Introduction to Java

# What is Java?

* ***language***: The programming language Java.
  + We write our Java programs in text (source) ﬁles.
* ***compiler***: We compile our source ﬁles with the javac compiler.
  + Creates Java byte code ﬁle(s).
  + Java byte code is an abstract machine language.
  + Can be “run” with a (Java) virtual machine (vm).
* ***vm***: We execute/interpret the byte code with the java vm.
  + This “runs” the program.

# State and Behaviour

* Classes ***deﬁne*** your objects.
* Each class corresponds to an ***object type***.
* When you create a class, you think about:
  + **Knowing**: What the object knows.
    - The object’s ***instance variables*** deﬁne its memory.
    - Instance variables are also known as ***attributes***.
    - The object’s attributes deﬁne the object’s ***state***.
  + **Doing**: What the object does.
    - What object do “are” its (instance) ***methods***.
    - The object’s instance methods deﬁne its ***behaviour***.

# Class versus Object

* A class is not an object.
* A class constructs objects at run time.
* The class acts as a ***blueprint*** for the object.
* One class may deﬁne several objects (a.k.a. ***instances***).

# The Class

* Every class has a unique object that represents the class.
* These objects have attributes.
  + We call them class attributes.
* Class and instance attributes are not the same.
  + Each instance of the class owns its instance attributes.
  + Each class owns its class attributes.
  + The class to class attribute relationship is one-to-one.
  + The class to instance attribute relationship is one-to-many.
  + Means you cannot use class attributes to represent object state.
* An object can access its class’ class attributes.
* You declare a class attribute by adding the word static.
* There are also class methods (add static).

## *Example*

public class Beer {

private static final Double DISCOUNT = 0.50; // class attribute

private final String name; // instance attribute

private final double price; // instance attribute

// Constructor

public Beer( final double price, final String name ) {

this.name = name;

this.price = price;

}

// Instance Method

public void sell( ) {

final double sellingPrice = price - DISCOUNT;

System.out.println( "Selling " + name + " at " + sellingPrice );

}

// Class Method

public static void main( String[] args ) {

final Beer miDaza = new Beer( 5.00, "Mi Daza" );

final Beer redemption = new Beer( 5.60, "Redemption" );

System.out.println( "Selling beers with " + DISCOUNT + " discount." );

miDaza.sell( );

redemption.sell( );

}

}

# Objects and Classes

* Programmers construct their Java program from objects.
* Similar to a builder building a house from parts:
  + Doors; Windows; Walls; …
* Each part has its own function.
* The parts work together to form the house:
  + The house is the sum of the parts.
* The builder doesn’t have to construct the parts.
* All he does is composing them.

# Using Objects

* Objects are the ﬁrst citizens of Java programs.
* You make an object work by calling its methods.
* Each method is a sequence of instructions.
* You can call a method even if you don’t know its instructions.
* Each method provides a service.
  + The method performs the service when you call the method.
* Diﬀerent methods may provide diﬀerent services.

# Classes

* Each object belongs to a unique class.
* Diﬀerent objects may belong to diﬀerent classes.
* An object that belongs to a class is called an ***instance*** of the class.
* A class may have more than one instance.
* Each class has its own *Application Programming Interface* (api).
* The api describes how to use the class:
  + The names of the methods;
  + The types of the arguments;
  + The purpose of the arguments;
  + The return value;
  + Side eﬀects; …
* The api deﬁnes a common protocol.
* Diﬀerent classes may have diﬀerent apis.

# Variables

* Most programs require computations.
* A single computation may require many sub-computations.
* You (usually) store the results of a computation in a ***variable***.
* A variable has several properties:
  + A name;
  + A memory location to store its value;
  + Its current value.
* To change a variable’s value, you assign it a new value.
* Before you can use a variable, you must declare it.
* A variable declaration determines:
  + The variable’s name;
  + The variable’s type (the kind of its values);
* A variable declaration may also determine the initial value;

# Assignment and Equality

* In mathematics you use = for equality.
* In Java you use = for assignment.
* But assignment and equality are not the same.
* The symbols are the “same” but they don’t mean the same.
* Mathematical equality is commutative: if a = b, then b = a.
* However, you can’t write the following in Java:

1 = a; // ?

* In mathematics a = a+1 is impossible.
* However, writing the following is valid in Java.

counter = counter + 1;

# Types

* A type is a collection of related values.
* E.g. Java has a whole range of numeric types.
  + **whole numbers**
    - byte;
    - short;
    - int;
    - long.
  + **ﬂoating point**
    - float;
    - double.
* If you want to assign a value to a variable, **The value must be in the the variable’s type**.
* This avoids logical errors:
  + Dog dog = new Cat( "Felix" );?
  + Debit debit = new Credit( 666 );?
* For whole numbers, the type int is usually a good.
* For ﬂoating point numbers, use double.

# Primitive Types and Object Reference Types

* A type starting with a lowercase letter is a primitive type. E.g. int, bool, char, float, …
* These values are bit patterns with well-deﬁned operations.
* Types starting with an uppercase letter are object/reference types. Dog, Cat, …
  + Object reference values reference objects.
* Primitive type values don’t reference objects;
* Java also has numeric object reference classes.
  + Best view these types as wrapper classes for primitive type values.

# Wrapper Classes

* Java has a wrapper class for each primitive type.
  + **Integer** For ints:
    - final Integer iRef = new Integer( 42 );
    - final int val = iRef.intValue( );
  + **Double** For doubles:
    - final Double dRef = new Double( 3.14 );
    - final double val = dRef.doubleValue( );
  + **Boolean** For booleans:
    - final Boolean bRef = new Boolean( true );
    - final boolean val = bRef.booleanValue( );

# Autoboxing and Unboxing

* Writing code to convert to and from wrapper classes is tedious.
  + new Integer( 42 ), lucky.intValue( ), …
  + You must type more.
  + It increases the code size.
* That’s why Java automates (some) conversions.
  + Automatic conversion to the wrapper class is called **autoboxing**.
  + Automatic conversion from the wrapper class is called **unboxing**.
* The conversion is done at runtime.

# Autoboxing

* Let val be an value with primitive type type.
  + If you use val and Java expects an object, Java will autobox val.
* The type of val determines the wrapper class:
  + int → Integer;
  + double → Double;
  + boolean → Boolean;

Integer lucky = new Integer( 42 );

Integer lucky = 42; // autoboxing

Double devil = 666; // doesn’t work

# Unboxing

* Unboxing turns wrapper class objects to primitive type values.
* The wrapper class type determines the primitive type.
  + Integer 7→ int;
  + Double 7→ double;
  + Boolean 7→ boolean; …
* The conversion is done at runtime. Java

int multiplicand = lucky.intValue( );

int multiplier = lucky; // unboxing

int square = multiplicand \* multiplier;

# Caching

* Java caches a limited number of wrapper class values.
* Guarantees shallow equality for small number of boxed values.
  + If o1.equals( o2 ) then o1 == o2.
* For example, new Integer( 0 ) == new Integer( 0 ).
* In general this may not always work:
  + Almost always: new Integer( 666 ) != new Integer( 666 ).
* Caching is implemented because it saves memory.
* In general caching works for “small” primitive values: Boolean, byte, char, short, int.

# Constant Variables

* A constant (variable) can only be assigned a value once.
* You declare a constant by adding the keyword **final**.

final int ANSWER = 42;

* Making a variable constant is a form of documentation.
* It lets the compiler help you detect logic errors.
* You cannot use an unassigned variable in a method.

# Comments

* A comment is text that is ignored by the compiler.
* Comments have several purposes:
  + They describe the purpose of a variable or a method.
  + They describe a relationship between two or more variables.
    - This is called an **invariant**.
  + They are used to create api documentation.
* You should always document your programs.
* One line Comments: // …..
* Multi-Line Comments: /\* …. \*/
* JavaDoc Comments: /\*\* …. \*/

# Variable Names

* Use names that are meaningful.
* The name should describe the variable’s purpose.
* By convention each variable name should be a noun.
  + **non-constant**
    - Each name should start with a lowercase letter.
    - The rest should be letters and digits.
    - At word boundaries, you use an uppercase letter.
    - All other letters should be lowercase.
    - E.g. sum, currentColour, …
  + **constant**
    - Use sequences of words, digits, and underscores.
    - Each word is spelt with uppercase letters.
    - At word boundaries, you use an underscore.
    - E.g. CENT, CENTIMETRES\_PER\_INCH, ….
* Variable names should be descriptive.
* This is a form of documentation:
  + It helps you remember what the variable does.
  + It helps others understand the purpose of the variable.
* Choosing a good name helps you understand the purpose.
* Always think about the purpose a variable should have.
* When you know the purpose, the name will follow.
  + A counter variable: int counter;
  + A bank account: Account account;
  + The wheel of a unicycle: Wheel wheel;
* If you can’t ﬁnd a proper name for a variable:
  + You don’t really know its purpose.
  + You may as well get rid of the variable.

# Working with Objects

* Before you can use an object, you must construct (create) it.
* To construct an object, you call its constructor.
  + The constructor constructs and initialises the object.
* There may be diﬀerent ways to construct an object.

final Rectangle bar = new Rectangle( x, y, width, height );

* 1. The new operator creates memory to represents the object;
  2. The constructor uses its arguments to initialise the object;
  3. The constructor returns a reference to the object;
  4. The reference is assigned to the object reference value bar.
  5. The reference may be used to call the object’s instance methods.

# Method declarations

* To deﬁne/declare a method you provide:
  + The name of the method;
  + The return type;
  + The names and types of the formal parameters;
  + The types of the formal parameters.
* You use void for a method without return value.
* If the argument types are diﬀerent, the names may overlap.
  + This is called **overloading**.

# Accessor and Mutator Methods

* A method that returns information about an object without modifying the object is an **accessor method**.
  + double width = rectangle.getWidth( );
* A method that modiﬁes an object’s instance variables is a **mutator method**.
  + rectangle.setWidth( 4.0 );

# State and Behaviour

* How do we implement the class?
* We must determine what the class instances do and know.
* What the instance does is its **behaviour**.
  + Object behaviour is implemented as **instance methods**.
* What the instance knows is its **state**.
  + Object state is implemented as **instance variables**.
* An object’s behaviour should determine its state:
  + Never, ever start with object state.
  + Start thinking about the behaviour.
  + If behaviour requires state, you implement the state.
  + Otherwise, you don’t.

# Tally Counter Class

// Class for representing tally counter objects.

public class Counter {

// The current tally counter value. Instance attribute declaration.

private int value;

// Returns the current counter value. Instance method declaration.

public int getValue( ) {

return value;

}

// Increment the counter value. Instance method declaration.

public void incrementValue( ) {

value = value + 1;

}

}

# Instance Variables

* Each Counter object has its own value variable.
* Let’s assume tally is a Counter object reference (variable):
  + To access its value you write tally.value.
* The Counter object owns the variable.
* Diﬀerent Counter objects may have diﬀerent values for value.

# Instance Methods

* Counter objects can call Counter instance methods.
* Calling them is similar to accessing the instance variable:
  + tally.incrementValue( );
  + int current = tally.getValue( );

# Encapsulation

* Objects should be self-governing.
* They should control their own instance variables.
* An object is self-governing if its instance variables are private.
* This is called *hiding* the instance variables.
  + Variable hiding prevents direct variable access by external clients.
* Hiding the instance variables makes the object self-contained.
  + It’s as if the object’s instance variables are in a capsule.
  + This is why instance variable hiding is usually called *encapsulation*.

# Why do we need Encapsulation?

* Direct attribute access is unsafe/dangerous
  + A malicious external agent may corrupt the attribute’s value.
* Encapsulation simpliﬁes the complexity of the api.
  + Makes learning, using, designing, reasoning, testing and maintaining the api easier.

# Contract

* We hide all instance variables.
* We hide all methods that aren’t/shouldn’t be part of the api.

# Hidding Methods

* Java also lets you hide method declarations. Java

public int squareOfAnswer( ) {

return answer( ) \* answer( );

}

private int answer( ) {

return 42;

}

* Hiding methods has similar advantages as hiding attributes.

# Automatic Variables

* A variable that is declared in a method is called automatic.
  + It only lives for the lifespan of its block during its method call.
* Use automatic variables for intermediate computations.
* **Don’t use instance attributes for intermediate computations.**

# Arrays

* Arrays are a special data type in Java.
* Arrays are **objects** that contain other things.
* There are two kinds of arrays:
  + Arrays consisting of primitive type values;
  + Arrays consisting of object reference values;
* The type of the array determines the type of its values.
* Before you can use an array you must create it (it’s an **object**).
  + When doing this, you must specify the array’s length.
  + The length remains ﬁxed.
* You can put things into the array.
* You can retrieve things from the array.
* You can only access arrays with index values:
  + Only int index values are allowed.
  + They must be non-negative;
  + They must be smaller than the length of the array.

