIFO) structure

Stacks in the real world

- Pez dispensers
- A stack of textbooks
• A stack of dishes

Basic stack operations
The stack ADT provides methods to
• Check if stack is empty
• Access the top object
• Push onto the top of the stack
• Pop off the top of the stack

Consider a stack of strings

Implementing the stack
How should we actually implement the stack? A template as a starting point:

```
public class StringStack {
    public StringStack() {
        // FILL IN
    }

    public boolean isEmpty() {
        return false;  // FILL IN
    }

    public String top() {
        return null;   // FILL IN
    }

    public void push(String s) {
        // FILL IN
    }

    public String pop() {
```
Implementing the stack via lists

Use a `GenericList<String>` behind the scenes

```java
public class StringStack {

    // Hidden from the stack user
    private final GenericList<String> list = new GenericList<String>();

    public StringStack() { }

    public boolean isEmpty() {
        return list.isEmpty(); // FILL IN
    }

    public String top() {
        return list.get(0); // Always constant time
    }

    public void push(String s) {
        list.add(0, s);
    }

    public String pop() {
        return list.remove(0);
    }
}
```

Implementing the stack

The `push` and `pop` methods will be the only way to modify the internal list

```java
public class StringStack {

    // Hidden from the stack user
    private final GenericList<String> list = new GenericList<String>();

    public StringStack() { }

    public boolean isEmpty() { return list.isEmpty(); }

    public String top() { return list.get(0); // Always constant time
    }

    public void push(String s) { list.add(0, s); }

    public String pop() { return list.remove(0); }
}
```
Implementing a generic stack

As with our generic list, to make a generic stack it suffices to just parameterize the content type

```
StringStack

«constructor» + StringStack()
«query» + boolean isEmpty() + String top()
+ void push(String) + String pop()

GenericStack<SomeType>

«constructor» + GenericStack()
«query» + boolean isEmpty() + SomeType top()
+ void push(SomeType) + SomeType pop()
```

Using the generic stack

Executing code:

```java
GenericStack<String> stack = new GenericStack<String>();
for (int i = 0; i < 5; ++i) {
    stack.push("Word " + i);
    System.out.println("Top: " + stack.top());
}
while (!stack.isEmpty()) {
    System.out.println("Pop: " + stack.pop());
}
```

we get output:

```
Top: Word 0
Top: Word 1
Top: Word 2
Top: Word 3
Top: Word 4
Pop: Word 4
Pop: Word 3
Pop: Word 2
Pop: Word 1
Pop: Word 0
```

Queues

A queue is an abstract data type that serves as a collection of elements with two principal operations:

- **Enqueue** adds an item to the back of the queue
- **Dequeue** removes an item from front of the queue
- Elements inserted into a queue come out in the *same order*
- A queue is a first-in-first-out (FIFO) structure
Basic queue operations

The Queue ADT provides methods to:

• Check if queue is empty
• Access the front object
• Enqueue to add to the back
• Dequeue to remove from the front

Consider a queue of strings

• Just as for the StringStack, we can implement using a private GenericList

Towards a generic queue

To make a generic queue, we just need to parameterize the type

Implementing a generic queue

We hide the internal list from the user again:

public class GenericQueue<SomeType> {
    private final GenericList<SomeType> list = new GenericList<SomeType>();
    public GenericQueue() { }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public SomeType front() {
// Index 0 is always constant time
return list.get(0);
}
public void enqueue(SomeType s) {
    // Append to end
    list.add(s);
}
public SomeType dequeue() {
    // Remove from front
    return list.remove(0);
}
}

Using the generic queue

final GenericQueue<String>
    queue = new GenericQueue<String>();
for(int i=0; i<5; ++i) {
    queue.enqueue("Word " + i);
    System.out.println("Front: " + queue.front());
}
while (!queue.isEmpty()) {
    System.out.println("Dequeue: " + queue.dequeue());
}

Front: Word 0
Front: Word 0
Front: Word 0
Front: Word 0
Front: Word 0
Dequeue: Word 0
Dequeue: Word 1
Dequeue: Word 2
Dequeue: Word 3
Dequeue: Word 4

Bounded stacks and queues

We can limit the number of entries in a stack or queue by just checking if the container is full before adding any items; if it is, throw an exception!
Bounded queues via arrays

public class BoundedArrayStringQueue {
    private final String[] buffer;
    public BoundedArrayStringQueue(int capacity) {
        buffer = new String[capacity];
    }
}

// ...

