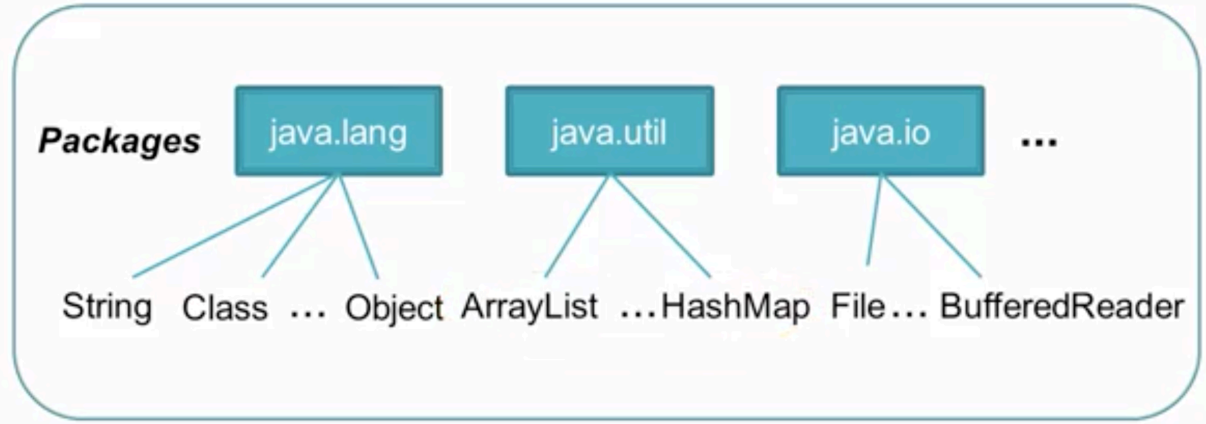
Java API

- Library of *well-tested* classes
- Java 8 ~ **4240** classes
- Developed by experts
- Used by *millions* of programmers
- Part of both JDK & JRE

Packages



Why Packages?

- Meaningful organization
- Name scoping

  java.util.Date **!=** java.sql.Date
- Security

Java API: Important Packages

- java.lang ~ Fundamental classes
- java.util ~ Data structures
- java.io ~ Reading & writing
- java.net ~ Networking
- java.sql ~ Databases

API Benefits

- Focus on writing new logic
- APIs *improve performance* over time
- Gain new functionality too

Accessing Classes

- Same package ~ *direct* access
- Different package
  - ***import***
  - Fully-qualified class name ~ *rare!*

import Statement

```
import java.util.ArrayList;

class FooClass {
  void foo() {
    ArrayList list = new ArrayList();
    …
  }
}
```

Importing Single vs Multiple Classes

- Import *single* class
  - Explicit import (or *Single-Type-Import*)
- Import *multiple* classes
  - Separate explicit import
  - * import (or *Import-On-Demand*)

* import

Imports *all* classes in a package

```
import java.util.*;
```

select * from <table_name>

Explicit import or * import?

- * import can ***break*** code

```
import java.util.*;                          import java.util.*;
import java.sql.*;      + java.sql.Date      import java.sql.*;
…                                            …
Date date; // from util                      Date date; // compiler error
```

- *Better clarity* with explicit import
- Explicit import seems to be preferred

Fully-qualified Class Name

- Alternative to *import*

```
java.util.ArrayList list = new java.util.ArrayList();
```

- Required if using *java.util.Date* & *java.sql.Date*

java.lang is imported by ***default***

Solution 1

Use only **one explicit** import

```
import java.util.Date;
import java.sql.*;

…

Date date; // from util
java.sql.Date date2;
```

Solution 2

Use **only fully-qualified** names

```
import java.util.*;
import java.sql.*;

…
java.util.Date date;
java.sql.Date date2;
```

Invalid Imports

```
import java.util.Date;
import java.sql.Date;
```

Any Side Affects in Using import?

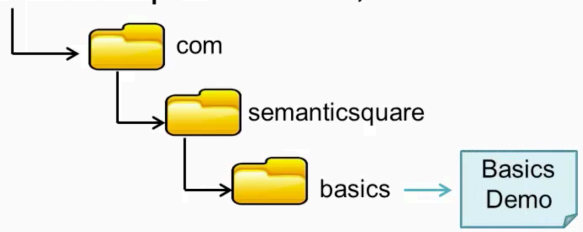
Nope!!

- *Does not make your class bigger!!*
- *Does not affect runtime performance*
- Saves from typing fully-qualified name ~ compiler does this

Set-up Matching Directory Structure

- basics;

- com.semanticsquare.basics;



package Statement

- **package** package-name

```
package com.semanticsquare.basics;
```

- Must be ***first*** statement above any imports

```
package com.semanticsquare.basics;
import java.util.ArrayList;
class BasicsDemo {
   …
}
```

- ✓ Ensure *matching directory structure* exists
- ✓ Use *package* statement

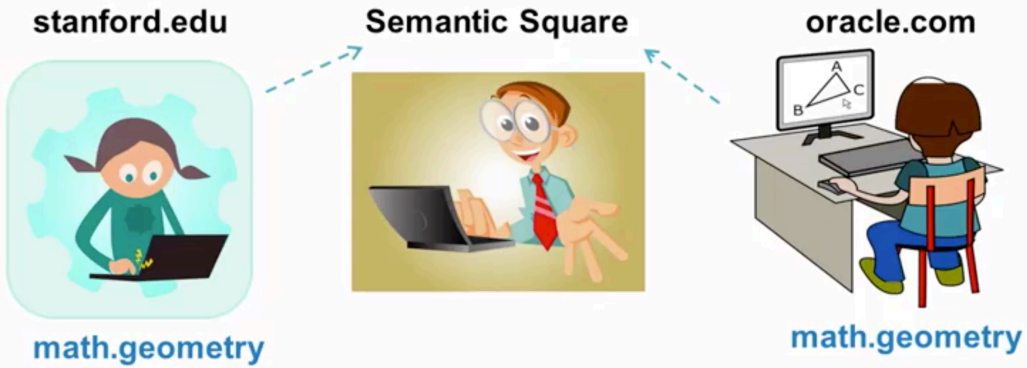
Package name is part of class name

java *BasicsDemo* ~ *will not work*
java *com.semanticsquare.basics.BasicsDemo*

Sub-packages

- *java.util* & *java.util.concurrent*
- java.util.* ~ imports *only* util
- java.util.*.* ~ **invalid**

Package Naming



package name conflict

Avoiding Package Name Conflicts

Use organization's *reverse internet domain name*

edu.stanford.math.geometry
com.oracle.math.geometry

Components Naming Conventions

- **Lowercase alphabets**, *rarely digits*
- Short ~ generally, *less than **8*** characters
- Meaningful abbreviations, e.g., util for utilities
- Acronyms are fine, e.g., awt for Abstract Window Toolkit
- Generally, single word
- Never start with *java* or *javax*

