MIT OpenCourseWare
http://ocw.mit.edu

6.005 Elements of Software Construction
Fall 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

6.005 elements of software construction

Subclassing and Interfaces

Rob Miller
Fall 2008

© Robert Miller 2008

Today's Topics

More Java

- subclassing
- interfaces
- packages
- collections

© Robert Miller 2007

Getting Weather Information

Let's build on Page to get weather conditions

- Yahoo Weather feed: http://weather.yahooapis.com/forecastrss?p=**02139**
- Returns an XML page that looks like this:

```
...<title>Conditions for Cambridge, MA at 3:54 pm EDT</title>...
<yweather:condition text="Cloudy" code="26" temp="82" date="Sat, 06 Sep 2008 3:54 pm EDT" />...
```

Some string manipulation can extract the current weather condition:

```
Page p = new Page(new URL("http://weather...=02139");
String yweather = Match.between(p.getContent(),
                         "<yweather:condition", "/>");
String condition = Match.between(yweather,
                              "text=\"", "\"");
```

© Robert Miller 2007

Simple Pattern Matching

```
public class Match {
  /* Finds the first match to pattern in string, and
   * returns the rest of the string.  e.g.,
   *     after("Your Name: Ben", "Name: ") => "Ben"
   * Returns null if pattern never occurs in string. */
  public static String after(String string, String pattern)

  /* Finds the first match to pattern in string, and
   * returns the part of the string before it. e.g.,
   *     before("hello/there", "/") => "hello"
   * Returns null if pattern never occurs in string. */
  public static String before(String content, String pattern)

  /* Finds the first match to left in string, and
   * the first match to right after that, and
   * returns the substring between the two matches.
   * e.g. between("a <b>bold</b> word", "<b>", "</b>") => "bold"
   * Returns null if left...right never occurs in string.*/
  public static String between(String content, String left,
String right)
```

© Robert Miller 2007

Weather as a Subclass of Page

```
public class Weather extends Page {
  private String condition;
  private int temperature;

  public Weather(String zipcode) throws IOException {
    super("http://weather.yahooapis.com/forecastrss?p="
         + zipcode);
  }

  public String getCondition() {
    return condition;
  }

  public int getTemperature() {
    return temperature;
  }
  ...
}
```

extends means Weather is a subclass of Page

super () calls the superclass's constructor

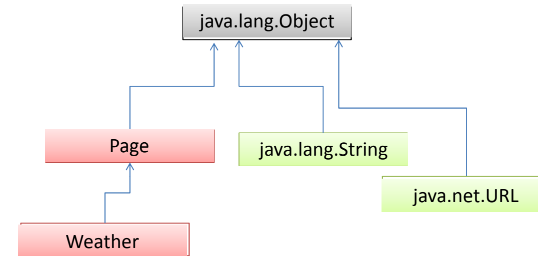
As a subclass of Page Weather **inherits** the methods and fields of Page: url, content, getURL(), getContent(), download()

© Robert Miller 2007

Class Hierarchy and java.lang.Object

Subclassing creates a hierarchy

- Every class implicitly extends java.lang.Object, the root of the hierarchy
- So every class inherits methods from Object, e.g. equals() and toString()



© Robert Miller 2007

Declared Type vs. Actual Type

- A variable has a **declared** type at compile-time
- The variable refers to an object with an **actual** type at runtime

Actual type is either the same or a subclass of declared type

static (compile time)	dynamic (run time)
Page p = new Page("http://...");	p → Page object
p.getContent();	
p = new Weather("02139");	p → Weather object
p.getContent();	
p.getTemperature();	

Weather has the same methods and fields that Page does (it inherits them), so it's safe for p to refer to a Weather object at runtime

But Java's static type checking won't allow you to use a method or field that isn't in the **declared type** of the variable (Page), even if it's in the actual type at runtime (Weather).

© Robert Miller 2007

More Access Control

- **protected** fields and methods can be used in this class or any of its subclasses

```
protected URL url;
protected String content;
```

  - This would allow Weather to access the fields it inherits from Page
  - But it can already access them through getURL() and getContent(), so we won't bother
- But let's move Page's downloading code into a protected method:

```
protected void download() throws IOException {
    Page p = getPageFromCache(url);
    if (p != null) {
        this.content = p.content;
    } else {
        this.content = Web.fetch(url);
        putPageInCache(this);
    }
}
```

© Robert Miller 2007

Overriding a Method

```
public class Weather extends Page {
  ...

  @Override
  protected void download() throws IOException {
    super.download();

    String yweather = Match.between(this.getContent(),
                          "<yweather:condition", "/>");
    this.condition = Match.between(content,
                          "text=\"", "\"");
    this.temperature = Integer.valueOf(Match.between(content,
                          "temp=\"", "\""));
  }
}
```

Overriding provides a new body for an inherited method

super.*method* (...) calls the superclass's implementation of *method*

Don't confuse **overloading** (two methods in the same class with the same name but different arguments) and **overriding** (a method implemented in both superclass and subclass, with the same name and same arguments)

© Robert Miller 2007

Method Selection

The actual type of the object is used to select the method body to call

static (compile time)	dynamic (run time)
Page p = new Weather("02139");	p → Weather object
p.download();	calls Weather's download() method, not Page's

The declared type of the variable is irrelevant to method selection

declared type is used by static type checking, to ensure that the method will exist at runtime

but actual type is used at runtime to select the method body to call

The location of the method call is also irrelevant to method selection

Page's constructor has a call to download() in it. Which version of download() will it call?