## Initialisation

final int[] numbers = new int[ 10 ];

System.out.println( "length of numbers: " + numbers.length );

final String[] words = new String[ 5 ];

System.out.println( "length of words: " + words.length );

# Getting Stuff from the Array

* An array is best viewed as a tray/sequence with cups.
* Each cup has a number: 0, 1, …
* The cups contain what’s in the array: Object references.
* The number of cups is the length of the array.
* Let array be a Java array.
* Then array[ i ] is the ith cup of array.

final int[] numbers = new int[ 10 ]; …

System.out.println( "The first value is " + numbers[ 0 ] );

System.out.println( "The last value is " + numbers[ 9 ] );

# Writing Stuff to the Array

* The notation array[ index ] works just as with getting.
* Cups in the arrays work just like variables, so
  + array[ index ] = value assigns a value to the “indexth” cup.

final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;

numbers[ 9 ] = 42;

System.out.println( numbers[ 0 ] + " == 1" );

System.out.println( numbers[ 9 ] + " == 42" );

# Default Values

* When the jvm creates an array, it initialises the array’s contents.
* Each cup in the array is ﬁlled with the same value.
* This value depends on the type of the array:
  + Numeric 0;
  + boolean false;
  + char ’\u0000’;
  + Object null.

# Arrays do not Grow

* The length attribute of a Java array is final.
* So you cannot assign values <array>.length.
* The minimum size of any array is 0.
* The maximum size of any array is Integer.MAX\_VALUE.

# Partially Filled Arrays

* You must ﬁll the array before you can use it.
* You usually start ﬁlling at the bottom (index 0).
* Then ﬁll the next position (index 1) and so on.
* You need a counter to keep track of the current index.

## *Example*

final Scanner scanner = new Scanner( System.in );

final int[] values = new int[ scanner.nextInt( ) ];

int size = 0;

int next = 0; // We need this to enter the loop.

while ((size != values.length) && (next >= 0)) {

System.err.println( "Next value (negative value to stop): " );

next = scanner.next( );

if (next >= 0) {

values[ size++ ] = next;

}

}

final double percentage = 100.0 \* size / values.length );

System.out.println( "Percentage filled is " + percentage );

# Common Errors

* Index too Large, bigger than the array’s length.
* Index too Small (-1).
* Uninitialised Values.

# Representing Bank Accounts

* Consider a bank account application.
* Each account has an owner and a balance.
  + We could represent the owners using a String array;
  + We could represent the balance using a double array.

## Parallel Array Implementation

public class AccountManager {

private final String[] owners;

private final double[] balances;

public AccountManager( final int size ) {

final Scanner scanner = new Scanner( System.in );

this.owners = new String[ size ];

this.balances = new double[ size ];

for (int index = 0; index != size; index++) {

owners[ index ] = scanner.next( );

balances[ index ] = scanner.nextDouble( );

}

}

…

}

## Class-Based Implementation

public class AccountManager {

private final Account[] accounts;

public AccountManager( final int size ) {

final Scanner scanner = new Scanner( System.in );

accounts = new Account[ size ];

for (int index = 0; index != size; index++) {

final String owner = scanner.next( );

final double balance = scanner.nextDouble( );

accounts[ index ] = new Account( owner, balance );

}

}

…

}

## Comparison

* **Stability** The parallel array implementation is “unstable:”
  + Adding/removing attributes aﬀects api of all methods that depend on them.
* Security The parallel array implementation is not safe:
  + Parallel array clients need access to all arrays:
    - withdraw( owners, balances, nr, amount );
    - This gives the client access to all account details.
    - They can even modify the array.
    - It violates encapsulation.
  + Direct access for Account clients:
    - account.withdraw( amount ).
  + Perhaps better to add service at AccountManager level:

public void withdraw( final Account account, final double amount ) {

if (<conditions are right>) {

account.withdraw( amount );

}

}

# Why do we Need Methods?

* Methods are interfaces of parameterised computations.
* Method calls provide reusable computations.
* Building blocks of complex computations.
* Calls are the only mechanism to change private variables.

# Parameter Taxonomy

* **Formal parameter**: A parameter in a method deﬁnition.
* **Actual parameter**: A parameter in a method call.

# The Pass-by-Value Mechanism

1. Create a fresh variable for each parameter.
2. For I from 1 to n (from left to right):
   1. Evaluate the ith actual parameter.
   2. Assign the result of the ith evaluation to the ith fresh variable.
3. Carry out statements in the method body.
4. Return result (if any).
5. Remove fresh variables.

# Storing the Value of a Temporary Variable

* Actual parameter values are stored on the stack.
  + When method is called, variables are created on top of stack
  + When method returns this scratch space is released.
* The stack also stores the values of local variables in blocks.
  + When block is entered, variable are created on top of the stack.
  + When control leaves the block this scratch space is released.

# Class Design: How To?

* When we design an application, how do we choose the classes?
* Once we’ve decided on the classes,
  + How do we choose the attributes, and
  + How do we choose the methods?
* The answer is in the problem speciﬁcation.

# Clues

* To ﬁnd classes: look for actors in the spec.
  + This works, because the actors correspond to the objects,
    - And each object is an instance of its class.
  + We may implement the object in a class named after the actor:
    - Toy and Toy;
    - Writer and Writer;
    - Dog and Dog; …
* The actors do things (verbs): these are the methods.
* The actors own things, these are the attributes.

# Example: Playing with Toys

* There are **hands** and **toys**;
* Each **toy** has its own **name**;
* Each **hand** has its own **type**: left or right.
* A **toy** is (either) used or free;
* Initially, each **toy** is free;
* A **hand** is (either) empty or full;
* Initially, each **hand** is empty;
* A **hand** can only take a free **toy**;
* A full **hand** cannot take any **toy**;
* When a **hand** takes a **toy**, the toy becomes taken;
* When a **hand** takes a **toy**, the hand becomes full;
* A **hand** can drop its **toy**;
* When a **hand** drops its **toy**, the **hand** becomes empty; and
* When a **hand** drops its **toy**, the **toy** becomes free.

## The Toy Class

public class Toy {

private final String name;

private boolean used;

public Toy( String name ) {

this.name = name;

used = false;

}

// Getter and setter methods omitted.

@Override public String toString( ) {

return "Toy[ name = " + name + " ]";

}

}

## The Hand Class

public class Hand {

public static final String LEFT = "left";

public static final String RIGHT = "right";

private final String type;

private Toy toy;

public Hand( String type ) {

this.type = type;

toy = null;

}

public void take( Toy toy ) {

if (isFull( )) {

// We cannot take a Toy if Hand is full.

System.err.println( "\*\* " + this + " is full." );

System.err.println( "\*\* Cannot take " + toy + "." );

} else if (toy.getUsed( )) {

// We cannot take a used Toy.

System.err.println( "\*\* " + toy + " is taken." );

System.err.println( "\*\* Cannot take it." );

} else {

// Take toy.

// Formally mark toy as used.

toy.setUsed( true );

// Make toy our current Toy.

this.toy = toy;

}

}

public void drop( ) {

if (isEmpty( )) {

// We can only drop a toy if we have one.

System.err.println( "\*\* " + this + " is empty." );

System.err.println( "\*\* Cannot drop any toy." );

} else {

// Drop our current toy.

// Formally mark toy as free.

toy.setUsed( false );

// Make hand empty.

toy = null;

}

}

public String getType( ) { return type; }

public boolean isEmpty( ) { return toy == null; }

public boolean isFull( ) { return !isEmpty( ); }

@Override

public String toString( ) {

return "Hand[ type = " + type + ", toy = " + toy + " ]";

}

}

## The main

private static final String GAME = "computer game";

private static final String PUZZLE = "puzzle";

public static void main( String[] args ) {

Hand left = new Hand( Hand.LEFT );

Hand right = new Hand( Hand.RIGHT );

Toy game = new Toy( GAME );

Toy puzzle = new Toy( PUZZLE );

left.take( game );

right.take( game ); // Results in error message

right.take( puzzle );

left.drop( );

left.drop( ); // Results in error message

}

# The for Statement

* Mainly used for bounded iteration.

for (<initialisation>; <condition>; <update>) {

<stuff>

}

* The statement starts by carrying out <initialisation>.
* Carries out <stuff> while <condition> holds.
* After each iteration <update> is carried out.

int digit; // Declare induction variable.

for (digit = 0; digit <= 1; digit++) {

System.out.print( "Next binary digit is " );

System.out.println( digit );

}

# The while Statement

* Mainly used for unbounded iteration.

while (<condition>) {

<stuff>

}

* This carries out <stuff> while <condition> holds.

final double initialBalance = 10000.0;

final double targetBalance = 20000.0;

final double interestRate = 5.00;

double balance = initialBalance;

int years = 0;

while (balance < targetBalance) {

years++;

final double interest = balance \* interestRate / 100.0;

balance = balance + interest;

}

System.out.println( "initial balance: " + initialBalance );

System.out.println( "target balance: " + targetBalance );

System.out.println( "years: " + years );

System.out.println( "balance: " + balance );

# The do-while Statement

do {

<statement>

} while (<condition>);

<statement>

while (<condition>) {

<statement>

}

# Adding Numbers

int i, sum;

i = 0;

sum = 0;

while (i < 100) {

i = i + 1;

sum = sum + i;

} // sum == 1 + 2 + ... + 100

# Invariants

* *Invariants* relate the values of the variables in your program.
  + **Concretize**: Makes relationships explicit (documentation).
    - This helps when writing the program.
  + **Correctness**: They may help you prove the program is correct.
  + **Maintenance**: They help you maintain your program.
* Good programmers state invariants as comments in programs.

# Linear Search

int index = 0;

// index <= array.length and

// !satisfies( array[ prev ] ) for 0 <= prev < index

while (index < array.length && !satisfies( array[ index ] )) {

index ++;

// index <= array.length and

// !satisfies( array[ prev ] ) for 0 <= prev < index.

}

// index <= array.length and

// (!satisfies( array[ prev ] ) for 0 <= prev < index) and

// (index >= array.length || satisfies( array[ index ] ))

# Example: Battleship - The Speciﬁcations

* Implement a Battleship-style game called Sink-a-dot-Com.
* Game is played on 7×7 grid.
* We’re sinking “dot.coms” instead of ships.
* Initially there are three dot.coms.
* Each dot.com occupies three cells on the grid.
* The program randomly places the dot.coms on the grid.
* While there are dot.coms left:
  + The program prompts the user to guess a cell.
  + The program reads in the user’s guess.
  + The program checks the cell against the dot.com positions.
  + Finally, the program takes an appropriate action:
    - If the guess is a kill then the dot.com is deleted.
    - If the guess is a hit then the cell is deleted.
    - Otherwise, the program reports a miss.

## Simpliﬁed Version

* We have only one dot.com.
* We represent it as a 3-valued int array.
* The values are location cell numbers.
* The location cells are consecutive numbers between 1 and 7.
* User now guesses location cells.
* If the user guesses right we announce a hit.
* If there are three hits the game ends.
* Otherwise we continue.

## Developing the SimpleDotCom Class

* Figure out what the class is supposed to do.
* List the instance variables and methods.
* Write *prep code* for the methods.
* Write *test code* for the methods.
  + Helps clarify what the methods need to.
  + Helps design the method api.
  + Test code acts as documentation/contract.
  + By writing test code early, we can use it straight away.
* Write *real code* for the methods: write the class.
* Debug and reimplement as required.

## Write Prep Code

PseudoCode

public String checkYourself( final String guess ) {

final int cell = <convert guess to int>;

final boolean found = <find cell in locationCells>;

<increment hits if found>;

return <use found and hits and return result as String>;

}

PseudoCode

private void setLocationCells( ) {

final int cell = <generate first cell number>;

<set locationCells to {cell, cell+1, cell+2}>;

}

## Write Real Code

public String checkYourself( final String guess ) {

final int cell = Integer.parseInt( guess );

final boolean found = findLocation( cell );

hits += (found ? 1 : 0);

return getResultAsString( found );

}

private void setLocationCells( ) {

final int maxStartValue = 1 + MAX\_CELL\_VALUE - CELLS\_IN\_DOT\_COM;

final int cell = 1 + generator.nextInt( maxStartValue );

for (int position = 0; position != CELLS\_IN\_DOT\_COM; position ++) {

locationCells[ position ] = cell ++;

}

}

private boolean findLocation( final int cell ) {

int position = 0;

boolean found = false;

while ((position != locationCells.length) && !found) {

found = locationCells[ position ++ ] == cell;

}

return found;

}

private static final String MISS\_MESSAGE = "miss";

private static final String KILL\_MESSAGE = "kill"

private static final String HIT\_MESSAGE = "hit";

private String getResultAsString( final boolean found ) {

final String result;

if (!found) {

result = MISS\_MESSAGE;

} else if (hits == CELLS\_IN\_DOT\_COM) {

result = KILL\_MESSAGE;

} else {

result = HIT\_MESSAGE;

}

return result;

}

## Debug and Reimplement as Required

private static final boolean TESTING = TRUE;

private static final long INITIAL\_DEBUG\_SEED = 0;

private static final long INITIAL\_SEED = (TESTING) ? INITIAL\_DEBUG\_SEED : (new Random( ).nextLong( ));

private final Random generator = new Random( INITIAL\_SEED );

public static void main( String[] args ) {

final SimpleDotCom dotCom = new SimpleDotCom( );

System.out.println( dotCom.checkYourself( "0" ) );

System.out.println( dotCom.checkYourself( "1" ) );

System.out.println( dotCom.checkYourself( "2" ) );

System.out.println( dotCom.checkYourself( "3" ) );

System.out.println( dotCom.checkYourself( "4" ) );

}

# Relation with Owner

* Java has *class* and *instance* methods.
  + For a class method, you put static in the declaration.
* It also has *class* and *instance* variables.
  + For a class attribute, you put static in the declaration.
* *Instance methods & instance variables* are owned by instances.
  + There is one method/variable per instance of the class (one-to-one).
  + To access the method/variable you need the instance.
  + Instance attributes are for representing object state.
  + Instance methods are for object behaviour.
* *Class methods & class variables* are owned by the class.
  + There is one method/variable per class (one-to-one).
  + To access the method/variable you need the class.
  + Class attributes are for representing class state.
  + Class methods are for “class” behaviour.
* The class-to-instance (attribute/method) relation is one-to-many.