Strings

Object of class ***java.lang.String***

```
String s = new String(); // empty string
String s = new String("hello!");

char[] cArray = {'h', 'e', 'l', 'l', 'o', '!'};
String s = new String(cArray);

String s = "hello!"; // string literal    (recommended)
```

Strings

- String class uses **character array** to store text
- Java uses **UTF-16** for characters
- String is *sequence of unicode characters*
- String is *immutable*

String object ~ *immutable sequence of unicode characters*

String is Special

- String literal
- **+** operator

```
String s = “hello” + “ world!”; // “hello world!”
```

- String pool ~ *saves memory*

3rd Party String Utilities

- Apache Commons Lang ~ **StringUtils**
- **Guava’s** String Utility Classes

Common Operations

- Comparing
- Searching
- Examining individual characters
- Extracting substrings
- Case translation
- Replace
- Split

String Literal vs Using new

- String (via *string literal*)
  - Stored in *string pool* on heap
  - Literals with same content *share storage*
- String (via *new*)
  - Same as regular object
  - No storage sharing

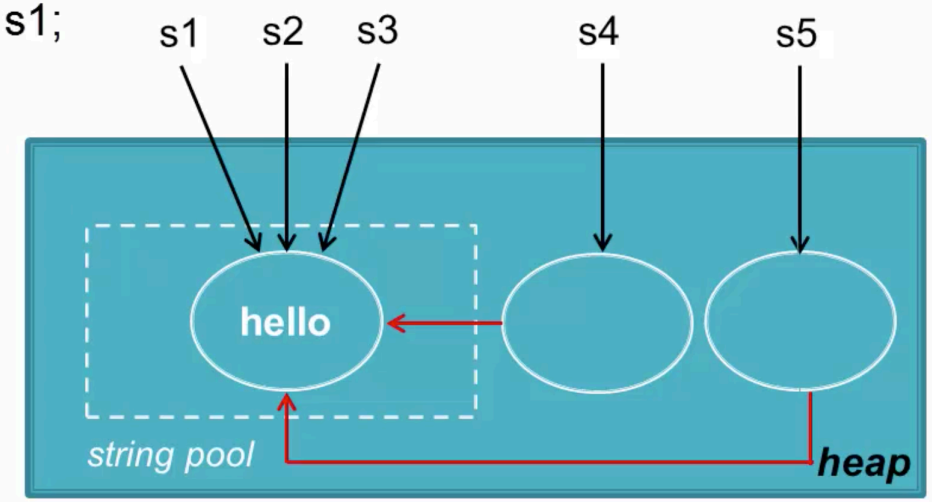
String *s1* = "hello";

String *s2* = "hello";

String *s3* = s1;

String *s4* = **new** String("hello");

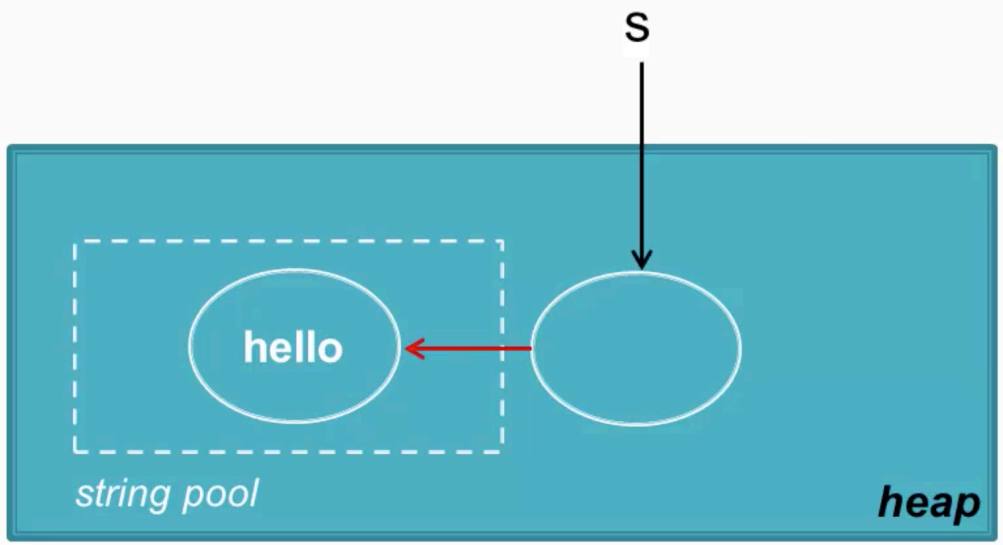
String *s5* = **new** String("hello");



s1 == s2? True

s4 == s5? False

String *s* = **new** String("hello");



String Pool

- Stores ***single*** copy of each string literal as **string object**
- Only ***one*** string pool
- Also called **string table**

String Interning by JVM

encountering a string literal for first time

✓ Create new **String** object with given literal

✓ Invoke ***intern***()

if (string *in* **string pool**)
  return *existing* reference
else
  add to string pool and return reference

Few Examples

- String *s* = “hel” + “lo”; ~ *interned* too
- String *s1* = “lo”;
  String *s2* = “hel” + s1; ~ *not interned*
  s2 = s2.***intern***(); ~ *explicit interning*

Is explicit interning useful?

Most likely not. Mostly for JVM

E.g., Natural language processing (NLP) ~ *needs benchmarking!!*

String Immutability

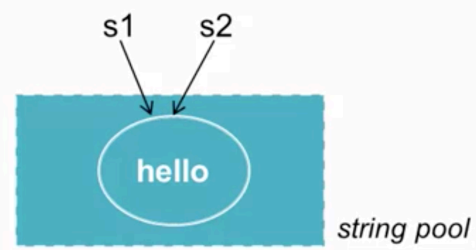
Value can *never* be changed

String s1 = new String(“abcd”); // or “abcd”
s1 = new String(“1234”); // above object is abandoned

Why Immutability?

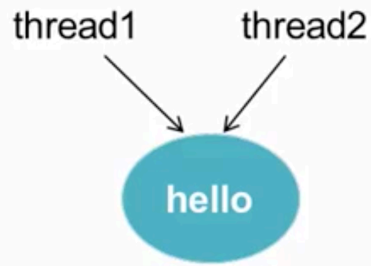
1. String interning

If mutable, *sharing* is not possible



Why Immutability?

2. Concurrency



Why Immutability?