© Robert Miller 2007

Weather Data from the Cache

What's in Page's cache now?

- Even though the cache's declared type is Page[], at runtime it might contain a mix of Page and Weather objects
- This is OK, because all those objects behave like Pages
- But what if we want to take advantage of the cached Weather objects – i.e., reuse their temperature and condition values?
- First we have to make the cache accessible to Weather:

```
protected static Page getPageFromCache(URL url) { ... }
protected static void putPageInCache(Page page) { ... }
```

- Then we try this code in Weather.download():

```
Page cachedPage = getPageFromCache(this.url);
if (cachedPage != null) {
    this.temperature = cachedPage.temperature;
    ...
```

Type error -- the declared type, Page, doesn't have this field.

© Robert Miller 2007

Type Testing and Downcasting

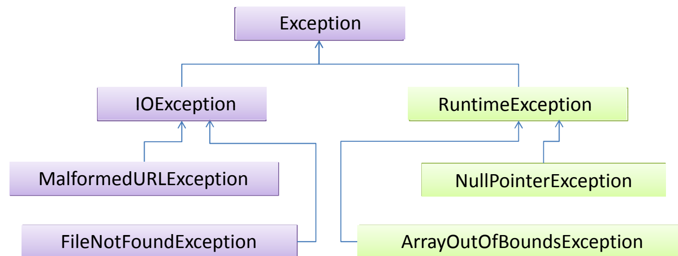
a* instanceof *B tests whether *a*'s actual type is B (or a subclass of B)

```
Page cachedPage = getPageFromCache(this.url);
if (cachedPage instanceof Weather) {
    // found a weather page in the cache
    Weather cachedWeather = (Weather) cachedPage;
    this.condition = cachedWeather.condition;
    this.temperature = cachedWeather.temperature;
} else {
    // extract condition and temperature from the content
    ...
}
```

Fix the type error by **downcasting** from Page to Weather, which asserts to Java that you *know* it will be a Weather object at runtime (because you just tested it with instanceof)

Don't confuse **casting** of object types (which merely changes the declared type at compile time and doesn't affect the runtime object at all) and **coercion** of primitive types (which actually produces a different runtime value; e.g. `(int)0.5` produces the value 0). They use the same syntax in Java!

Exception Class Hierarchy



Exceptions are normally **checked** at compile time – Java requires them to be either caught or declared.

But RuntimeExceptions are **unchecked** at compile time. You can catch or declare them, but Java doesn't require it.

© Robert Miller 2007

Multiple Catch Clauses

```
try {
  fetch("http://www.mit.edu/");
} catch (MalformedURLException e) {
  System.out.println("Bad URL: " + e);
} catch (IOException e) {
  System.out.println("IO problem: " + e);
}
```

a thrown exception is tested against each clause until finding the first one whose **declared type** is compatible with the exception object's **actual type**

What clause does "htt://www.mit.edu" run?

What clause does "http://www.mit.edu/foobar" run?

What if we switch the order of the clauses? (Tricky! What's the relationship between the MalformedURLException and IOException classes?)

© Robert Miller 2007

Page Cache as a Class

Recall how we cached web pages using static fields and methods in Page

```
private static Page[] cache;
private static int cachePointer;

protected static Page getPageFromCache(URL url) { ... }
protected static void putPageInCache(Page page) { ... }
```

It's sensible to wrap this behavior up into its own class

- Easier to understand: a "cache" abstraction with get and put operations
- Ready for change: we can easily change the data structure we use to implement it, even if we reuse the cache idea throughout the program
- Safe from bugs: users of the cache don't have access to its internal representation, only to the get and put operations

© Robert Miller 2007

Cache Implemented with an Array

```
public class ArrayCache {
    private Page[] array = new Page[100];
    private int pointer = 0;

    public Page get(URL url) {
        for (Page page : array) {
            if (page != null && page.getURL().equals(url)) {
                return page;
            }
        }
        return null;
    }
    public void put(Page page) {
        array[pointer] = page;
        ++pointer;
        if (pointer >= array.length) pointer = 0;
    }
}
```

representation

operations

This class needs no constructor, because all its fields are initialized in their declarations.