• What else do we need to make this work:
  – Other fields to track?
  – How should the queue methods work?

Trace indices in the array

public class BoundedArrayStringQueue {
    private final String[] buffer;
    private int readNext=0, writeNext=0;
    public BoundedArrayStringQueue(int capacity) {
        buffer = new String[capacity];
    }
}

// ...

• What else do we need to make this work:
  – Other fields to track?
  – How should the queue methods work?
Start with the mutators

public class BoundedArrayStringQueue {
    private final String[] buffer;
    private int readNext=0, writeNext=0;
    // ...

    public void enqueue(String s) {
        if (isFull())
            throw new IllegalStateException("Cannot add to full queue");
        buffer[writeNext++] = s;
        if (writeNext == capacity) writeNext=0;
    }

    public String dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue empty");
        }
        final String result = buffer[readNext++];
        if (readNext == capacity) readNext=0;
        return result;
    }
}

When the indices line up

What does it mean when readNext==writeNext?

- It’s how we’ve set up the array at first
- But after if add capacity elements to an empty array, we’d see readNext==writeNext again!
- That condition cannot be our flag for both "empty" and "full"
- But it is associated with both "empty" and "full"
- We need an extra bit of state to let us distinguish the two

Empty and full

public class BoundedArrayStringQueue {
    // ...other fields as before...
    private boolean full=false;

    public boolean isEmpty() {
        return readNext == writeNext && !full;
    }

    public boolean isFull() {
        return full;
    }
}
public void enqueue(String s) {
    // Add this line after the rest
    if (writeNext == readNext) full=1;
}

13 Java collections

Collections
So far, we have created our own generic data structures:

• GenericList<T>
• GenericStack<T>
• GenericQueue<T>

These are all examples of collections

• A collection is an object that groups multiple elements into a single unit. It is used to store, retrieve, manipulate, and communicate aggregate data

Java provides this same functionality (and much more) within the java.util package, which includes numerous data collection classes that share a common interface

The Collection interface
In Java, the java.util.Collection<E> interface specifies common operations that any collection can do

• Add elements
• Remove elements
• Provide an iterator
• Check for membership

```java
public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object)
    Object[] toArray();
    Iterator<E> iterator();
    void clear();
    boolean add(E);
    boolean remove(Object);
    ...}
```
The Java collections framework

In addition to `Collection`, the `java.util` package provides interfaces for several common abstract data types (ADTs)

```java
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // optional
    boolean remove(Object element); // optional
    Iterator<E> iterator();
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // optional
    boolean removeAll(Collection<?> c); // optional
    boolean retainAll(Collection<?> c); // optional
    void clear(); // optional

    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

A collections framework is a unified architecture for representing and manipulating collections

Implementations in the collections framework

In order to make use of these interfaces, we need concrete classes that implement them

The `java.util` package provides several options

```java
public class ArrayList<E> implements Collection<E> {
    // Implementation...
}
```

```java
public class LinkedList<E> implements Collection<E> {
    // Implementation...
}
```

The Collection interface
Generics in the Collection interface

Many of the methods in the Collection interface are generic

```java
public interface Collection<E>
    extends Iterable<E> {
    //...
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);

    Iterator<E> iterator();

    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

- E is the type parameter for the collection
- T is the type parameter for this method
- The second toArray method checks at run-time that the elements of this collection conform to T
- Many of these methods are for backwards-compatibility with old versions of Java which did not have generics
  - Sun wanted as much old code as possible to continue to be valid
  - So the only method type changes were those necessary to ensure type integrity

Wildcards and bounds the Collection interface

Other methods in the Collection interface are specified in terms of wildcards

```java
public interface Collection<E> extends Iterable<E> {
    //...
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    //...
}
```

- ? represents any type
  - But sometimes this would cause runtime exceptions if it does not properly conform
  - So some uses must be bound
- The bound ? extends E guarantees that objects in the given collection conform to the element type E
  - And this can be checked at compile-time
"Optional" methods
Several of the methods in the Collection interface are marked *optional*

```java
public interface Collection<E> extends Iterable<E> {
    //...
    boolean add(E element); // optional
    boolean remove(Object element); // optional
    boolean addAll(Collection<? extends E> c); // optional
    boolean removeAll(Collection<?> c); // optional
    boolean retainAll(Collection<?> c); // optional
    void clear(); // optional
    //...
}
```

- The "contract" for using `java.util` collections classes does not require an implementations to support these methods
  - Compare with the "contracts" for `equals`, `compareTo`, etc.