# Encapsulation

* Consider an encapsulated (private) instance attribute, attr.
  + The attribute is only visible inside the class.
* Consider an instance, instance, of the deﬁning class, C.
* Statements can only access/reference/see instance.attr if
  + They are deﬁned inside C; and
  + They have reference to instance.
* There are two ways statements can reference to instance:
  + **Directly** They can access the object reference instance; or
  + **Indirectly** They’re in an instance method of C that was called using the object referrence instance.

# Notation for Class

* The notation for class methods depends on where “you” are.
* You may always write ‘<classi.hmethod>( <arguments> ).’
* In the deﬁning class you may write ‘<method>( <arguments> ).’
* Same for variables: you may always write ‘<class>.<variable>.’
* Inside the deﬁning class you may also write ‘<variable>.’

## Example

public class Inside {

public static int attribute;

public static void method( ) {

int var1 = attribute;

int var2 = Inside.attribute;

System.out.println( var1 + " = " + var2 );

}

}

public class Outside {

public static void method( ) {

// System.out.println( attribute ); // Not allowed.

System.out.println( Inside.attribute );

}

}

# Notation for Instances

* The notation for instance variables and methods is similar.
* You may always use ‘<reference>.<method>( <arguments> ).’
* You may use ‘<method>( <arguments> )’ in deﬁning class.
  + (Provided you’re in an instance method.)
* For attributes you may write ‘<reference>.<variable>.’
* But in the deﬁning class you may also write ‘<attribute>.’
  + (Provided you’re in an instance method.)
* The dotless notation is only allowed inside instance methods.
* Inside instance methods you use ‘this’ for the “current” object.
* Using ‘<instance variable>’ without dot-notation means
  + ‘this.<instance variable>.’
* For instance methods this is the same.
* So ‘<instance method>(<arguments>)’ means
  + ‘this.<instance method>(<arguments>).’

## Example

public class Inside {

private int attribute;

private static void classMethod( int var ) {

System.out.println( var );

}

public void instanceMethod1( ) {

classMethod( attribute );

instanceMethod2( );

}

public void instanceMethod2( ) {

classMethod( this.attribute );

}

}

public class Outside {

public static void main( String args[] ) {

Inside inside = new Inside( );

inside.instanceMethod1( );

inside.instanceMethod2( );

}

}

# Simulating Instance Methods Java

public class Simulation {

private int attribute;

public static void classMethod( Simulation current ) {

System.out.println( current.attribute );

}

public void instanceMethod( ) {

classMethod( this );

}

}

public class Main {

public static void main( String args[] ) {

Simulation simulation = new Simulation( );

// The following calls are effectively identical.

simulation.instanceMethod( );

Simulation.classMethod( simulation );

}

}

# Chair Wars

* Larry and Brad work for a huge it company.
* Larry is an old-fashioned procedural programmer.
* Brad is one of them modern oo programmers.
* A few weeks ago they had to implement a fancy application.
  + There were diﬀerent shapes on the screen.
  + Each had its own picture and its own sound.
  + The pictures could rotate and play their sound.
* The best programmer got a prize.
* Turns out Brad won a fancy chair because of his oo skills.
* Brad’s ﬁnal solution had ﬁve classes:
  + One Shape ***superclass*** for default, common shape behaviour.
  + A dedicated class for each actual shape.
  + Each dedicated class was a ***subclass*** of the Shape class.
  + All, except for Amoeba, inherited all behaviour from Shape.
  + Amoeba ***overrode*** behaviour for playSound, and rotate.
  + This let Amoeba objects do things diﬀerently.
* Larry thought Brad’s ﬁnal class had lots of duplicated code:
  + “Your classes have same code for playSound and rotate.”
  + “This makes it impossible to maintain your code.”
  + “For each change, you need to edit 4 classes.”
    - Editingnclass ﬁles is n times more work than editing 1 ﬁle.
    - Each edit increases the probability of errors: more errors.
* But then Brad explained his design with Inheritance.

# Inheritance

* There are two main advantages of **inheritance**:
  + Increases ability to reuse implementation eﬀort.
  + Separates class-speciﬁc from general code.
* Code is structured in classes so as to maximise reuse.
* **Common** code is put in a common, more **abstract** class.
* The common, more abstract class is called the **superclass**.
* The code in the superclass is shared by **subclasses**.
* The subclasses are more **speciﬁc**:
  + Subclass provides same functionality as its superclass.
  + If the superclass has a public method then so does the subclass.
  + Here, the subclass **inherits** the public method from its superclass.
  + However, the subclass functionality may be more **speciﬁc**.
    - E.g., the subclass may implement a method in a **diﬀerent** way.
    - Here, the subclass **overrides** the method of its superclass.
  + Subclasses may also have more speciﬁc, **additional** behaviour.
* A subclass is said to **extend** its superclass.

## Example

* Let’s suppose we have a Surgeon and a GP class.
* Let’s also suppose we have a Doctor class.
* **Both** Surgeons and GPs are Doctors; they are more speciﬁc:
  + A Surgeon is-a Doctor.
  + A GP is-a Doctor.
  + So the Surgeon and GP classes **extend** the Doctor class.
* **Both** have a method called treatPatient( ).
* **Both** have a property worksAtHospital (a boolean).
  + **Any** Doctor has it.
  + For a Surgeon it is true.
  + For a GP it is false.
* Surgeons and GPs diﬀer from Doctors **in general**:
  + **Surgeon**:
    - Has additional makeIncision( ) method
    - Has **special** implementation for treatPatient.
    - **Overrides** default treatPatient( ) implementation.
  + **GP**:
    - Has additional attribute makesHouseCalls.
    - Has additional method giveAdvice( ).
* We put the **more general** code in the Doctor class.
* This is the code that **any** Doctor should have:

public class Doctor {

public boolean worksAtHospital;

public void treatPatient( ) { // Default patient treatment. }

public void chargePatient( ) { // Let’s face it, they all do. }

}

public class Surgeon extends Doctor {

public Surgeon( ) {

worksAtHospital = true;

}

@Override

public void treatPatient( ) { // Specific patient treatment. }

public void makeIncision( ) { // Additional behaviour. }

}

public class GP extends Doctor {

public boolean makesHouseCalls;

public GP( boolean makesHouseCalls ) {

worksAtHospital = false;

this.makesHouseCalls = makesHouseCalls;

}

public void giveAdvice( ) { // Additional behaviour. }

}

# The Fota Challenge

public class Animal {

private final String picture;

private final boolean eatsGrass;

private final int hungerLevel;

public Animal( final String picture, final boolean eatsGrass, final int hungerLevel ) {

this.picture = picture;

this.eatsGrass = eatsGrass;

this.hungerLevel = hungerLevel;

}

public void eat( ) { // Default eating behaviour.

System.out.println( "Eating " + hungerLevel + " portions of " + food( ) + "." );

}

private String food( ) {

return (eatsGrass ? "grass" : "meat");

}

public void makeNoise( ) { } // Should be overridden.

public void roam( ) { } // Should be overridden.

public String toString( ) { <omitted>}

}

public class Hippo extends Animal {

private static final int HIPPO\_HUNGER\_LEVEL = 10;

private static final String HIPPO\_PICTURE = "hippo.jpg";

public Hippo( ) {

picture = HIPPO\_PICTURE;

eatsGrass = true;

hungerLevel = HIPPO\_HUNGER\_LEVEL;

}

// Inherits eating behaviour from Animal class.

@Override // Makes sure we actually override an existing superclass method

public void roam( ) {

System.out.println( "I’m Lazy: not roaming." );

}

@Override // Makes sure we actually override an existing superclass method

public void makeNoise( ) {

System.out.println( "Grunt." );

}

}

import java.util.ArrayList;

public class Main {

public static void main( final String[] args ) {

final ArrayList<Animal> animals = new ArrayList<Animal>( );

animals.add( new Dog( ) );

animals.add( new Cat( ) );

animals.add( new Hippo( ) );

for (Animal animal : animals) {

System.out.println( "next: " + animal );

animal.roam( );

animal.eat( );

animal.makeNoise( );

}

}

}

public class Canine extends Animal {

private static final boolean EATS\_GRASS = false;

public Canine( final String picture, final int hungerLevel ) {

super( picture, EATS\_GRASS, hungerLevel );

}

@Override

public void roam( ) {

System.out.println( "Roaming in my pack." );

}

}

public class Feline extends Animal {

private static final boolean EATS\_GRASS = false;

public Feline( final String picture, final int hungerLevel ) {

super( picture, EATS\_GRASS, hungerLevel );

}

@Override

public void roam( ) {

System.out.println( "Roaming alone." );

}

}

public class Dog extends Canine {

private static final int DOG\_HUNGER\_LEVEL = 4;

private static final String DOG\_PICTURE = "dog.jpg";

public Dog( ) {

picture = DOG\_PICTURE;

// eatsGrass is false by default.

hungerLevel = DOG\_HUNGER\_LEVEL;

}

// Inherits eating behaviour from Animal class.

// Inherits roaming behaviour from Canine class.

@Override public void makeNoise( ) {

System.out.println( "Arf. Arf." );

}

}

public class Cat extends Feline {

private static final int CAT\_HUNGER\_LEVEL = 1;

private static final String CAT\_PICTURE = "cat.jpg";

public Cat( ) {

picture = CAT\_PICTURE;

// eatsGrass is false by default.

hungerLevel = CAT\_HUNGER\_LEVEL;

}

// Inherits eating behaviour from Animal class.

// Inherits roaming behaviour from Feline class.

@Override

public void makeNoise( ) {

System.out.println( "Mew. Mew." );

}

}

## Why do we Need the Animal Class?

* We need it for inheritance, so we can:
  + Share common code, and
  + Deﬁne a common api for Animals.
* We need it for polymorphism, so we can:
  + Write code that will still work if we add subclasses.

## Some Classes Should Never be Instantiated

* We never wanted the Animal class to be instantiated.
* We want Cat and Dog objects, but not Animal objects.
* The spell abstract prevents classes from being instantiated.

public abstract class Animal { … }

* Now javac won’t let you instantiate abstract classes:

Animal animal = new Animal( );

## Subclasses can be Abstract Too

public abstract class Canine extends Animal { … }

# Abstract and Concrete Classes

* A class is abstract if it’s deﬁned with the keyword abstract.
* Otherwise it is concrete.
* You can still use abstract polymorphic reference variables.

Dog dog = new Dog( );

Cat cat = new Cat( );

Animal animal = dog;

animal = cat;

* But, you can only instantiate concrete classes.

Cat cat = new Cat( );

Animal dog = new Dog( );

* Instantiating an abstract base class array is also allowed.

Animal[] animals = new Animal[ 3 ];

# Abstract Methods

* Java also has *abstract methods*.
  + They are deﬁned in abstract classes,
  + They are deﬁned with the keyword abstract, and
  + They have no body.

public abstract void roam( );

## Why Have Abstract Methods

* Abstract classes must be *extended*.
* Abstract methods must be *overridden*.
  + They deﬁne the nature of the common protocol.
  + They don’t require a default implementation.
  + Saves you from forgetting to implement proper behaviour.

# Implementing Abstract Methods

* Abstract methods have no body.
  + They only occur in abstract classes.
  + They have no default behaviour.
* Each concrete subclass needs the behaviour for its api.
* Therefore, you have to implement the abstract method.
* You implement an abstract method by providing a body.
  + This is called *overriding* the method.
  + This may be done in any class on the shortest path from concrete class to the abstract class that deﬁnes the abstract method.
  + So, implementing in abstract subclasses is allowed.
  + Of course, a method may be overridden, and overridden, ….

## How does this Work? Implementing the Method

public abstract class Animal {

public abstract void makeNoise( );

}

public class Dog extends Animal {

@Override

public void makeNoise( ) { … }

}

## Example: The Barber Paradox?

* A barber shaves people who don’t shave themselves.
* There’s a small town with only one barber.
* Who shaves the barber?
* Sometimes you need to decide class/interface membership.
* For example, when a polymorphic variable’s type is too loose.

public class Person {

public static void main( String[] args ) {

final Barber barber = Barber.orderBarber( );

final Person person = new Person( );

person.shave( ); barber.shave( );

}

public void shave( ) {

final Person person = this;

if (person **instanceof** Barber ) {

final Barber barber = (Barber)person;

barber.shaveYourself( );

} else {

final Barber barber = Barber.orderBarber( );

barber.shave( person );

}

}

}

public class Barber extends Person {

private Barber( ) { }

public static Barber orderBarber( ) {

return new Barber( );

}

public void shaveYourself( ) {

System.out.println( "Shaving myself" );

}

public void shave( final Person person ) {

System.out.println( "Shaving person" );

}

}

# Deciding Subclass Membership

* The spell **reference instanceof Clazz** tests for class membership of Clazz.
* It returns:
  + true if reference references an instance of Clazz;
  + true if reference references an instance of a subclass of Clazz;
  + false otherwise (including when reference == null).
* The test also works for interfaces.