3. Security

FileInputStream(String name)

String Concatenation

+ operator

String s = “hello” **+** “ world!”;

String s = “hello” + “ world!” + “125”; ~ *“hello world!125”*

String s = “hello” + “world!” + 125;

String s = “hello” + “world!” + 125 + 25.5 ~ *“hello world!12525.5”*

String s = 125 + 25.5 + “hello” + “world!” ~ “150.5hello world!”

String Concatenation

- StringBuilder
- StringBuffer

StringBuilder

- From Java 5
- Example

```
StringBuilder sb = new StringBuilder();
sb.append("hello");
sb.append(" world!");
String s = sb.append(" Good").append(" morning").toString();
```

- Other methods: *length*, *delete*, *insert*, *reverse*, *replace*
- Not synchronized

StringBuffer

- Obsolete. Use *StringBuilder!*
- Synchronized ~ slow
- API compatible with StringBuilder

Beware the performance of string concatenation

Item 51: + Operator

- Combining few strings is fine
- With each concatenation,
  - Contents of both strings are ***copied***
  - New **StringBuilder** is created and appended with both strings
  - Return string via toString()

Item 51: + Operator Example

Concatenating *a*, *b*, *c* in a loop

```
s += “a”; // copy of “” & a are made to generate a
s += “b”; // copy of a & b are made to generate ab
s += “c”; // copy of ab & c are made to generate abc
```

Also, StringBuilder is created for each concatenation

Time consuming ~ **$O(N^2)$**, Space consuming

Item 51: Use StringBuilder

- **O(N)**
- A/C one benchmark,
  - StringBuilder = ***300x*** times **+** operator
  - StringBuilder = ***2x*** times StringBuffer

Escape Sequence

- Character preceded by \
- To use ***special characters*** in strings & character literals

Escape Sequences

- \" ~ double quote *(not required in char literal)*
- \' ~ single quote *(not required in String literal)*
- \n ~ new line
- \t ~ tab
- \\ ~ backslash
- \r ~ carriage return
- \b ~ backspace
- \f ~ formfeed

char c = '\u0041';

Accessibility for Classes/Interfaces

- Inside package
- Inside & outside package ~ **public**

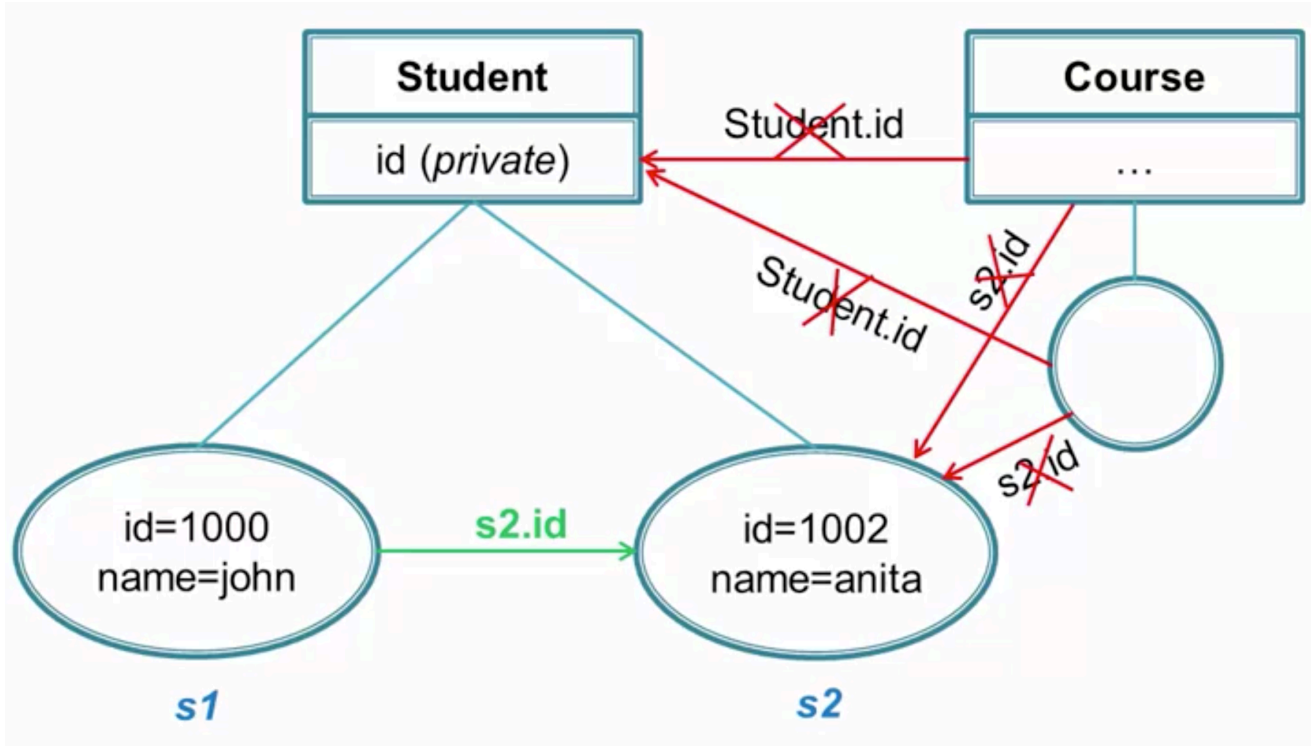
```
public class BasicsDemo {
    …
}
```

private Access Modifier

- **private** int id;
- **private** void go() { … }
- Private means *private to class and not object*

Accessibility for Class Members

- Inside class ~ **private**
- Inside package
- Inside package + *any* subclass ~ **protected**
- Inside & outside package ~ **public**


Student
id (private)
Course
...
Student.id
Student.id
s2.id
s2.id
id=1000
name=john
s2.id
id=1002
name=anita
s1
s2

Encapsulation

Data + Methods

```
public class Student {
    // variable declarations
    public int id;
    public String name;
    public String gender;

    // method definitions
    public boolean updateProfile(String newName) {
        name = newName;
        return true;
    }
}
```

Item 14: In public classes, use accessor methods, not public fields

Encapsulation alone does not lead to *good design*

```
public class Student {
    // variable declarations
    public int id;
    public String name;
    public String gender;

    // method definitions
    public boolean updateProfile(String newName) {
        name = newName;
        return true;
    }
}
```

tight coupling!!

Tight Coupling

- Can't enforce **invariant** (or range)
  - Can't restrict gender to **{**male, female, transgender**}**
- Can't change **data representation**
  - Can't change gender from *String* to *int*

```
public class Student {
   private String gender;
   public void setGender(String gender) {
      this.gender = gender;
   }
   public String getGender() {
      return gender;
   }
}
```

setter (mutator)

getter

```
public class Student {
   private String gender;
   public void setGender(String gender) {
      if (gender.equals("male") || gender.equals("female")
                                || gender.equals("transgender")) {
         this.gender = gender;
      } else {
         throw new IllegalArgumentException("Wrong gender passed!!");
      }
   }
   public String getGender() { … }
}
```

```
public class Student {
   private int iGender;
   private String gender;
   public void setGender(String gender) {
      if (gender == "male") { iGender = 1; }
      else if (gender == "female") { iGender = 2; }
      else if (gender == "transgender") { iGender = 3; }

      if (iGender == 0)
         throw new IllegalArgumentException("Wrong gender passed!!");
      this.gender = gender;
   }
   public String getGender() { … }
}
```

loose coupling!!