- However, they *must* still be implemented
  - For example by throwing an `UnsupportedOperationException`  

**The List interface**

```java
public interface List<E> extends Collection<E> {
    // Access by index position
    E get(int index);
    E set(int index, E element); // optional
    void add(int index, E element); // optional
    E remove(int index); // optional
    boolean addAll(int index, Collection<? extends E> c); // optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

**Implementations of the List interface**

To *use* the `List` interface, we need an actual concrete class that implements it
Two basic approaches (with many variations):

- **Direct access** (ArrayList): Stores elements in an underlying array (a contiguous chunk of memory)
- **Linked access** (LinkedList): Stores elements in a chain of nodes (not necessarily contiguous)

### The `java.util.LinkedList<E>` class

The built-in `LinkedList<E>` class implements the `List<E>` and `Collection<E>` interfaces, and providing additional functionality as well.

### The `java.util.ArrayList<E>` class

The built-in `ArrayList<E>` class also implements the `List<E>` interface by using an array "under the hood".

### Iterator and ListIterator

We have seen the `Iterator` interface already:

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

The ListIterator interface is a subinterface of Iterator that also provides abilities to add/change objects and to move backwards

public interface ListIterator<T> extends Iterator<T> {
  public boolean hasPrevious();
  public T previous();
  public void add(T x);
  public void set(T newVal);
}

Finding elements in a collection
We have seen some basic patterns for searching for data in a linear collection (e.g., array, linked list):

- Linear search
  - Straightforward
  - Worst-case $O(n)$

- Binary search
  - Worst-case $O(\log_2 n)$
  - Requires sorted data and random access

- Can we do better?

A little bit of magic in our collections
Suppose we had access to a magic method:

- Given an element as input, it immediately returns the location (index) where the element should be located
- We could then go to that location and then determine whether or not the element is actually present
- This would transform search into a constant-time lookup!

What sorcery is this!?

Hash functions!

- Same idea used in library indexing
Computing a hash code
In addition to methods like `toString` and `equals`, every `Object` in Java has its own method for hashing:

```java
public int hashCode()
```

- Converts the *data fields* of an object (its state) to a single integer value

```java
final Integer obj1 = new Integer(2011);
System.out.println(obj1.hashCode());
final String obj2 = new String("2011");
System.out.println(obj2.hashCode());
2011
1537246
```

Overriding the `hashCode` method
Every class we write *inherits* a default implementation of `hashCode`

- Like the default implementation of `equals`, it is only *guaranteed* to return the same hash code for two objects if they are *identical* (same physical object in memory)

```java
final Person
    p1 = new Person("Tim"),
    p2 = new Person("Tim");
final int
    x1 = p1.hashCode(),
    x2 = p2.hashCode(); // Not guaranteed to equal x1!
```

But we can override this standard behavior

```java
public class Person {
    private String name;
    //...
    public int hashCode() {
        return 1000 + name.hashCode();
    }
}
```

Laws for `hashCode` values
An implementation of `hashCode` should adhere to these requirements:

- `hashCode` returns a consistent value every time it is invoked within a single application run (in particular, it must not be random)
- If two objects are equal according to `equals`, then they *must* return the *same* hash code
- If two objects are unequal according to `equals`, they are not *required* to return different hash codes
For you as a programmer, this means

- If you override `equals`, you really should override `hashCode`
  - But the compiler will not force you to do so

- If you override `hashCode`, you do not need to override `equals`

More info on the [Javadoc page for Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--)

**Using the magic hash function**

With `hashCode`, every object has its own integer code. This leads to the following idea:

- Store all data in a large array called a **hash table**
- To **store** an object, place it at the index matching its hash code
- To **look up** an object, search the index matching its hash code

![Hash Table Diagram]

**Some issues with `hashCode`**

Two objects with the same **content** should produce the same hash code:

```java
Integer obj1 = new Integer(2011);
Integer obj2 = new Integer(2011);
System.out.println(obj1.hashCode() + " : " + obj2.hashCode());
```

For example:

```
2011 : 2011
```

Two objects with different **content** could still produce the same hash code:

```java
String aStr = "Aa";
String bStr = "BB";
System.out.println(aStr.hashCode() + " : " + bStr.hashCode());
```

Conforming output could be:

```
2112 : 2112
```
Some issues with `hashCode`

So we cannot assume that every object has a unique hash code.

- Only so many integers to choose from (in machine representation)
- Two distinct objects **collide** if they possess the same hash code.
- Hash codes may also be **negative**

```java
String s = "1999999999999999999999";
System.out.println(s.hashCode());
```

gives:

-657850008

- We can still make the idea of hash table lookup work, but it requires a bit thought

Resolving collisions in a hash table

Given that collisions are possible, an effective hash table needs to be able to resolve them

- For each “bucket” (cell) in the array, store a **list** of objects that reside there
  
  – This is called **separate chaining**

- So our hash table can be implemented as an array of lists

- Another option for resolving collisions is to store a new entry at the first available cell after its hash index; this idea is known as **probing**

![Diagram of a hash table with separate chaining](image)

Finalizing the details of the hash table

For each object that needs to be stored in our hash table, we would:

- Compute the hash code

- Convert to an array index

- Go to that array index, and store object in the list at that location
The process for look-up is similar

- Overall running time is **bounded** by the size of the **largest** list in the array

```
0  0
1  81  1
2  64  4
3  25
4  36  16
5
6
7
8
9  49  9
```

### Implementing a hash table

```java
final LinkedList<Vehicle>[]
    table = new LinkedList[100]; // Compiler warning!
for (int i=0; i<table.length; ++i) {
    table[i] = new LinkedList<>();
}

// Store an element
Vehicle v = new Car("Honda Civic", 214118);
int code = v.hashCode();
int idx = Math.abs(code) % table.length;
table[idx].add(v);

// Look up
Vehicle t = new Truck("Ford F-150", 1234, 2);
int idx2 = Math.abs(t.hashCode()) % table.length;
if (table[idx2].contains(t)) {
    System.out.println("Truck is in table!");
}
```

### The java.util.HashSet Class

Rather than reinventing the wheel, we can take advantage of built-in collections for hashing, for example `${HashSet}$`

- Implements the **Collection** interface
- Implements the **Set** interface, so no duplicates allowed
  - `$add$` returns `false` if we try to store a duplicate
- The `$contains$` method is **very fast**
Java Utility HashSet

```java
java.util.HashSet<E>

«query» + int size()
+ boolean isEmpty()
+ boolean contains(Object)
+ Object[] toArray()
+ Iterator<E>
iterator()[0.5em]
«update» + void clear()
+ boolean add(E) + boolean
remove(Object) ...```

Using a HashSet

```java
final String
  s = "If I knew the answer I would tell you that I knew",
// Split on whitespace
final String[] words = s.split(" ");

HashSet<String> hs = new HashSet<String>();
for (String word : words) {
  hs.add(word);
}

System.out.println(" Total words: " + words.length);
System.out.println("Unique words: " + hs.size());
System.out.println(hs);

Prints:
  Total words: 12
Unique words: 9
[that, would, answer, knew, you, tell, the, If, I]

Objects get stored in whatever order their hash codes give

Using a HashSet

// Continued from previous
String s1 = "answer";
String s2 = "question";

System.out.println(hs.contains(s1));
System.out.println(hs.contains(s2));

true
false
```

- Programmer can quickly check if an object is present or not
- Does not need to know how the hash function works
- Saves a lot of time for larger sets (no loops)
The `java.util.HashMap` class

The `java.util.HashMap<K,V>` class implements the `Map<K,V>` interface

- But **not** the `Collection<E>` interface

- Similar to other collections, we can check size and if it is empty

- Need to use `put` to add a `(key, value)` pair
  - Returns the old value if updated

- Fast look-up via hashing

- **No** iterator access to the map directly; must ask for keys and/or values as a separate collection and iterate

```
java.util.HashMap<E>

... «query» + int size() +
boolean isEmpty() + boolean
containsKey(Object key) +
boolean containsValue(Object value) + V get(Object key) +
Set<K> keySet() + Collection<V>
values() + Set<Map.Entry<K,V»
entrySet()[] «update» + V
put(K key, V value) + V remove(K key)[0.2em]
```