# Overriding toString( )

* The Object class deﬁnes public String toString( );
  + It’s an instance method.
* It should return a “meaningful” representation of its instance.
* Arguably most classes should override the method.
* It’s especially useful when testing.

public class Person {

private final String firstName;

private final String surname;

…

@Override

public String toString( ) {

return firstName + " " + surname;

}

}

public class Die {

private final Random generator; // not printed

private int faceValue;

…

@Override

public String toString( ) {

return Integer.toString( faceValue );

}

}

public class DataBaseConnection {

private final Database db;

private final long id;

private final Port port;

…

@Override

public String toString( ) {

return "DatabaseConnection[ id = " + id

+ ", db = " + db // ????

+ ", port = " + port

+ … // all attributes

+ " ]";

}

}

public interface Testable {

public String getTestOutput( );

}

public class Port implements Testable { … }

public class Database implements Testable { … }

public class DataBaseConnection implements Testable {

private final Database db;

private final long id;

private final Port port;

…

// Better!

@Override

public String getTestOutput( ) {

return "DatabaseConnection[ id = " + id

+ ", db = " + db.getTestOutput( )

+ ", port = " + port.getTestOutput( )

+ … // all attributes + " ]"l

}

}

# Overriding equals( )

* public boolean equals( Object object ):
  + Also deﬁned in the Object class.
* Method is supposed to test for deep equality.
* Easy if you know the base class of object:

public class Person {

private final String firstName;

private final String surname;

…

@Override

public boolean equals( Object object ) {

final Person that = (Person)object;

return this.firstName.equals( that.firstName )

&& this.surname.equals( that.surname );

}

}

* But what if you don’t know the base class?

public class Person {

private final String firstName;

private final String surname;

…

@Override

public boolean equals( Object object ) {

final boolean result;

if (object instanceof Person) {

final Person that = (Person)object;

result = this.firstName.equals( that.firstName )

&& this.surname.equals( that.surname );

} else {

result = false;

}

return result;

}

}

# Controlling Inheritance

* In Java a subclass inherits all public methods and attributes.
* This is useful but public methods may lead to problems.
  + E.g. what if a malicious subclass overrides a method?
* It’s clear that more control is needed.
* In Java you can restrict inheritance and method overriding:
  + If you make a class final you can’t extend it.
  + If you make a method final you can’t override it.

# Making the Class Final

public class Word {

public void word( ) {

System.out.println( "It is." );

}

}

public class Rebuttal extends Word {

// You can extend this class.

public void word( ) {

System.out.println( "Oh no it isn’t." );

}

}

public final class LastWord extends Rebuttal {

// You cannot extend this class.

…

@Override

public void word( ) {

System.out.println( "Oh yes it is." );

}

}

Why Make a Class Final?

* Inheritance Violates Encapsulation.
* With method overriding, client classes may change behaviour.
* Almost as bad as providing them direct attribute access.
* Here methods, not attributes, are exposed to modiﬁcation.
  + **Security**: Makes sure the class does what it should do.
    - An overridden method may misbehave.
      * Makes it impossible to enforce invariants.
    - A String should behave as a String.
      * Should be impossible to override this method.
  + **Maintenance**: Clients may start to rely on overridden behaviour.

# Making the Method Final

public class Example {

// You may not override this method.

public final void finalMethod( ) { … }

// You may override this method.

public void overridableMethod( ) { … } }

# Multiple Inheritance

* Let’s introduce Pets to our Animal class hierarchy.
* The Pets can beFriendly( ).
* Other animals don’t have beFriendly( ) behaviour.
* Our design should allow for polymorphic pet variables.

Adding Pets to our Fota Application

**Option I: Adding the Pet Method to the Animal Class**

* **Pros**: The are two main advantages:
  + All Pets will inherit Pet behaviour, and
  + Animal can act as a polymorphic type for Pets.
* **Cons**: There are also disadvantages:
  + We don’t have a proper Pet type.
  + Non-Pets will also get beFriendly( ) behaviour.
  + Still must override beFriendly( ) for Dog & Cat.
* **Conclusion**: Clearly the disadvantages outweigh the advantages.
* **Cause**: The Is-A test fails for non-Pets.

**Option II: As Option I but Make Animal Class Abstract**

* **Pros**: The advantages are better than before.
  + We can make all animals behave appropriately.
  + Animal can act as a polymorphic type for Pets.
* **Cons**: We still don’t have a proper Pet type. Must override beFriendly( ) in all concrete classes.
* **Conclusion**: This design is worse than Option I.
* **Cause**: The Is-A test fails for non-Pets.

**Option III: Put the Pet Method where It Belongs**

* **Pros**: The following are some advantages.
  + Deﬁnition of beFriendly( ) is where it belongs.
  + Implementing beFriendly( ) requires little eﬀort.
  + All animals behave appropriately.
* **Cons**: The following are some disadvantages.
  + We still don’t have a proper Pet type.
  + The befriendly( ) method isn’t abstract.
    - We can’t guarantee a consistent beFriendly( ).
  + We lose a proper polymorphic type for Pets.
* **Conclusion**: This design makes Pets diﬃcult to work with.
* **Cause**: Polymorphism is a requirement for most applications.

**Option IV: Two Superclasses for Pets**

* **Pros**: The following are the advantages.
  + The beFriendly( ) method is where it belongs.
  + Implementing beFriendly( ) requires little eﬀort
  + Guarantees consistent beFriendly( ) deﬁnitions.
  + Pet can act as a polymorphic type for pets.
* **Cons**: Java doesn’t allow multiple inheritance.
* **Conclusion**: This design is ideal but impossible.
* **Cause**: A decision by the Java language designers.

# Implementing Duck Games

* Joe works at SimuDuck™.
* SimuDuck™ specialises in duck pond simulation games.
  + These games involves lots of quacking and swimming ducks.
* Joe is in charge of SimuDuck™’s most popular game.
* The game is written in Java and is based on inheritance.

## What Has Gone Wrong?

* It was inheritance that was causing the problem.
  + The Duck class deﬁned the default fly( ) behaviour.
  + This was inherited by all Duck subclasses.
  + None of the subclasses overrode the behaviour.
  + Therefore all ducks had the default fly( ) behaviour.
  + Including RubberDucks.

What Should Joe Do?

**Should he override fly( ) in the RubberDuck Class?**

* If he did that he might have to duplicate code later.
  + For example, what if a WoodenDecoyDuck was added later?
  + RubberDuck and WoodenDecoyDuck were almost the same,
    - Yet shared no code….
* Of course he could introduce a common superclass.
  + But that would mean much work.
  + Also there was no guarantee that work would stop there.

**Should he Use Interfaces?**

* Joe really wants software that doesn’t change.
* He does realise that change is the only constant.
* Code changes should have little impact on existing code.
* That would save much time rewriting existing code.

First Design Principle

**Encapsulate what Varies**

* We’ve seen that inheritance didn’t work for Joe.
  + When the (Duck) superclass changes this aﬀects all subclasses.
* Interfaces cannot change but they have no implementation:
  + No code reuse.
* Encapsulate what Varies: Identify the aspects of your application that vary and separate them from what stays the same.
* We implement each aspect as a behaviour:
  + Implement separate classes for diﬀerent behaviour.
  + Lets us choose speciﬁc behaviour by selecting a speciﬁc class.
  + Reusing the implementation comes for free.
  + Separates the implementation: increases ﬂexability.

# Encapsulate what Varies

* With interfaces, all classes **implemented** Flyer and Quacker.
* This is what caused the code duplication.
* We’re going to encapsulate what varies:
  + We separate what varies: fly( ) and quack( ) behaviour.
  + We deﬁne a Flyer interface.
    - Encapsulate each diﬀerent fly( ) behaviour as separate class.
  + We also deﬁne a Quacker interface.
    - Encapsulate each diﬀerent quack( ) behaviour as separate class.
  + We reuse the behaviour in the actual Duck subclasses.
    - This is done using **delegation**.
    - (It involves a design pattern.)

# Second Design Principle

**Program to an Interface**

* We need to design classes that implement duck behaviour.
* Behaviour is **assigned** to speciﬁc Duck instance attributes.
  + Assigning behaviour can even be done at runtime.
* Program to an interface, not to an implementation.

# Programming to an Interface

* We use an interface (a supertype) for each behaviour.
  + Flyer, Quacker, ….
  + Speciﬁc classes implement speciﬁc behaviours.
  + We use instances of these classes to use the behaviour.
* Before we depended on an **implementation**:
  + Default or overridden **class** behaviour.
* Now we depend on an **interface**:
  + An **object** with a type.
* Clients are now **unaware** of actual type and class of object.
  + This greatly reduces subsystem dependencies.

# Implementing the MallardDuck

public class MallardDuck extends Duck {

public MallardDuck( ) {

super( new SqueekQuack( ), new FlyWithWings( ) );

}

@Override

public void display( ) {

System.out.println( "MallardDuck here...." );

}

}

# Implementing the MutableDuck

public class MutableDuck extends Duck {

public MutableDuck( ) {

super( new SqueekQuack( ), new FlyWithWings( ) );

}

public void setQuackBehaviour( Quacker quacker ) {

// Assumes quacker is public/not final now.

this.quacker = quacker;

}

public void setFlyBehaviour( Flyer flyer ) {

// Assumes flyer is public/not final now.

this.flyer = flyer;

}

@Override

public void display( ) {

System.out.println( "MutableDuck here...." );

}

}

# Inheritance versus Object Composition

* **Inheritance**: Lets us create subclasses: *white-box reuse*.
  + Subclass inherits superclass behaviour.
  + Subclasses can override superclass behaviour.
  + You get code reuse for free.
  + You cannot change behaviour at runtime.
  + Violates encapsulation.
    - Subclass may rely on superclass implementation.
    - Subclass may break when superclass is changed.
* **Composition**: Lets you *compose* classes: *black-box reuse*.
  + A client class may use an object.
  + You get code reuse but it takes more eﬀort.
  + Lets you change behaviour at runtime.
  + Respects encapsulation.
    - Helps encapsulated classes focus on a single task.

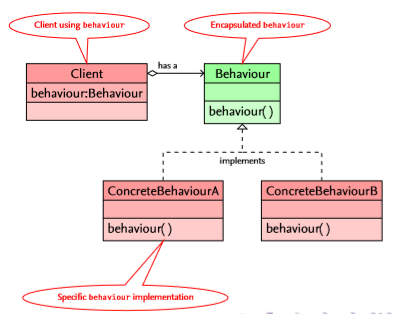
# Has-A can be better than Is-A: Third Design Principle

* In our new design we rely on Has-A (more then on Is-A):
  + Each Duck has-a flyer, and
  + Each Duck has-a quacker.
* “Has-A” lets us implement behaviour by *composing* classes.
* The result is a more ﬂexible design:
  + It lets us encapsulate behaviour.
  + We can change behaviour at runtime.

**Favour Composition over Inheritance.**

# The Strategy Pattern

* The *Strategy Pattern*:
  + Deﬁnes a class of algorithms;
  + Encapsulates each algorithm; and
  + Makes them interchangeable.
* Lets the algorithms vary independently from clients using it



# Overloading

* A subclass may *override* a public method from a superclass.
  + Allows subclasses to implement more speciﬁc behaviour.
  + E.g. diﬀerent rotate( ) behaviour in the Amoeba subclass.
* A method’s signature comprises its name and its argument types.
* Two methods with the same name may also overload each other.
  + Has nothing to do with inheritance.
  + The methods must have a diﬀerent signature.
    - E.g. diﬀerent numbers of arguments.
    - E.g. same number of arguments but at least one diﬀerent type.
* Class constructors may also overload each other.

## Example

public void f( int x ) { /\* stuff \*/ }

public int f( double x ) { /\* stuff \*/ }

private void f( int x, double y ) { /\* stuff \*/ }

public void f( double x, int y ) { /\* stuff \*/ }

# Constructor Overloading

* Class constructors may also overload each other.
* They may even call each other.
* When they do, you write this( … ) for the constructor call.
  + You don’t add new.
  + Calling this( … ) should be the ﬁrst call in a class constructor.
* Lets you implement easy client-friendly constructors.
  + A very general constructor does the work.
  + The friendly versions call this( … ).
  + The friendly version may also do additional conﬁguring.

## Example

public class NamedObject {

private static final String DEFAULT\_NAME = "Object";

private final String name;

// Default constructor

public NamedObject( final String name ) {

this.name = name;

}

// Special-purpose constructor

public NamedObject( ) {

this( DEFAULT\_NAME );

}

}

# Motivation for Interfaces

* Let’s assume you have a sorting algorithm.
* The algorithm works for certain kinds of objects.
* Let’s say it works for Integers.
* Let’s say you’d like to use the algorithm for Doubles.
* Ideally you’d like to reuse the algorithm’s implementation.
* But how?

## Example

public int linearSearch( final Comparable[] things, final Comparable key ) {

int index = 0;

while ((index != things.length) && (things[ index ].compareTo( key ) != 0)) {

index++;

}

return (index < numbers.length) ? index : -1;

}

# We Need a Contract

* To reuse the method, we need a contract.
* The contract restricts the type of parameter:
  + We must make sure the parameter has the behaviour we need.
* The contract restricts how the parameters may be used:
  + We’re only allowed to use certain kinds of instance methods.
* In Java the contract is called an interface.
* Using an interface is a multi-stage process;
  + You deﬁne the interface (once).
  + You implement the interface (any number of times).