Loosely Coupled Systems

- *Develop*, *test*, *use*, and *optimize* in isolation
- Useful in multiple projects
  - *Decreases risk!!*

Item 13: Minimize the accessibility of classes and members

Item 13: Accessibility for Class Members

- Design ***minimal public API*** of your class
- Make all other members ***private***
- Make a member **default**, only if ***really*** needed
  - Frequent changes implies *reexamine your design!!*

Accessibility for Classes/Interfaces

- If possible, let it be **default**
- If only one class uses it, make it **private nested**

- ❖ Packages
- ❖ Strings
- ❖ Access Levels
- ❖ Information Hiding

Accessing Classes

- import
  - Explicit import preferred over * import
  - *Doesn't* make classes *bigger*
  - *Doesn't* affect runtime performance
  - Saves from typing fully-qualified class names
  - **java.lang** is imported by default

Creating Package

- ✓ Ensure *matching directory structure* exists
- ✓ Use *package* statement

Information Hiding → ***Loose Coupling***

Item 14: In public classes, use accessor methods, not public fields

Item 13: Minimize the accessibility of classes and members

Strings

- Object of class ***java.lang.String***
- String object is ***immutable***
- Uses *character array* to store text
- Java uses **UTF-16** for characters

String object ~ *immutable sequence of unicode characters*

String Pool

- Stores string literals as string objects
- Resides on heap
- Stores *single* copy of each string literal ~ *saves memory*
- String interning ~ process of building string pool

Question 1:**An import statement increases the size of the class. True or False?**

Question 2:**Let's consider the following code fragment:**

```
1.String s1 = "John";
2.String s2 = new String("John");
3.String s3 = s2.intern();
```

Now, would the expression (s1 == s2) return true or false?

Question 3:**With below statements, where does the String object that 's' references reside?**

```
1.final String s1 = "Good";
2.final String s2 = " Morning";
3.String s = s1 + s2 + "!";
```

Question 4:**If I have to construct a String by concatenating 10000 strings, which of the following approaches is most efficient? +Operator, StringBuilder, StringBuffer**

Question 5: **Which of the variables in the class XYZ below are accessible to code outside the package?**

```
package abc;
public class XYZ {
int b;
private int a;
public int d;
protected int c;
}
```

Options: b,c,d or c,d or d or b,d

Question 6:**Which of the class declarations is incorrect?**

```
Public class XYZ {....}
Private class XYZ{....}
Class XYZ{...}
```

Static Methods

Math Class

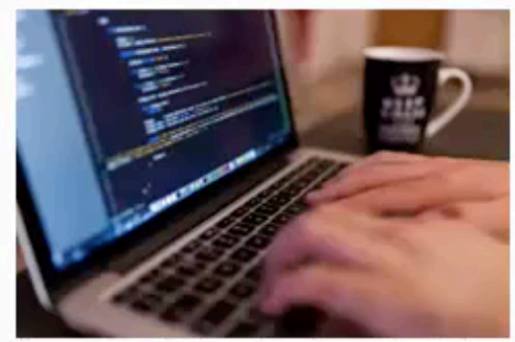
Static Variables

Initializer Blocks

Final Variables

Constant Variables

Coding Conventions



Method Types

- Instance methods
- Static methods *utility methods*

Private Constructor

Item 4: Enforce *noninstantiability* with a *private* constructor

java.lang.Math

- Math.*random*()
  - *double* between 0.0 & 1.0 (exclusive)
  - 0 <= **(int) (Math.random() * 5)** < 5
- Math.*abs*()
  - Absolute value
  - Math.abs(-240) → 240

- Math.*round*()
  - **Nearest** *long* or *int* based on argument
  - Math.round(24.8) → 25L, Math.round(24.25f) → 24 (*int*)
- Math.*ceil*()
  - Smallest double **>=** argument & equal to integer
  - Math.ceil(20.1) returns 21.0
- Math.*floor*()
  - Largest double **<=** argument & equal to integer
  - Math.floor(24.8) returns 24.0

- Math.*min*()/*max*()
- Math.*sqrt*()
  - Positive square root of a double
  - **NaN** if argument is NaN or -ve
  - NaN ➔ Not-a-Number (*undefined*), *e.g.*, 0.0/0.0
- cbrt, logarithmic & trignometric functions

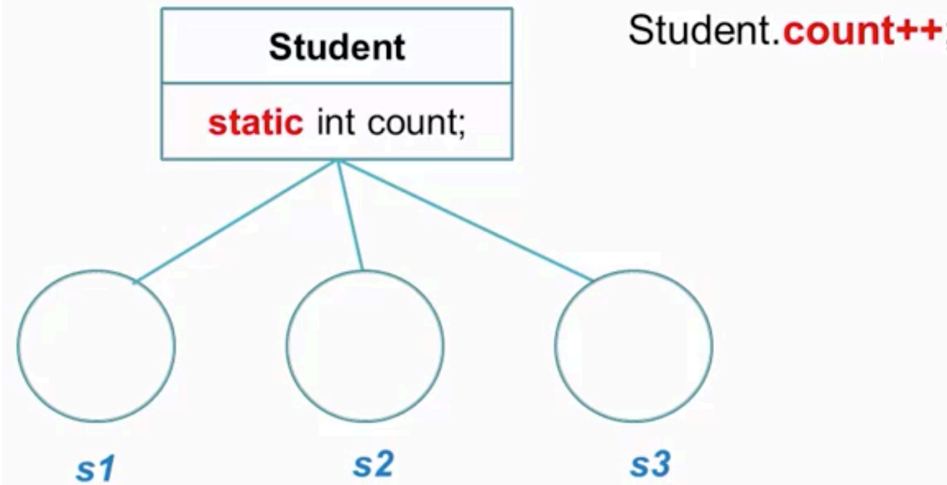
Static Methods

- Keyword **static** in declaration
- *main* method is static
- Class-level methods
  - No access to *state*, i.e., can't access instance variables/methods
- Can access *static variables*
- Can access *static methods*
- **Invocation**: *className.methodName()*, e.g., Math.*min()*
  - Saves *heap space*