# Deﬁning the Interface

* Deﬁning an interface is like deﬁning a class.
* You provide the name of the interface.
* You provide the api of the public instance methods.
* You don’t provide an implementation of the instance methods.
* Since there is no implementation, classes can’t rely on it.
* Let’s you build a loosly-coupled collection of classes.

## Example

public interface Sellable {

public double getPrice( ); /\* No Implementation \*/

public void sellTo( final Buyer buyer ); /\* No Implementation \*/

}

# Implementing the Interface

* Once you’ve deﬁned the interface, you may implement it.
* Implementing the interface may be done in any class.
* Implementing an interface means deﬁning its public methods.
  + This is called *overriding* the methods.
* Classes may implement as number of interfaces they like.

## Example

public class Cat implements Sellable {

private final double price;

private Buyer owner;

public Cat( … ) {

…

}

@Override

public double getPrice( ) {

return price;

}

@Override

public void sellTo( final Buyer buyer ) {

owner = buyer;

}

}

public class FreeBread implements Sellable {

private final double price;

private Buyer owner;

public FreeBread( … ) {

…

}

@Override

public double getPrice( ) {

return price;

}

@Override

public void sellTo( final Buyer buyer ) {

owner = buyer;

}

}

public static void main( Sting[] args ) {

final Cat cat = new Cat( "Felix" );

final FreeBread pan = new FreeBread ( "white", "chrunchy" );

final Buyer mary = new Buyer( "Mary" );

cat.sellTo( mary );

pan.sellTo( mary );

}

Substitution Principle

**Interface Version**

* Let’s assume we have an interface Interface.
* Let’s assume we have a variable Interface var.
* At runtime you may assign var any reference to an instance of a class that implements Interface.
* More generally, if a class implements Interface you may use its instances when an Interface object reference is expected.
  + This is called the Liskov substitution principle.
* So let’s assume the Dog class implements the Animal interface.
* Then you can use a Dog when Java expects an Animal.

Animal animal = new Dog( );

**Class Version**

* Let’s assume we have an class Clazz.
* Let’s assume we have a variable Clazz var.
* At runtime you may assign var any reference to an instance of a class that extends Clazz.
* More generally, if a class extends Clazz you may use its instances when an Clazz object reference is expected.
  + This is called the Liskov substitution principle.
* So let’s assume the Dog class extends the Animal class.
* Then you can use a Dog when Java expects an Animal.

Animal animal = new Dog( );

# Polymorphism

* The term polymorphism means
  + The occurrence of something in several, diﬀerent forms.

**Without Polymorphism**

* The type of reference variable and object are the same:

Dog animal = new Dog( );

**With Polymorphism**

* The type of reference variable and object may be diﬀerent:

Animal animal = new Dog( );

Animal animal = new Cat( );

* The reference type must implement the interface/extend the class.
* The type of the object, not the type of the reference, determines which instance method is called.
* This is also known as late binding.

Animal[] animals = new Animal[ 2 ];

animal[ 0 ] = new Dog( );

animal[ 1 ] = new Sheep( );

animal[ 0 ].makeNoise( ); // Barks

animal[ 1 ].makeNoise( ); // Bleats

# For a Polymorphic Method Deﬁnition

* Formal parameters and return types can be polymorphic.
* With formal parameter Animal the actual parameter may be Dog.
* Likewise, return type may be Animal but a Cat may be returned.

# Case Study

public interface Animal {

public void makeNoise( ); /\* No Implementation \*/

…

}

public class Cat implements Animal {

…

@Override

public void makeNoise( ) {

System.out.println( "Mew. Mew." );

}

}

public class Dog implements Animal {

…

@Override

public void makeNoise( ) {

System.out.println( "Arf. Arf." );

}

}

public class Hippo implements Animal {

…

@Override

public void makeNoise( ) {

System.out.println( "Grunt" );

}

}

public class Vet {

public void giveShot( Animal animal ) {

System.out.print( "Giving shot: " );

animal.makeNoise( );

}

}

public class PetOwner {

public static void main( String[] args ) {

Vet vet = new Vet( );

Animal[] animals = { new Cat( ),

new Dog( ),

new Hippo( ) };

for (Animal animal : animals) {

vet.giveShot( animal );

}

Animal animal = animals[ 0 ];

vet.giveShot( animal );

animal = animals[ 1 ];

vet.giveShot( animal );

}

}

# Delegation

* With interfaces we can simulate inheritance.
* It’s a lot more work but the resulting design may be better:
  + The interface has no implementation.
  + Great, so you can’t depend on an implementation.
  + Makes classes more interchangable.
* Relies on object compostion rather than inheritance.
  + (Has-a as opposed to is-a.)
* We can re-use existing implementation eﬀorts using delegation.
* To implement the interface in class C:
  + We need a concrete class that implements the interface.
  + C implements the interface using a concrete class instance.
  + C simply delegates the work to the concrete instance.
  + Usually C owns the concrete instance.

## Example

public interface Noisy {

public void makeNoise( );

}

public class ConcreteNoisy implements Noisy {

private final String sound;

ConcreteNoisy( final String sound ) {

this.sound = sound;

}

@Override

public void makeNoise( ) {

System.out.println( sound );

}

}

public class Dog implements Noisy {

// We use the polymorphic type @Noisy@, not @ConcreteNoisy@.

// That way we can only use @Noisy@ behaviour.

private final Noisy concreteNoisy;

public Dog( ) {

concreteNoisy = new ConcreteNoisy( "Arf. Arf." );

}

@Override public void makeNoise( ) {

// Here we delegate the noise making

concreteNoisy.makeNoise( );

}

}

public class Cat implements Noisy {

// We use the polymorphic type @Noisy@, not @ConcreteNoisy@.

// That way we can only use @Noisy@ behaviour.

private final Noisy concreteNoisy;

public Cat( ) {

concreteNoisy = new ConcreteNoisy( "Mew. Mew." );

}

@Override public void makeNoise( ) {

// Here we delegate the noise making

concreteNoisy.makeNoise( );

}

}

A class may only have one direct superclass.

A class may implement any number of interfaces.

# Multiway Branching

if (var == 0) {

// First stuff

} else if (var == 1 || var == 3) {

// Second stuff

} else if (var == 2 || var == 4) {

// Third stuff

} … {

} else {

// Final stuff

}

switch (var) {

case 0: // First stuff

case 1:

case 3: // Second stuff

case 2:

case 4: // Third stuff

…

default: // Final stuff

}

## Example

switch (character) {

case ’A’:

case ’B’:

case ’C’:

System.out.println( "Range: A--C." );

break;

case ’e’:

System.out.println( "It’s an ’e’" );

break;

default:

System.out.println( "It’s not in {A,B,C,e}" );

}

# The int-enum Anti-Pattern

* An enumerated type represent a related set of constants.
  + The seasons of the year;
  + The suits in a deck of cards;
  + The graduation levels (pass, 2h2, 2h1, 1h); ….
* Common, but ﬂawed, implementation that uses int constants.
* This technique is called the int enum pattern.
* Never, ever, ever, use it.

## Example

public static final int APPLE\_FUJI = 0;

public static final int APPLE\_PIPPIN = 1;

public static final int ORANGE\_NAVEL = 0;

public static final int ORANGE\_TEMPLE = 1;

public static final int ORANGE\_BLOOD = 2;

# Int Enums are Flawed

**Type safety: Int enums don’t provide type safety.**

if (APPLE\_FUJI == ORANGE\_BLOOD) {/\*??\*/ } //Comparing Apples and Oranges??

int apple = ORANGE\_BLOOD; //Value Out of Range!

* **Maintainability**: Programs with int enums are brittle.
  + Int enums are compile-time constants.
  + They are compiled into clients that use them.
  + Client will break if enum constant changes.
* **Ease of use**: Int enums are diﬃcult to use.
  + It is diﬃcult to translate them to Strings.
  + No reliable iteration over all allowed values.
* **Namespace**: Int enum types have no private name space.

## Example

public abstract class Beef {

public static final Beef SHANK = new Beef( ) {

@Override public double price( ) { return 1.0; }

};

public static final Beef SIRLOIN = new Beef( ) {

@Override public double price( ) { return 2.0; }

};

public abstract double price( );

private Beef( ) { }

public static void main( String[] args ) {

final Beef shank = Beef.SHANK;

final Beef sirloin = Beef.SIRLOIN;

…

}

}

# Java enums to the Rescue

* As of Release 1.5 Java provides the enum type.
* It overcomes most, if not all, shortcomings of int enums.

public enum Apple { FUJI, PIPPIN }

public enum Orange { NAVEL, TEMPLE, BLOOD }

* Each ‘public enum <class>{<constants>}’ is a class.
* Each constant in <constants> is an instance of the class: an object.
* For each constant in any enum class, Java automatically deﬁnes one public final class attribute.
* Name of <constant> in <class> is <class>.<constant>.
* All Java enum constructors are (implicitly) private.
* All instance methods are final, except for toString( ).

# Why enums are Good

**Type safety: Java enums are type safe.**

if (Apple.FUJI == Orange.BLOOD) { /\*??\*/} //Comparing Apples and Oranges??

Apple apple = Orange.BLOOD; //Value Out of Range!

* **Maintainability**:
  + enums aren’t compiled as constants into clients.
  + Rearranging values doesn’t break clients.
* **Ease of use**:
  + Translating to Strings is easy: toString().
  + Iterating over all enums is easy: values().
* **Namespace**: Enum classes have a private name space.

# Methods in enum Classes

* **compareTo( that )**: Compares this enum with that for order.
* **equals( that )**: Returns true if this enum equals that.
* **hashCode( )**: Returns a hash code for this enum.
* **toString( )**: Returns the name of this enum constant.
* **name( )**: Returns the original name of this enum.
* **ordinal( )**: Returns the ordinal of this enum.

# Java Enums are Objects

* int enums only have a value.
* Java enums are objects.
  + They have state.
  + They have behaviour.
* Makes Java enums much more ﬂexible.

# State and Behaviour

* Consider the eight planets of the solar system.
* Each planet has a mass and a radius.
* Using the mass and radius we compute the surface gravity.

## Example: Implementing the Planet Class

public enum Planet {

MERCURY( 3.303e+23, 2.439e6 ),

VENUS ( 4.869e+24, 6.052e6 ),

EARTH ( 5.975e+24, 6.378e6 ),

MARS ( 6.419e+23, 3.393e6 ),

JUPITER( 1.899e+27, 7.149e7 ),

SATURN ( 5.685e+26, 6.027e7 ),

URANUS ( 8.683e+25, 2.556e7 ),

NEPTUNE( 1.024e+26, 2.477e7 );

// Universal gravitational constant in m^3/kg s^2.

private static final double G = 6.67300E-11;

private final double mass; |

private final double radius; } State

private final double gravity; |

Planet( double mass, double radius ) {

this.mass = mass;

this.radius = radius;

gravity = G \* mass / (radius \* radius);

}

public double getMass( ) { return mass; } |

public double getRadius( ) { return radius; } }Behaviour

public double getGravity( ) { return gravity; } |

}

public class WeightTable {

public static void main( String[] args ) {

for (Planet planet : Planet.values( )) {

double weight = surfaceWeight( planet, 1.0 );

System.out.println( "1kg on " + planet

+ " has a surface weight of "

+ weight + "." );

}

}

private static double surfaceWeight( final Planet planet, final double mass ) {

return mass \* planet.getGravity( );

}

}

# Speciﬁc Behaviour

* Our Planet application is very well behaved.
* All method results depend on input and attributes only.
* This is not always the case.
* For example, consider a calculator application.
  + There are four operations PLUS, MINUS, TIMES, and DIVIDE.
  + We’d like to apply operations to doubles and get the result:
    - double apply( double first, double second ).
  + assertTrue( 1.00 == PLUS.apply( 0.0, 1.0 ) ) && assertTrue( -1.00 == MINUS.apply( 0.0, 1.0 ) ), ….
  + The result also depends on the enum constant.

# How do we Implement This?

**Constant-Speciﬁc Methods**

public enum Operation {

PLUS { @Override

public double apply( double x, double y ) { return x + y; } },

MINUS { @Override

public double apply( double x, double y ) { return x - y; } },

TIMES { @Override

public double apply( double x, double y ) { return x \* y; } },

DIVIDE { @Override

public double apply( double x, double y ) { return x / y; } };

public abstract double apply( double first, double second );

}

**Adding More Intuitive Printing**

public enum Operation {

PLUS { @Override

public String toString( ) { return "+"; }

@Override

public double apply( double x, double y ) { return x + y; }},

<rest of class omitted>

**Using the Operation Class**

public class Calculator {

public static void main( String[] args ) {

final double first = 6;

final double second = 2;

for (Operation op : Operation.values( )) {

double result = op.apply( first, second );

System.out.println( first + " " + op + " " + second + " = " + result );

}

}

}

**Getting Really Fancy Now??**

public enum Operation {

PLUS {

@Override

public String toString( ) { return "+"; }

@Override

public double apply( double x, double y ) { return x + y; }

}, MINUS {

@Override

public String toString( ) { return "-"; }

@Override

public double apply( double x, double y ) { return x - y; }

}, TIMES {

@Override

public String toString( ) { return "\*"; }

@Override

public double apply( double x, double y ) { return x \* y; }

}, DIVIDE {

@Override

public String toString( ) { return "/"; }

@Override

public double apply( double x, double y ) { return x / y; }

};

public abstract double apply( double first, double second );

}

**Factoring out Identical Behaviour**

public enum Operation {

PLUS( "+" ) {

@Override

public double apply( double x, double y ) { return x + y; }

}, MINUS( "-" ) {

@Override

public double apply( double x, double y ) { return x - y; }

}, TIMES( "\*" ) {

@Override

public double apply( double x, double y ) { return x \* y; }

}, DIVIDE( "/" ) {

@Override

public double apply( double x, double y ) { return x / y; }

};

public abstract double apply( double first, double second );

private final String symbol;

Operation( String symbol ) {

this.symbol = symbol;

}

@Override

public String toString( ) {

return symbol;

}

}

## Example: Payroll Application

* Employees have a pay rate that depends on their grade.
* Our application gets the pay rate as its input.
* An employee’s pay for a given day of the week is given by
  + pay = base pay+overtime pay for that day.
* The base pay is given by pay rate×hours worked.
* The overtime pay is given by overtime
  + pay = pay rate×overtime hours/2.
* **Weekdays**: Hours worked in excess of hours per shift (8).
* **Weekend**: Hours worked on that day.

**First Stab at Implementation**

public enum SimplePayrollDay {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

private static final int HOURS\_PER\_SHIFT = 8;

public double pay( double hoursWorked, double payRate ) {

double basePay = hoursWorked \* payRate;

double overtimePay = overtimePay( hoursWorked, payRate );

return basePay + overtimePay;

}

public double overtimePay( double hoursWorked, double payRate ) {

double overtime;

switch (this) {

case SATURDAY:

case SUNDAY: // Weekend

overtime = hoursWorked;

break;

default: // Weekday

double difference = hoursWorked - HOURS\_PER\_SHIFT;

overtime = (difference < 0 ? 0 : difference);

}

return overtime \* payRate / 2;

}

}

**What’s Wrong?**

* What if we add an extra type of day?
* For example, a Bank Holiday (special kind of Monday).
* We’d have to modify overtimePay( ).
* The application will break if we forget to make the change.

**How to Fix It?**

* We need diﬀerent strategies for paying overtime.
* Strategy for toString( ) in our computation is 100% shared.
* With the payrole application some strategies are shared, not all.
* Currently we have two strategies.
  + Each is determined by the kind of day: week days/weekend days.
  + The kind of day is a property of the day.
  + A property can be implemented as an attribute.
  + The attribute now determines the kind of day:
    - We can compute the kind of day from the attribute.
    - The kind of day determines the strategy.
    - Therefore, the attribute determines the strategy.
* We could implement our attribute as a boolean: isWeekday.
  + This would work now, but the requirements may change:
    - Double overtime rate for Christmas days?
* Probably better to have a strategy enum type.
  + The new strategy determines overtime pay computation.
* (Of course we implement it as an inner (enum) class.)

**A Better Implementation**

public enum PayrollDay {

SUNDAY( PayType.WEEKEND ),

MONDAY( PayType.WEEKDAY ),

TUESDAY( PayType.WEEKDAY ),

WEDNESDAY( PayType.WEEKDAY ),

THURSDAY( PayType.WEEKDAY ),

FRIDAY( PayType.WEEKDAY ),

SATURDAY( PayType.WEEKEND );

private static final int HOURS\_PER\_SHIFT = 8;

private final PayType type;

PayrollDay( PayType type ) { this.type = type; }

public double pay( double hoursWorked, double payRate ) {

double basePay = hoursWorked \* payRate;

double overtimePay = type.overtimePay( hoursWorked, payRate );

return basePay + overtimePay;

}

private enum PayType {

WEEKEND {

@Override

public double overtimePay( double hoursWorked, double payRate ) {

return hoursWorked \* payRate / 2;

}

}, WEEKDAY {

@Override

public double overtimePay( double hoursWorked, double payRate ) {

double difference = hoursWorked - HOURS\_PER\_SHIFT;

double overtime = (difference < 0 ? 0 : difference);

return overtime \* payRate / 2;

}

};

public abstract double overtimePay( double hoursWorked, double payRate )

}

}

**Why Strategy enums are Good for You**

* The overtime pay computation is what varies.
* The strategy enum isolates what varies.
* *Localises* the code for overtime pay computation.
* *Global* change in rules translates to local change in code:
  + Easy to remove days and strategies.
  + Easy to change strategies.
  + Easy to add new days for existing strategies.
  + Easy to add new days and new strategies.

public enum PayrollDay {

…

BANK\_HOLIDAY( PayType.BANK\_HOLIDAY ),

…

private enum PayType {

…

BANK\_HOLIDAY {

@Override

public double overtimePay( double hoursWorked, double payRate ) {

return hoursWorked \* payRate;

}

…

}

}

## Example: A Music Application

**Don’t Try This at Home**

public enum Ensemble {

SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET, DECTET;

public int size( ) { return 1 + ordinal( ); }

}

* This class will break if:
  + Constants are re-ordered.
  + Constants are removed.
  + Constants are added and there are “holes.”
  + Constants are added with the same size as existing ensembles.

**A Better Approach**

public enum Ensemble {

SOLO( 1 ), DUET( 2 ), TRIO( 3 ), QUARTET( 4 ),

QUINTET( 5 ), SEXTET( 6 ), SEPTET( 7 ), OCTET( 8 ),

DOUBLE\_QUARTET( 8 ), NONET( 9 ), DECTET( 10 );

private final int size;

private Ensemble( final int size ) {

this.size = size;

}

public int size( ) {

return size;

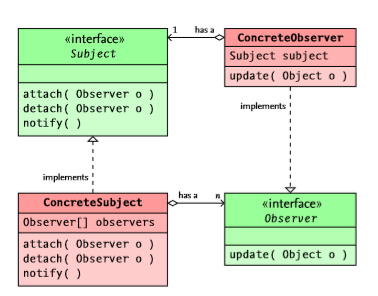
}

}

* Order can be changed.
* Constants can be removed.
* Constants can be added.

# The Observer Design Pattern

* The **observer pattern** is a commonly used design pattern.
* It deﬁnes a one-to-many object dependency.
* There is one Subject.
  + The Source of the News: A Newspaper.
* There are zero or more Observers.
  + Potential Readers.
* An Observer can be attached to the Subject.
  + Subscribe as Reader to the Newspaper.
* An Observer can be detached from the Subject.
  + Unsubscribe as Reader to the Newspaper.
* If the Subject’s state changes it updates all its Observers.
  + This is done by calling each Subject’s update( ) method.
  + Inform the Readers about News.



# Case Study

* We have a newspaper and readers of the newspaper.
* The readers can subscribe and unsubscribe.
* The newspaper informs the subscribers about new newsitems.

**Implementing the Interfaces**

public interface Subject {

// Subscribe to this newspaper.

public void attach( Observer subscriber );

// Unsubscribe from this newspaper.

public void detach( Observer subscriber );

// Notify this newspaper of a news event.

public void notify( String event );

}

public interface Observer {

// Inform this subscriber about a published event.

public void update( String event );

}

public class ConcreteObserver implements Observer {

// The name of the subscriber.

final String name;

public ConcreteObserver( final String name ) {

this.name = name;

}

// Inform this subscriber about a published event.

@Override

public void update( final String event ) {

System.out.println( name + " reading: " + event );

}

@Override

public String toString( ) {

return name;

}

}

public class ConcreteSubject implements Subject {

// The name of this newspaper.

private final String name;

// The subscribers of this newspaper.

private final ArrayList<Observer> subscribers;

public ConcreteSubject( final String name ) {

subscribers = new ArrayList<Observer>( );

this.name = name;

}

@Override public String toString( ) {

return name;

}

@Override // Subscribe a new customer.

public void attach( final Observer subscriber ) {

System.out.println( subscriber + " subscribes to " + this );

subscribers.add( subscriber );

}

@Override // Unsubscribe an existing customer.

public void detach( final Observer subscriber ) {

System.out.println( subscriber + " unsubscribes from " + this );

subscribers.remove( subscriber );

}

@Override // Inform this newspaper about hot news item.

public void notify( final String news ) {

// Inform all subscribers about the news item.

System.out.println( this + " got news item: " + news );

for( Observer subscriber : subscribers ) {

subscriber.update( news );

}

}

}

public class Main {

public static void main( String[] args ) {

final Subject eolas = new ConcreteSubject( "Eolas" );

final Subject examiner = new ConcreteSubject( "Examiner" );

final Observer john = new ConcreteObserver( "John" );

final Observer jane = new ConcreteObserver( "Jane" );

final Observer eoin = new ConcreteObserver( "Eoin" );

examiner.attach( john );

examiner.attach( eoin );

examiner.attach( jane );

eolas.attach( jane );

eolas.notify( "Assignment 1 to be handed back on Thursday." );

examiner.notify( "Eolas still reporting fake news." );

examiner.detach( jane );

examiner.notify( "No news today." );

}

}

# Windows

* Without a window you can’t write a gui application.
* In Java a window is represented as a JFrame object.
* The JFrame is where you put your window’s widgets in.
* Possible widgets are Buttons, Checkboxes, Sliders, Dialogue boxes, Text ﬁelds, And so on.
* The appearance of a JFrame may diﬀer from os to os.

Open that Window

* Create a JFrame.

JFrame frame = new JFrame( htitle stringi );

* Set the JFrame’s closing operation.

frame.setDefaultClosingOperation( JFrame.EXIT\_ON\_CLOSE );

* Make one or several widgets and add them to the JFrame.

JButton button = new JButton( "Click me" );

frame.getContentPane( ).add( button );

* Give the JFrame a size and make it visible.

frame.setSize( 300, 300 );

frame.setVisible( true );

# A Full Program

import javax.swing.\*;

public class DummyButton {

public static void main( String[] args ) {

JFrame frame = new JFrame( "Our second Button" )

JButton button = new JButton( "Click me" );

frame.getContentPane( ).add( button );

frame.setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE );

frame.setSize( 300, 300 );

frame.setVisible( true );

}

}

# Events

* It is quite obvious our button did nothing when we clicked it.
  + After all, we didn’t tell it what to do.
  + Let alone, how, and when.
* The JButton class knows when its buttons are clicked:
  + **Event**: Clicking the button generates a button event.
* To let the button do something when it’s clicked we need:
  + **Listener**: Listener to button events.
  + **Handler**: Listener instance method that is called for each event.

# Summary of Events

* The button event is activated when the button is clicked.
* The button event triggers the button event listener.
* The button event listener carries out the button event handler.

# That Sounds Familiar: The Observer Pattern

* The JButton is the Subject.
* Clicking the JButton is a user action.
* The JButton turns the user action into a button event object.
  + It may be thought of as a call to notify( event ).
* The button event is broadcast to all button event listeners.
* The Observers are the button event listeners.
* Each Observer implements its button event handler.
  + Each event handler is a dedicated update( ) method.
  + The call update( event ) sends the event to the listener.
    - The button sends the event by calling update( ).
    - By doing things in update( )’s body, the listener responds.

# Creating an Event Listener

* An event listener class implements an event listener interface.
  + Button event listeners implement the button listener interface,
  + Mouse event listeners implement the mouse listener interface,
  + And so on.
* Some interfaces have more than one notify( ) method.
* For buttons you usually only want to know when it’s clicked.
  + However, it is possible to distinguish between events pressing and releasing a button.
* The “click events” for JButtons are ActionEvent objects.
* So our listener must implement the ActionListener interface.
  + The method actionPerformed( ActionEvent event ) in the interface is equivalent to the Observer’s update( ) method.

## Example

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.Color;

public class SimpleGUI implements ActionListener {

private final JButton button;

private boolean alert;

public static void main( String[] args ) {

JFrame frame = <Create JFrame>

SimpleGUI gui = new SimpleGUI( );

<Remaining JFrame-related statements.>

}

public SimpleGUI( ) {

button = new JButton( "Click me" );

button.addActionListener( this );

}

@Override

public void actionPerformed( ActionEvent event ) {

button.setText( alert ? "Alarm" : "No panic" );

button.setBackground( alert ? Color.red : Color.green );

alert = !alert;

}

}

**A Button with a Counter**

import javax.swing.\*;

import java.awt.event.\*;

public class CountingButton implements ActionListener {

private int clicks;

private final JButton button;

public static void main( String[] args ) {

JFrame frame = <Create JFrame>

SimpleGUI gui = new SimpleGUI( );

<Remaining JFrame-related statements.>

}

public CountingButton( ) {

clicks = 0;

button = new JButton( "Click me" );

button.addActionListener( this );

}

@Override

public void actionPerformed( ActionEvent event ) {

String text = "# clicks = " + ++ clicks + ". Try again.";

button.setText( text );

}

}

# Nested Classes

* Classes deﬁned in other classes are called nested classes.
* There are two kinds of nested classes.
  + Static classes: these are called static (nested) classes.
  + Non-static classes: these are called inner classes.
* Both kinds of classes are part of the enclosing (deﬁning) class.
* The enclosing class is also referred to as the outer class.
* The diﬀerences between static and nested classes are subtle.

# Proper Inner Classes

* Deﬁned at top level of its outer class.
* An inner class instance depends on an instance of the outer class.
  + The inner instance can see its outer instance’s instance attributes.
  + Implicitly, the inner instance owns its outer instance’s reference.
  + Inner classes cannot have class attributes and class methods.
* You may create inner class instances in two kinds of methods.
  + **An instance method or constructor of the outer class.**
    - The new instance depends on the this of the method/constructor.
  + **An instance method or constructor of the inner class.**
    - The new instance depends on the instance that the current inner class instance depends on.