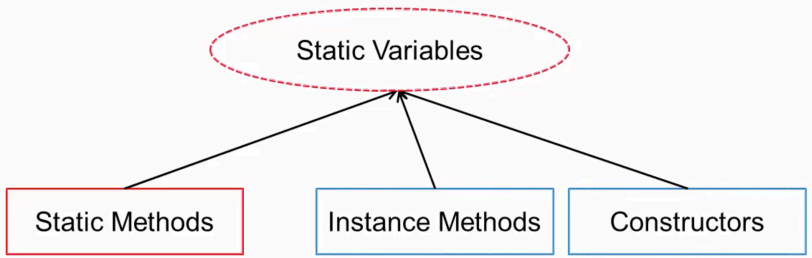
Static Variables

- Declared with keyword **static**
- Declared *directly within* class
- Gets *default* value
- **Accessibility**: *ClassName.variable*
- *Class variables*
  - One copy per class

shared across all objects



Accessibility within Class




```
class StaticExample {
   int instanceVar;
   static int staticVar;

   void instanceMethod() { // can access static & instance members
      instanceVar++;
      staticVar++;
      staticMethod();
   }
   static void staticMethod() { // can access only static members
      staticVar++;
      instanceVar++;      // compiler error
      instanceMethod(); // compiler error
      (new StaticExample()).instanceMethod(); // ok
   }
}
```

Static Initializer

- Initialization needs *multiple* lines
  - Populating a data structure
  - Initialization with error handling

Static Initializer Example 1

```
static HashMap map = new HashMap();

static {
   map.put("John", "111-222-3333");
   map.put("Anita", "222-333-4444");
}
```

Static Initializer Example 2

```
static Stuff stuff;

static {
   try {
     stuff = getStuff();
   } catch(Exception e) { … }
}
```

Example 2 – Private Static Method

```
static Stuff stuff = initializeStuff();

private static Stuff initializeStuff () {
   try {
     return getStuff();
   } catch(Exception e) { … }
   return null;
}
```

Static Initializer

- Multiple initializers ~ executed in order
- Cannot reference instance members

Instance Initializer

Initializes instance variables

```
{
   …
}
```

Constructors initialize state. *Why* instance initializer?

Share code between multiple constructors

Initializer *copied* into *beginning of every constructor*

Instance Initializer

- Multiple initializers ~ executed in order
- Can reference static members

Static initializer: Initialization needs multiple lines

Instance initializer: Constructors share code

final Keyword

This cannot be changed

```
public static final double PI = 3.14159265358979323846;
```

final Variable

- Implies **constant**
  - Primitive ~ value is constant
  - Reference variable ~ reference is constant, *not object content*
- Don't get *default* value
- Used with *instance*, *local*, or *static* variables

final Instance Variable

- Constant for ***life of the object***
- MUST be initialized in
  - Declaration
  - Constructor
  - Instance initializer
- **Demo** ~ let's extend *Student* example

final Local Variable

- Constant for ***life of the block***

```
public void register (final int courseId) {
    courseId++; // illegal
}
```

final Static Variable

- Constant irrespective of # *instances*

```
public static final int MAX_VALUE = 0x7fffffff;
```

- MUST be initialized in
  - Declaration
  - Static initializer
- Naming convention
  - All CAPS with underscore separating words
  - private static final int COPY_THRESHOLD = 10;

Constant Variables

- Compile-time constants

  public static **final** double *PI* = 3.14159265358979323846;
- Compiler optimization

  int x = Math.PI → int x = 3.14159265358979323846;

  Stored in ***.class*** file

Constant Variables

- ✓ final
- ✓ primitive or String
- ✓ Initialized in declaration statement
- ✓ Initialized with compile-time constant expression

Constant Variables – Valid Examples

- final int x = 23;
- final String x = “hello”;
- final int x = 23 + 5;
- final String x = “hello ” + “world!”;
- final int z = 5;

  final int x = 23 + z; // z is *hard-wired*

Constant Variables – Invalid Example

```
int z = 5;
final int x = 23 + z;

final int x = getVal();
```

```
public class Test {
   final int x;
   public Test () {
      x = 23;
   }
}
```

```
public class Test {
   static final int x;
   static {
      x = 23;
   }
}
```

Boxed Primitives

int ~ **Integer**
long ~ **Long**
byte ~ **Byte**
short ~ **Short**
float ~ **Float**
double ~ **Double**
boolean ~ **Boolean**
char ~ **Character**

Boxed Primitive Examples

```
Integer data = new Integer(25);
Boolean data = new Boolean(true);
Character data = new Character('c');
Double data = new Double(25.5);

Integer data = new Integer("25");
Integer data = new Integer("one"); // error at runtime
```

Uses of Boxed Primitives

- *String* to *primitive* conversions
  - int i = Integer.parseInt("25");
- Useful *public static* fields
  - MAX_VALUE, MIN_VALUE
- Utility methods
  - **Character**: *isLetter*, *isDigit*, *isLetterOrDigit*, *isLowerCase*, *isUpperCase*, *isWhitespace*
  - Integer.*toBinaryString*(int)
  - Double.*isNaN*(double)

Uses of Boxed Primitives

- Populating data structures
  - Can't add primitives

```
ArrayList list = new ArrayList();
list.add(25); // illegal before Java 5
list.add(new Integer(25));
```

- Generics
  - Parameterized types, e.g., ArrayList<Integer>

Common Methods

- Unwrap (Boxed to Primitive)
  - int i = (new Integer(25)).**intValue**();
- Parsing Strings
  - <orderId>25</orderId>
  - To *primitive*: int i = Integer.**parseInt**("25");
  - To *boxed*: Integer i = Integer.**valueOf**("25"); // *simple factory*
- To String
  - String s = Integer.**toString**(25);
- Wrap: Integer.***valueOf***(int) ~ *better performance!!*

Autoboxing

- Introduced in Java 5
- ***Automatically boxes*** *a primitive*

```
Integer boxed = 25;
Integer boxed = new Integer(25);
Auto-unboxing:
int j = boxed;
int j = boxed.intValue();
```

Method Invocation

```
ArrayList list = new ArrayList();
list.add(25);
```

autoboxing

```
list.add(new Integer(25));
```

Reduces Verbosity

Method Invocation

Autoboxing:

```
void go(Integer boxed) {}
go(25);
```

Auto-unboxing:

```
void go(int i) {}
go(new Integer(25));
```

Operations

```
Integer boxed = new Integer(25);
boxed++;
int i = 3 * boxed;
```

No autoboxing for ***arrays***

```
Integer[] items = new int[] {1,2}; // compiler error
```

Item 49: Prefer primitive types to boxed primitives

Time & Space Efficiency

```
void veryExpensive() {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum = sum + i;
    }
}
```

- auto-unboxes sum
- addition
- autoboxes again

Boxed Primitives are Classes!!