## Example: Inner Class

public class Outer {

private final int value;

public void outerMethod( ) {

Inner inner = new Inner( );

}

private class Inner {

private Inner( ) {

System.out.println( value );

}

…

}

}

## Inner Class: Second Example

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.Color;

public class InnerClassExample {

private final JButton button;

private boolean alert;

public static void main( String[] args ) {

final InnerClassExample gui = new InnerClassExample( );

gui.run( );

}

private InnerClassExample( ) {

button = new JButton( "click me" );

alert = false;

}

private void run( ) {

JFrame frame = new JFrame( "Inner Class Example" );

final JPanel panel = new JPanel( );

final Listener listener = new Listener( );

frame.getContentPane( ).add( panel );

panel.add( button );

frame.setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE );

frame.setSize( 300, 100 );

frame.setVisible( true );

}

private class Listener implements ActionListener {

private Listener( ) {

button.addActionListener( this );

}

@Override

public void actionPerformed( ActionEvent event ) {

button.setText( alert ? "Alarm" : "No panic" );

button.setBackground( alert ? Color.red : Color.green );

alert = !alert;

}

}

}

# Several Inner Classes

* You can have zero, one, or more inner classes.
* Very useful for gui applications.
  + **Outer class** Owns attributes that represent gui state.
  + **Inner class** Listens to the events.
    - Has access to the attributes.
    - Can modify them when an event occurs.

## Third Inner Class Example

public class EditorGUI {

private final ButtonGroup fontStyleGroup;

private final ButtonGroup sizeGroup;

…

public EditorGUI( ) { … }

private class FontGroupListener implements ActionListener { … }

private class SizeGroupListener implements ActionListener { … }

…

}

# Local Classes

* Java also lets you deﬁne classes in methods.
* These classes are called local classes.
  + A local class deﬁned in instance method is an inner class.
  + A local class deﬁned in a class method is a static class.
* Local classes may have names or not.
  + **With name**: These are called local (inner) classes.
  + **Without name**: These are called anonymous classes.
* Only use them when classes are really short.
  + With long classes, you usually can’t see the wood from the trees.

## Example: Local Inner Class

public class Outer {

public static void classMethod( ) {

private class LocalInner { … }

final LocalInner inner = new LocalInner( );

…

}

}

# Static Classes

* A static class is deﬁned at the top level of some other class.
* It has no access to outer class instance methods.
* It has no access to outer class instance attributes.

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.Color;

public class StaticDoubleListener {

public static void main( String[] args ) {

JFrame frame = new JFrame( "Two Listeners" );

final JButton firstButton = new JButton( "first" );

final JButton secondButton = new JButton( "second" );

final JPanel panel = new JPanel( );

final Listener first = new Listener( firstButton, secondButton );

final Listener second = new Listener( secondButton, firstButton );

frame.getContentPane( ).add( panel );

panel.add( firstButton );

panel.add( secondButton );

frame.setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE );

frame.setSize( 300, 100 );

frame.setVisible( true );

}

private static class Listener implements ActionListener {

private final JButton button;

private boolean alert;

private Listener( final JButton thisButton, final JButton thatButton ) {

button = thisButton;

thatButton.addActionListener( this );

}

@Override

public void actionPerformed( ActionEvent event ) {

button.setText( alert ? "Alarm" : "No panic" );

button.setBackground( alert ? Color.red : Color.green );

alert = !alert;

}

}

}

# Anonymous Classes

* An anonymous class is a class without name.
* It extends a single class or implements a single interface.
* It combines class deﬁnition & instance creation.
  + It cannot have an explicit constructor.
  + Its body should override all necessary methods.

## Example: Anonymous Class

public class Matrimony {

…

private static void unite( ) {

final Man john = new Man( ) {

@Override public void marry( final Woman wife ) { … }

};

final Woman mary = new Woman( ) {

@Override public void marry( final Man husband ) { … }

};

john.marry( mary );

}

}

# Interfaces can be “Nested” Too

public class Matrimony {

…

private interface Unitable { }

private interface Woman extends Unitable {

public void marry( final Man husband );

}

private interface Man extends Unitable {

public void marry( final Woman wife );

}

}

# Recursion

* Many concepts in computer science and mathematics are deﬁned or computed recursively, i.e. using recursion.
* The idea is to deﬁne a complicated concept in terms of itself.
* **Base Case**: Simple computation.
  + We don’t have to call the method itself.
* **Recursive Computation**: Complicated computation involving:
  + Simple computations.
  + Lower order computation(s).

Recursive Algorithm: Dictionary Search

**Dictionary Contains all Possible Words: One Word per Page**

* To search for the word givennpages do the following:
  + If there’s only one page (n = 1): We’ve found the word.
  + Otherwise (n > 1)
    - Find the page in the “middle.”
    - Read the word on the middle page.
    - If that word is our word: We’ve found the word.
    - If our word is smaller: search to the left.
    - Otherwise: search to the right.

# Pitfalls

* Recursive computations involve themselves.
* If we’re not careful we may get an inﬁnite chain of computations
* For example, we may be
  + Computing what’s on Box 1 with Box 2 on it, which involve
  + Computing what’s on Box 2 with Box 3 on it, which involves
  + Computing what’s on Box 3 with Box 4 on it, which involves ….
* Each recursive computation should eventually terminate.
* This only happens if they all reach some base case condition.
  + (The base conditions may be diﬀerent.)

Controlling the Size

**Guaranteeing Termination**

* Each computation should have a “size:”
  + A non-negative integer should do.
* The size should depend on one or several method parameters.
* Base-case computations have small ﬁxed sizes.
* Recursive sub-computations should get smaller and smaller.
* Using induction we can prove termination.

How does this Work?

**Dictionary Search Revisited**

* Let’s call the top computation C0.
* Le tC1 be the recursive computation of C0,
* Let C2 be the recursive computation of C1, and so on.
* Finally, let Si be the size of Ci.
  + By nature of the algorithm we have Si > Si+1.
* Let’s assume an inﬁnite chain of computations C0 ,C1 ,C2 ,... .
* Then we have an inﬁnite chain of integers S0 > S1 > S2 >··· .
* But this is impossible since Si ≥ 0, for all i.

# Computing Factorials

* Let nbe a positive integer.
* The factorial of n, denoted n!, is deﬁned as follows:
  + n! = 1×2×···×(n–1)×n.

**An Iterative Solution**

public static int factorial( int n ) {

int product = 1;

for (int i = 1; i <= n; i ++) {

product = product \* i;

}

return product;

}

**A Recursive Solution**

* **Base Case**: Clearly 1! = 1.
* **Recursion**: The recursion may be found by noticing that n! = (n–1)!×n.

| 1 if n = 1;

n! = {

| (n–1)!×n If n > 1.

public static int factorial( int n ) {

final int result;

if (n == 1) {

result = 1; // Base Case

} else {

result = factorial( n - 1 ) \* n; // Recursion

}

return result;

}

# The Fibonacci Sequence

* Fibonacci’s solution involves the series of numbers:
  + - 1,1,2,3,5,8,13,21,... .
  + Given the ﬁrst two we can compute the remaining numbers:

| 1 if n = 0;

Fn = { 1 if n = 1;

| Fn–1 +Fn–2 if n > 1.

public static int fibonacci( int n ) {

final int result;

if (n <= 1) { /\* Base Case \*/

result = 1;

} else { /\* Recursion \*/

result = fibonacci( n - 1 ) + fibonacci( n - 2 );

}

return result;

}

# The Towers of Hanoi

* We’re given a tower of 8 disks and three pegs: A,B, andC.
* Each disk has a hole in the centre.
* Initially, the disks are stacked in decreasing size on PegA.
* The objective is to transfer the stack to a diﬀerent peg, but
  + We’re only allowed to stack disks on pegs,
  + We’re only allowed to move one disk at a time, and
  + We can only stack a smaller disk on top of a larger disk.

## Intermediate State of the 3-Disk Version

* Here we **recursively** moved disks from C to B and were done!
* So, how did we arrive at the intermediate state?
* If we can solve this subproblem, we can solve the whole problem:
  + Start at initial state.
  + Solve the sub-problem to arrive at the intermediate state.
  + Use recursion to go from the intermediate to the target state.
* So, how did we get at the intermediate state?
  + We started with all disks stacked on Peg A.
  + We moved all disks except for the largest one from A to C.
    - But this is just the 2-disk version: move 2 disks from A to C.
  + We moved the largest disk to Peg B.

**Designing the Algorithm**

* **Base case**: If n = 1:
  + Move disknto target peg.
* **Recursion**: I fn > 1:
  + Move disks 1, …,n–1 to intermediate peg.
  + Move disk n to target peg.
  + Move disks 1, …,n–1 to target peg.

**Alternative solution**

* If n≥ 1 then
  + Move disks 1, …,n– 1 from source to intermediate peg.
  + Move disk n to target peg.
  + Move disks 1, …,n– 1 from intermediate to target peg.

/\*\*

\* @param n Number of disks.

\* @param source The source peg: should be 0, 1, or 2.

\* @param target The target peg: should be 0, 1, or 2.

\* <PAR> {@code source} and {@code target} should be different.</PAR>

\*/

private static void hanoi( final int n, final int source, final int target ) {

if (n >= 1) {

// Compute the number of the intermediate peg:

final int intermediate = 3 - source - target;

hanoi( n - 1, source, intermediate );

moveDisk( n, source, target );

hanoi( n - 1, intermediate, target );

}

}

public static void final hanoi( int n ) {

// move n disks from Peg 0 to Peg 1.

hanoi( n, 0, 1 );

}

private static void moveDisk( final int disk, final int source, final int target ) {

final String pegNames[] = { "A", "B", "C" };

System.out.println( "Move disk " + disk

+ " from " + pegNames[ source ]

+ " to " + pegNames[ target ] );

}

# Binary Search

* Binary search is an algorithm that:
  + Determines whether a given item is in a sorted list, and
  + If it is, returns the position of that element in the list.
* It works like the “dictionary search” algorithm.
* It repeatedly halves the number of elements.
  + It is a typical case of a divide and conquer algorithm.
  + Because of the halving it is sometimes called dichotomic.
* Requires (worst-case) time that is logarithmic in size of the input.

**The Basic Idea**

* Before studying the algorithm let’s deﬁne its main task.
* **Input**: The input of the algorithm consists of:
  + An item; and
  + A list of items sorted in non-decreasing order.
  + For simplicity the items in list are unique.
* **Output**: The output of the algorithm is an int.
  + The output depends on one of the following cases.
  + **Item is in list**: The index of item in the list.
  + **Item is not in list**: A negative number.
* For simplicity we’ll assume that all items are ints.
* Furthermore, we’ll assume that the list is presented as an array.

**The Algorithm**

binSearch( item, items, lo, hi )

* lo > hi: Return -1.
* lo <= hi:
  + Determine “the” middle index.
    - We implement this as mid = (lo + hi) / 2.
    - Unfortunately, this is not correct due to overﬂow.
    - You can ﬁx this by implementing it as
      * ‘mid = lo + (hi - lo) / 2’ or as
      * ‘mid = (hi + lo) >>> 1’.
  + Compare item and items[ mid ].
    - item == items[ mid ]:
      * Return mid.
    - item < items[ mid ]:
      * Return binSearch( item, items, lo, mid - 1 ).
    - item > items[ mid ]:
      * Return binSearch( item, items, mid + 1, hi ).

**Implementation in Java**

public static int binSearch( int item, int[] items ) {

return binSearch( item, items, 0, items.length - 1 );

}

private static int binSearch( int item, int[] items, int lo, int hi ) {

final int result;

if (lo > hi) {

result = - 1;

} else {

int mid = (lo + hi) / 2;

if (item == items[ mid ]) {

result = mid;

} else if (item < items[ mid ]) {

result = binSearch( item, items, lo, mid - 1 );

} else {

result = binSearch( item, items, mid + 1, hi );

}

}

return result;

}

# The Comparable Interface

* We’ve seen how to use binary search for ints.
* We should be able to generalise it for other comparable things.
* Implementing an interface is almost the same as extending a class.
  + If class B implements interface A, B behaves as A.
* A class implements the Comparable interface if it overrides int compareTo(Object that)
* Many classes implement the Comparable interface: Integer, Double, String, ….

private static int binSearch( Comparable item, Comparable[] items, int lo, int hi ) {

final int result;

if (lo > hi) {

result = - 1;

} else {

int mid = (lo + hi) / 2;

int outcomeOfComparison = item.compareTo( items[ mid ] );

if (outcomeOfComparison == 0) {

result = mid;

} else if (outcomeOfComparison < 0) {

result = binSearch( item, items, lo, mid - 1 );

} else {

result = binSearch( item, items, mid + 1, hi );

}

}

return result;

}

# What’s Going On?

* Many applications require collections of type-T objects.
* Program manipulates a collection, C, using objects of type T.
* To maximise re-use, C is implemented as collection of Object.
* Since Object is a superclass of T:
  + The compiler cannot assume C consists of type T objects.
  + Run-time errors may occur when taking things from C.
  + Run-time checks have to be added: performance degradation.
* It would be nicer if we could tell the compiler:
  + Trust me, all object in C are instances of (subclasses of) T.
    - This would help us detect/ﬁx errors at compile time.
    - This would avoid errors at runtime.
    - This would increase eﬃciency.

Solution: Generic Types

* A *generic* class depends on one or several type parameters:
  + A list with instances of the same class,
  + A binary tree with instances of the same class, …
* Instances of generic classes must have speciﬁc types:
  + A list of JButton objects,
  + A binary tree of Integer objects, …
* Generic types are used in combination with collections.
  + Lets you add objects to/remove objects from the collection.
* Java collection classes are all implemented as generic classes.
* If a generic class, G, is parameterised over a type, T.
  + You write G<T>.
* It guarantees that all objects “in” G have the same type: T.
  + Allows programmer to state what’s in the collection.
  + Allows the compiler to detect errors at compile time.
  + They eliminate the need for adding certain runtime checks.
  + They avoid runtime errors. Avoids code duplication.

## Example

**Don’t Try This at Home**

import java.util.\*;

public class CompileTimeError {

public static void main( String[] args ) {

ArrayList<Integer> nums;

nums = new ArrayList<Integer>( );

nums.add( "mistake" ); // compile-time error

nums.add( 1 );

Integer i = nums.get( 1 );

i = nums.get( 0 );

}

}

# The Comparable Interface

* An important interface is Comparable.
* A Comparable object can compare itself to other objects.
* To implement Comparable<T> you must override the method
  + int compareTo( T that ).
* compareTo( ) should implement deep comparison
* Comparison depends on result of compareTo( that ):
  + **Negative**: this is less signiﬁcant than that.
  + **Positive**: this is more signiﬁcant than that.
  + **Zero**: this and that are equally signiﬁcant.

## Example

public class Example implements Comparable<Example> {

int attribute;

@Override

public int compareTo( final Example that ) {

return ( this.attribute < that.attribute ? -1 :

this.attribute > that.attribute ? 1 : 0 );

}

}

## A Simple Generic Class

public class GenericClass<T> {

private T attribute;

public GenericClass( T value ) { attribute = value; }

public T getAttribute( ) { return attribute; }

public void setAttribute( T value ) { attribute = value; }

}

public class SimpleMain {

public static void main( String[] args ) {

GenericClass<Integer> gi;

GenericClass<Double> gd;

gi = new GenericClass<Integer>( 42 );

gd = new GenericClass<Double>( 3.14 );

final Integer oi = gi.getAttribute( );

final Double od = gd.getAttribute( );

System.out.println( oi + " " + od );

}

}

# Substitution Principle

When Java expects a value of a given type, you may also provide a value of a subtype of that type.

import java.util.ArrayList;

public class Example {

public static void main( String[] args ) {

final ArrayList<Number> nums;

nums = new ArrayList<Number>( );

nums.add( 42 );

nums.add( 3.14 );

System.out.println( nums );

}

}

**Subtyping**

ArrayList<Number> Does not Extend ArrayList<Integer>

Don’t Try This at Home

final ArrayList<Number> nums = new ArrayList<Number>( );

final ArrayList<Integer> ints;

ints = nums; // compile-time error.

nums.add( 3.14 );

// ints.toString == "[3.14]" ?

ArrayList<Integer> Does not Extend ArrayList<Number>

Don’t Try This at Home

final ArrayList<Number> nums;

ArrayList<Integer> ints = new ArrayList<Integer>( );

nums = ints; // compile-time error.

nums.add( 3.14 ); // nums is alias of ints.

// ints.toString == "[3.14]" ?

# Wildcards with extends

public interface Collection<T> {

…

public boolean addAll( Collection<? extends T> collection );

…

}

* dest.addAll( source ) adds all items from source to dest.
* Only makes sense if the things in source are subtypes of T.
* The spell ? in Collection<? extends T> is a wildcard.
  + It is any type (class/interface) extending T.
* So Collection<? extends T> collection guarantees that:
  + Any object in collection is-a T.
* Assume Sub is some subtype of some type Sup.
  + Then Collection<? extends Sup> is supertype of Collection<Sub>.

## Example

final ArrayList<Integer> ints = new ArrayList<Integer>( );

ArrayList<? extends Number> nums;

ints.add( 42 );

nums = ints; // Not allowed before.

Number num = nums.get( 0 ); // grand

Don’t Try This at Home

nums.add( 3.14 ); // compile-time error

Wildcards with super

* We just studied the spell ? extends T.
* It is for collections consisting of instances from subclasses of T.
  + The ? denotes any subclass of T.
  + It lets you safely get things from collections.
  + Collection<Sub> is a subtype of Collection<? extends Sup>. Java

final ArrayList<Integer> ints = new ArrayList<>( );

final ArrayList<? extends Number> nums = ints;

* Java also has a spell ? super T.
* It is for collections with instances of superclasses of T.
  + The ? denotes any superclass of T.
  + The spell ? super T lets you safely put things into collections.
  + Collection<? super Sub> is a supertype of Collection<Sup>. Java

final ArrayList<Number> nums = new ArrayList<>( );

final ArrayList<? super Integer> ints = nums;

# Using ? super T

ArrayList<? super Integer> ints = new ArrayList<Integer>( );

final ArrayList<Number> nums = new ArrayList<Number>( );

ints.add( 42 ); // grand

ints = nums; // Not allowed before.

nums.add( 1 ); // grand

Don’t Try This at Home

Number num = ints.get( 0 ); // compile-time error.

# The Get and Put Principle

* Use ? extends E for collections you get Es from.
* Use ? super E for collections you put Es into.
* Use E for collections you get Es from.
* Use E for collections you put Es into.

public static <T> void copy( ArrayList<? super T> destination,

ArrayList<? extends T> source ) { … }

## Examle

ArrayList<Integer> ints = new ArrayList<Integer>( );

ArrayList<? super Integer> nums;

ints.add( 42 ); // put

Integer i = ints.get( 0 ); // get

nums = ints;

nums.add( 1 ); // put

copy( nums, ints ); // put and get.

copy( ints, ints ); // put and get.

Don’t Try This at Home

copy( ints, nums ); // compile-time error.

Linked Lists

**Recursive Class Deﬁnition**

* Java already has an interface called List, so we implement our lists as MyList instances.
* Each MyList instance has an attribute called nodes,
  + This attribute represents what’s in the list.
  + If the list is empty, the value of nodes is null.
  + Otherwise, nodes represents a non-empty list.
  + Each Link instance represents a non-empty lists.
* Each Link instance has a *head* and a *tail* attribute.
* The *head* is the ﬁrst item in the list.
* The *tail* represents the remaining items in the list.

public class MyList { private Link nodes; … }

public class Link { private Link tail; private Comparable head; … }

# The Class MyList

public class MyList {

private Link nodes;

public MyList( ) { nodes = null; }

public void add( final Comparable item ) { nodes = new Link( item, nodes ); }

public Comparable getHead( ) { return nodes.getHead( ); }

public void print( ) { Link.print( nodes ); }

public void qsort( ) { nodes = Link.qsort( nodes ); }

}

# The Link Class

public class Link {

private Comparable head;

private Link tail;

public Link( final Comparable item, final Link list ) {

head = item;

tail = list;

}

public Comparable head getHead( ) { return head; }

public static void print( final Link list ) {

if (list != null) {

final String separator = list.tail == null ? "" : " ";

System.out.print( list.head + separator );

print( list.tail );

}

}

/\*public static void print( final Link list ) {

Link link = list;

while (link != null) {

final String separator = link.tail == null ? "" : " ";

System.out.print( link.head + separator );

link = link.tail;

}

}\*/

/\*public static void print( final Link list ) {

Link link = list;

String separator = "";

while (link != null) {

System.out.print( separator + link.head );

separator = " ";

link = link.tail;

}

}\*/

}

# The Method qsort( )

* We shall sort our list using the QuickSort algorithm.
* We do not have an array, but we use the same idea:
* **Base** **case**: If the list is empty then it is already sorted.
* **Recursion**: Otherwise:
  + The list is not empty.
  + Let head be the head of the list.
  + Partition the tail of the list into two lists leq and gt:
    - leq contains members less than or equal to head.
    - gt contains members greater than head.
  + Sort leq and gt.
  + Add head to the front of gt. Let gtExtended be this list.
  + Append leq and gtExtended.

## Implementing qsort( )

public static Link qsort( final Link list ) {

final Link result;

if (list == null) {

result = list;

} else {

final NodeList pivot = list.head;

final Partition partition = new Partition( pivot, list.tail );

final Link leqSorted = qsort( partition.leq );

final Link gtSorted = qsort( partition.gt );

pivot.tail = gtSorted;

result = append( leqSorted, pivot );

}

return result;

}

public static Link append( final Link start, final Link end ) {

final Link result;

if (start == null) {

result = end;

} else if (end == null) {

result = start;

} else {

result = start;

Link current = start;

while (current.tail != null) {

current = current.tail;

}

current.tail = end;

}

return result;

}

Implementing the Inner Class Partition

private static class Partition {

private Link leq; // members less than or equal to the pivot.

private Link gt; // members greater than the pivot.

private Partition( final Comparable pivot, final Link list ) {

Link leq = null;

Link gt = null;

Link link = list;

while (link != null) {

// initialise current link

final Link current = link;

// prepare link for next iteration

link = link.tail;

// add current link to destination partition

if (pivot.compareTo( current.head ) < 0) {

current.tail = gt;

gt = current;

} else {

current.tail = leq;

leq = current;

}

}

this.leq = leq;

this.gt = gt;

}

}

Don’t Try This at Home

public class MainSort {

public static void main( String[] args ) {

MyList list = new MyList( );

list.add( 1 );

list.add( "Bummer!" );

System.out.println( "Before sort." );

list.print( );

System.out.println( );

list.qsort( );

System.out.println( "After sort." );

list.print( );

System.out.println( );

}

}

Generic Linked Lists

**Design Options: Instance Method qsort( )**

public class MyList<T extends Comparable<T>> {

private Link<T> list;

public MyList( ) { list = null; }

public void add( final T item ) { list = new Link<T>( item, list ); }

public T getHead( ) { return list.getHead( ); }

public void print( ) { Link.print( list ); }

public void qsort( ) { list = Link.qsort( list ); }

}

**Design Options: Class Method qsort( )**

public class MyList<T> {

private Link<T> list;

public MyList( ) { list = null; }

public void add( final T item ) { list = new Link<T>( item, list ); }

public T getHead( ) { return list.getHead( ); }

public void print( ) { Link.print( list ); }

public static <S extends Comparable<S>> void qsort( final MyList<S> list ) {

list.list = Link.qsort( list.list );

}

}

**Using T for the Class Method is the Same as Using S**

public class MyList<T> {

private Link<T> list;

public MyList( ) { list = null; }

public void add( final T item ) { list = new Link<T>( item, list ); }

public T getHead( ) { return list.getHead( ); }

public void print( ) { Link.print( list ); }

public static <T extends Comparable<T>> void qsort( final MyList<T> list ) {

list.list = Link.qsort( list.list );

}

}

public class Link<T> {

private T head;

private Link<T> tail;

public Link( final T item, final Link<T> list ) {

head = item;

tail = list;

}

public T getHead( ) { return head; }

…

}

**Implementing print( )**

public static <S> void print( final Link<S> list ) {

Link<> visitor = list;

String separator = "";

while (visitor != null) {

System.out.print( separator + visitor.head );

separator = ",";

visitor = visitor.tail;

}

System.out.println( );

}

**Implementing qsort( )**

public static <S extends Comparable<S>> Link<S> qsort( final Link<S> list ) {

final Link<S> result;

if ((list == null) || (list.tail == null)) {

result = list;

} else {

final S pivot = list.head;

final Partition<S> p = new Partition<S>( pivot, list.tail );

final Link<S> leqSorted = qsort( p.leq );

final Link<S> gtSorted = qsort( p.gt );

pivot.tail = gtSorted;

result = append( leqSorted, pivot );

}

return result;

}

**Implementing append( )**

public static <S> Link<S> append( final Link<S> start, final Link<S> end ) {

final Link<S> result;

if (start == null) {

result = end;

} else {

result = start;

Link<S> visitor = start;

while (visitor.tail != null) {

visitor = visitor.tail;

}

visitor.tail = end;

}

return result;

}

**Implementing the Partition Class**

private static class Partition<S extends Comparable<S>> {

private final Link<S> leq;

private final Link<S> gt;

public Partition( final S pivot, final Link<S> list ) {

Link leq = null;

Link gt = null;

Link<S> visitor = list;

while (visitor != null) {

final Link<S> current = visitor;

visitor = visitor.tail;

if (pivot.compareTo( current.head ) >= 0) {

current.tail = leq;

leq = current;

} else {

current.tail = gt;

gt = current;

}

}

this.leq = leq;

this.gt = gt;

}

}

# Iteration

Type[] things = <magic>;

for (Type thing : things) { <use thing>}

* Generalised for loops work for any kind of array.
* Also works for collection classes.

final ArrayList<Type> things = new ArrayList<Type>( );

for (Type thing : things) { <use thing>}

* But ArrayLists aren’t arrays.
* So how does this work?

# Iterable<E>

* The ArrayList class implements the Iterable interface.
* To implement the interface you only have to do one thing
  + Override Iterator iterator( ).

final ArrayList<String> strings = <magic>;

for (String str : strings) { // Use string. }

final ArrayList<String> strings = <magic>;

final Iterator<String> iterator = strings.iterator( );

while (iterator.hasNext( )) {

String string = iterator.next( );

// Use string.

}

* **boolean hasNext( )**: Returns true if there are more elements.
* **E next( )**: Returns the next element in the iteration.
* **void remove( )**: Removes last E returned by next( ).