- Primitives have only *values*
- Boxed primitives have *identities* too
  - == & != → identity comparison
  - <, <=, >, >= → auto-unboxing
  - Mixed-type computations lead to confusing results

```
Integer i;
void unbelievable() {
    if (i == 0)
        System.out.println("weird!");
}
```

Typographical – packages

- **Lowercase alphabetic** characters, *rarely digits*
- Generally, ***short*** (< 8 chars) & ***single word***
- Meaningful abbreviations, e.g., util for utilities
- Acronyms are fine, e.g., awt for Abstract Window Toolkit
- Never start with *java* or *javax*

Use organization's *reverse internet domain name*

Typographical – Case

- Class
  - Capitalize first letter of each word, e.g., BufferedWriter
- Methods & Variables
  - Camel-case, e.g., *getArea*, studentCount
  - *static final* Variables
    - All CAPS with underscore separating words

static final int COPY_THRESHOLD = 10;

Typographical – Abbreviations

- Class, Methods, and Fields
  - Avoid abbreviations except commonly used like min/max
  - Acronyms are fine, e.g., *HttpUrl*
- Local Variables
  - Abbreviations & acronyms are fine
  - Meaningful individual characters are fine
    - x, y, z for co-ordinates
    - i for index

Grammatical – Methods

Performing ***action***

- Verb or verb phrase
- e.g., *append* or *calculateDistance*
- Use *descriptive* names
- Don't hesitate to use longer names

Grammatical – Classes

Singular noun or noun phrase

- e.g., User, BufferedWriter
- Simple & descriptive

Grammatical – Methods

boolean return type

- ***is*** followed by noun or noun phrase or adjective
  - e.g., noun → *isDigit*
  - e.g., adjective → *isEmpty*
  - e.g., *isActive* & *setActive*
- Sometimes, ***has*** is used, e.g., *hasLicense*

Grammatical – Methods

Non-boolean attribute of *object*

- Noun or noun phrase, e.g., *gender*, *hashCode*
- get***Attribute*** if there is setter, e.g., *getGender*

action - verb

returning boolean - ***is*** followed by noun/adj/boolean attribute name

Special Methods

- Object type conversions
  - *to**Type***, e.g., *toString*, *toArray*
- Static factory methods
  - *valueOf, of, getInstance, newInstance, getType, newType*

Grammatical – Fields

- Boolean
  - Usually, **adjectives**
  - *active* instead of *isActive*
- Non-Boolean
  - **nouns** or **noun phrases**
- *Singular* & *plural* nouns, e.g., item vs list
- Name objects of same class by ***purpose***
  - void sendMessage(User *sender*, User *receiver*)

Class Organization

- Variables ~ static followed by instance
- Static initializers
- Static nested classes
- Static methods
- Instance initializers
- Constructors
- Instance nested classes
- Methods

Class Size

- The Single Responsibility Principle
  - Helps create better *abstractions*
  - Helps in having fewer lines of code
- Less than *2000* lines

Methods

- Small & focused
  - Should do only **one** thing
- Refactor long methods
  - Software reuse
  - *Clean* and *readable* code

```
int search (int[] list, int key) {
  // Step 1: Sort
  …
  …
  …

  // Step 2: Binary search
  …
  …
  …
  …
}
```

→

```
int search (int[] list, int key) {
  sort(list);
  binarySearch(list, key);
}
```

Local Variables

Declare where *first* used

✓

```
void go() {
  // 50 lines of code

  double d = 3.14;
  foo(d);
}
```

✗

```
void go() {
  double d = 3.14;

  // 50 lines of code

  foo(d);
}
```

Braces

Beginning brace ~ End of line
Ending brace ~ Start of the statement

```
void go() {
  …
}
```

Indentation

Indent blocks by ***4*** spaces (or 1 tab)

```
public class XYZ {
    public void foo() {
        if () {
        }
    }
}
```

Wrapping Lines

Line length – ***80*** characters

- Break ***after*** **comma**, e.g., method calls & declarations
- Break ***before*** **operator**, e.g., if blocks, arithmetic ops.

Use **8-space** rule (or 2-tab)

Comments

- Code overview
- ***Non-obvious*** design decisions
- Frequent comments → poor code quality
- Use ***descriptive*** method & variable names

```
int search (int[] list, int key) {
    sort(list);
    binarySearch(list, key);
}
```

Comment Types

- Implementation comments
  - *// & /* … */*
  - Code documentation
  - Disable code, e.g., if (x < 7 */** && y > 3 **/*) { … }
- Documentation comments (or *javadoc* comments)
  - */** … **/*
  - API (*implementation free*)
  - *javadoc* ~ To extract to HTML files

Implementation Comments

Used *inside methods* or with *private fields*

- Block comments
- Single-line comments
- Trailing comments

Block Comments

- Describe a ***block*** of code
- 1 or more lines in /* … */
- Preceded by blank line
- Don't use //

Single-line Comments

- Short comments
- Preceded by blank line
- // or /* */

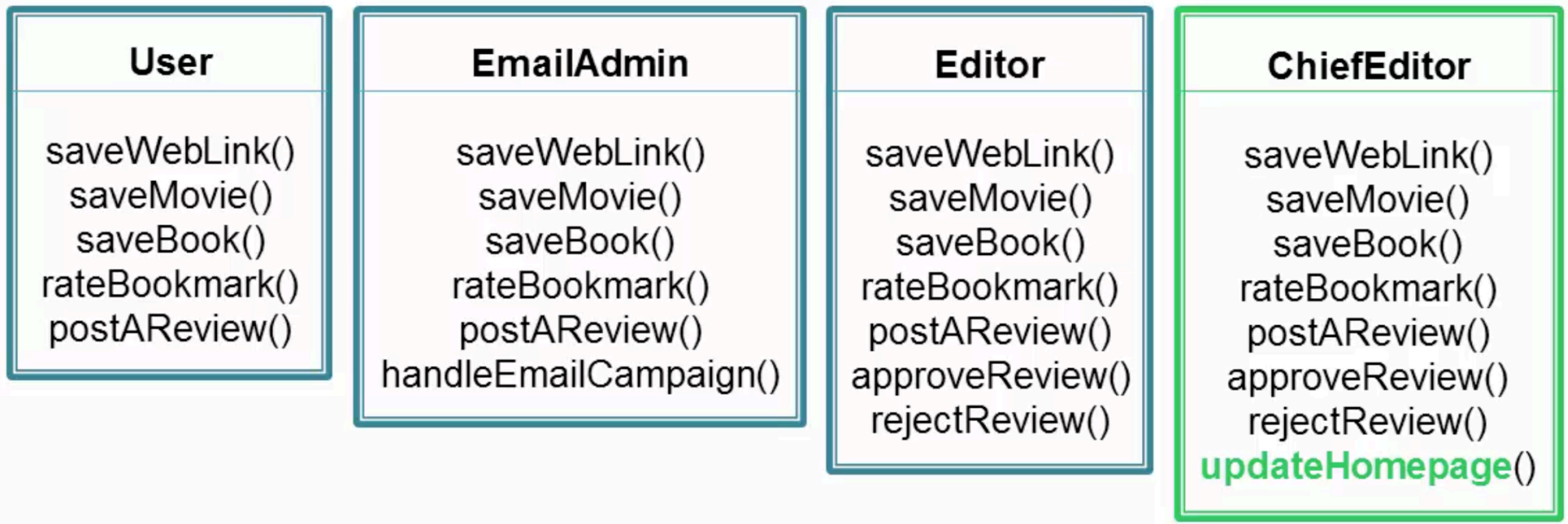
```
if (condition) {

    /* blah blah blah */
    ...
}
```

Trailing Comments

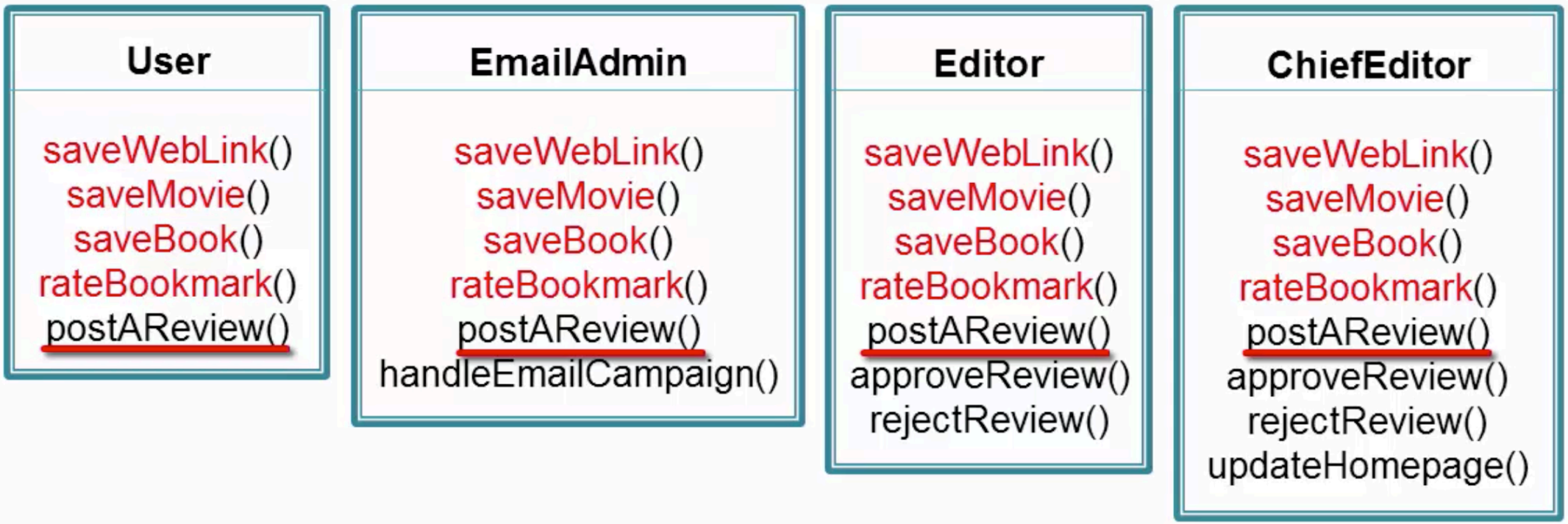
- Very short comments
- Appear on same line as code
- // or /* */

```
if (a == 1) {
    return true;          /* blah */
} else {
    return isPrime(a);    /* blah */
}
```

User Types


User
saveWebLink()
saveMovie()
saveBook()
rateBookmark()
postAReview()
EmailAdmin
saveWebLink()
saveMovie()
saveBook()
rateBookmark()
postAReview()
handleEmailCampaign()
Editor
saveWebLink()
saveMovie()
saveBook()
rateBookmark()
postAReview()
approveReview()
rejectReview()
ChiefEditor
saveWebLink()
saveMovie()
saveBook()
rateBookmark()
postAReview()
approveReview()
rejectReview()
updateHomepage()

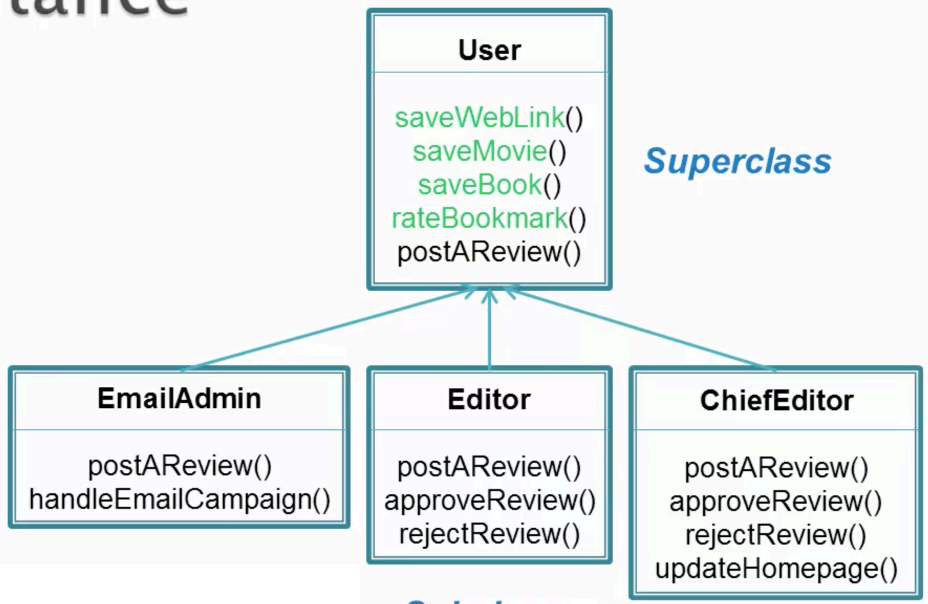
Motivation



duplicate code → *maintenance nightmare*

Solution: ***Inheritance***

Inheritance



Subclasses

Specialized versions of super classes

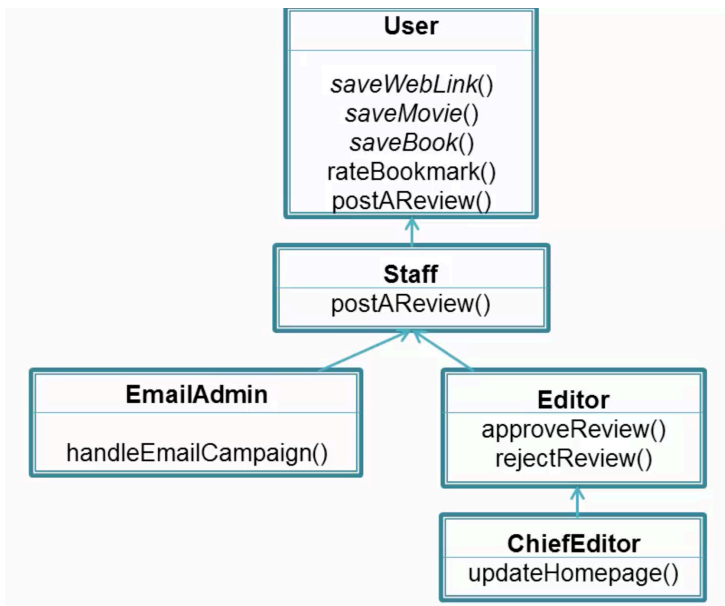
- ***Inherit*** members
- ***Add new*** members
- ***Override*** superclass methods

Subclass ← Superclass + Subclass capabilities

Terminology

Superclass → supertype or base class

Subclass → subtype or derived class



Extending a Class

- "extends"
  - class User {}
  - class Staff **extends** User {}
  - class EmailAdmin **extends** Staff {}
  - class Editor **extends** Staff {}
  - class ChiefEditor **extends** Editor {}

A class can extend from **only one** class

Inheritance Accessibility

- **private**
  - ***NOT*** inherited
- **default**
  - Inherited if from *family*
    - Only family can access inherited members
- **public**
  - Inherited
    - Anyone can access inherited members

Inheritance Accessibility

- **protected**
  - Inherited
    - Allow access to ***my*** family members

IS-A Test

Most ***fundamental test*** for inheritance

IS-A Test

- Staff IS-A User
- Editor IS-A Staff
- Editor IS-A User
- ChiefEditor IS-A User
- Surgeon IS-A Doctor
- Triangle IS-A Shape

HAS-A Test

- Bookmark IS-A Review ~ Nope
- Review IS-A Bookmark ~ Nope
- Bookmark **HAS-A** Review ~ *Yep*
- Bathroom IS-A Tub ~ Nope
- Tub IS-A Bathroom ~ Nope
- Bathroom **HAS-A** Tub ~ *Yep*

composition

Defining Contract

Class defines **contract**

"I have these kinds of methods …"

Defining Common Protocol

Supertype defines **common protocol**

"Myself & my subtypes have these kinds of methods …"

Polymorphism

Supertype = subtypes

*void updateProfile(**User** u) { … }*

✓ **flexible code**

✓ **clean code**

Reference type & actual object type can be different

```
User user = new Editor();
```

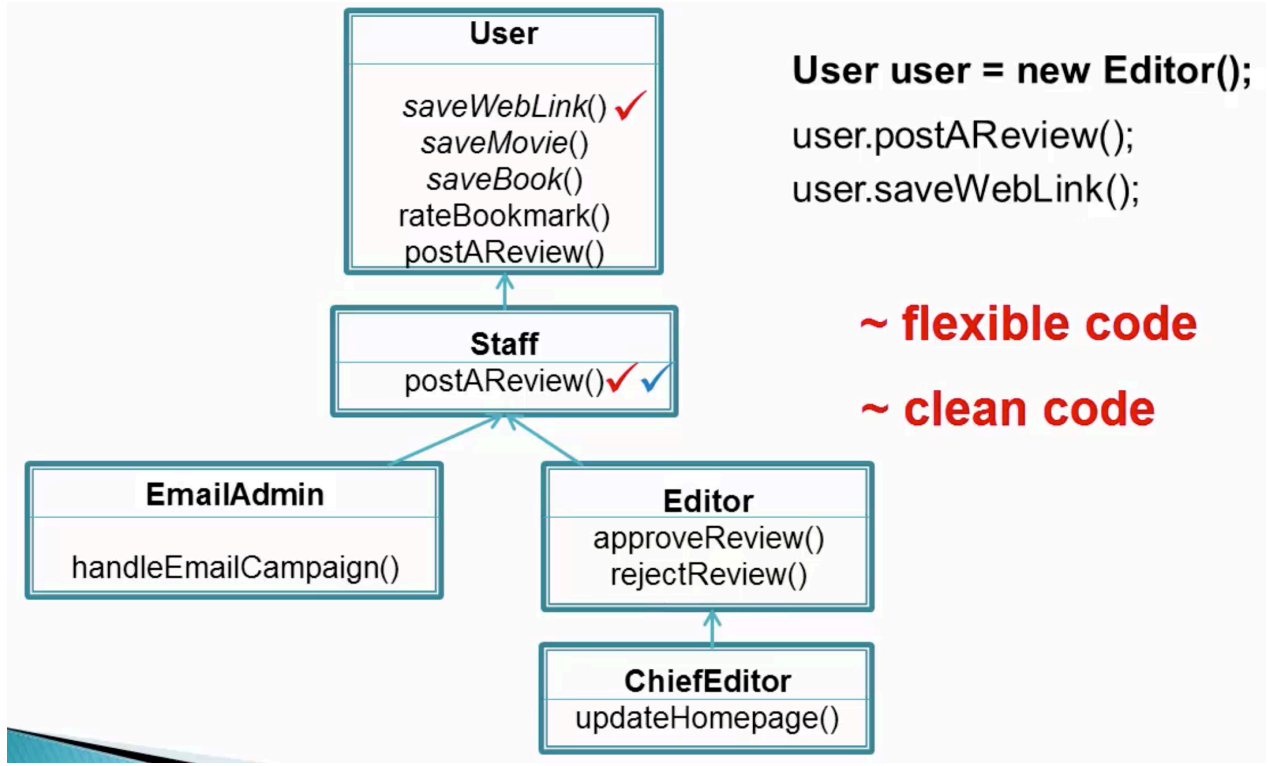
Reference Type ***Object Type***

Method Invocation

- Compiler uses **reference type** to decide ***whether*** a method can be invoked

```
User user = new Editor();
user.approveReview(); // compiler error
```

- JVM uses **object type** to decide ***which*** method is invoked
  - Invokes *most specific version*


User
saveWebLink()
saveMovie()
saveBook()
rateBookmark()
postAReview()
Staff
postAReview()
EmailAdmin
handleEmailCampaign()
Editor
approveReview()
rejectReview()
ChiefEditor
updateHomepage()
User user = new Editor();
user.postAReview();
user.saveWebLink();
~ flexible code
~ clean code