© Robert Miller 2007

Cache Interface

The essence of the cache are its get/put operations

- This essence can be captured by a Java **interface**, which contains only method declarations (not method bodies)

```
public interface Cache {
  public Page get(URL url);
  public void put(Page page);
}
```

interfaces can't have constructors or fields either – nothing but method declarations

- A class **implements** the Cache interface by declaring it and providing bodies for the two methods

```
public class ArrayCache implements Cache {
  ...
  public Page get(URL url) { ... }
  public void put(Page page) { ... }
}
```

- A caller can use Cache as an object type

Cache cache = new ArrayCache();
cache.get(new URL("http://www.mit.edu"));

since an interface has no constructor, you can't say new Cache() – you need to construct an object of a class that implements the interface

© Robert Miller 2007

Defining a Package Hierarchy

Let's organize our classes into packages

- web package: Page, Weather
- web.cache package: Cache, ArrayCache

Packages are folders in the filesystem

- The web.cache package corresponds to the path web/cache
- The folder contains a set of classes and interfaces, each in its own file, which all start with "package web.cache;" as their first line
- Eclipse handles this automatically when you make new packages and drag & drop classes into them

© Robert Miller 2007

Namespaces

Packages create separate namespaces for class names

- So you can use short names like Page and Cache without worrying that those classes already exist in another package

Namespaces are a vital pattern for organizing systems

- Easier to understand: names can be simpler and shorter
- Ready for change: reduces the scope of possible conflicts for new names
- Safe from bugs: names added in other namespaces don't affect this one

Widely used in Java and other systems

- Class creates a namespace for methods and fields
- Statement block {...} creates a namespace (**scope**) for local variables
- Domain name (@mit.edu) creates a namespace for user names

Namespaces are often hierarchical

- Sometimes inherit: e.g. local variables are inherited from enclosing scopes
- Sometimes don't: subpackages do not inherit classes from their parent package (and import web.* doesn't include web.cache.*)

© Robert Miller 2007

Cache Implemented With a List

```
public class ListCache implements Cache {
    private List<Page> list = new ArrayList<Page>();

    public Page get(URL url) {
        for (Page page : list) {
            if (page.getURL().equals(url)) {
                return page;
            }
        }
        return null;
    }

    public void put(Page page) {
        list.add(page);
    }
}
```

The List interface contains operations for a sequence data type: add(), get(), size(), etc.

The ArrayList class implements List using an array that grows as needed.

List and ArrayList are **generic** types, which means they take a type parameter. List<*Type*> represents a list of *Type* objects. Here we use List<Page> for a list of Page objects.

© Robert Miller 2007

Hailstone Sequences Done Right

For variable-length sequences, Lists are much better than Strings or arrays

```
/* Returns the hailstone sequence from n to 1 as a list.
 * e.g. if n=5, then returns the list (5,16,8,4,2,1).
 * Requires n >= 1. */
public static List<Integer> hailstoneSequence(int n) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(n);
    while (n != 1) {
        if (isEven(n)) n = n / 2; else n = 3 * n + 1;
        list.add(n);
    }
    return list;
}
```

Notice we used List<Integer> rather than List<int>. Generic types can only take object types as parameters, not primitive types. But every primitive type has a related object type (int/Integer, char/Character, long/Long, etc.), and Java automatically converts between them.

Cache Implemented With a Map

```
public class MapCache implements Cache {
    private Map<URL,Page> map = new HashMap<URL,Page>();;

    public Page get(URL url) {
        return map.get(url);
    }

    public void put(Page page) {
        map.put(page.getURL(), page);
    }
}
```

The Map interface represents a set of (**key, value** pairs and makes it easy to look up the value associated with a key. Here, the key is a URL, and the value is the page for that URL.

The HashMap class implements Map using a hash table (we'll have more to say about this in a later lecture).

Map is a powerful interface. It's ideal for a cache, but it has many other uses too. Learn it well and use it carefully!

© Robert Miller 2007

Anonymous Classes

An interface can be implemented by a nameless class

Starts like a constructor call…

```
Cache cache = new Cache() {
    private Page onlyPage;
    public Page get(URL url) {
        if (onlyPage != null && onlyPage.url.equals(url)) {
            return onlyPage;
        } else {
            return null;
        }
    }
    public void put(Page page) {
        onlyPage = page;
    }
};
```

…but includes a class body defining the interface's methods (plus other fields and methods if needed)

An anonymous class definition effectively creates a new class, but doesn't give it a name. Anonymous classes are frequently used in user interface programming, which is full of little interfaces to implement.

© Robert Miller 2007

Enumerations

enum defines a type with a small finite set of values

- Contrast with Page, which has an unbounded set of values (web pages!)
- Enums can have methods and fields too, like classes

```
public enum CompassPoint {
    NORTH,
    SOUTH,
    EAST,
    WEST;
}

public enum ANSWER {
    YES,
    NO,
    CANCEL;
}
```

enum values are referenced like public static constants – e.g. CompassPoint.NORTH

© Robert Miller 2007

Switch Statement

switch tests a value against a set of cases

- equivalent to a sequence of if-else clauses
- the value can be either an enum or integer type (int, char, etc.)

```
public static int degrees(CompassPoint point) {
    int result;
    switch (point) {
    case NORTH: result = 0; break;
    case EAST:  result = 90; break;
    case SOUTH: result = 180; break;
    case WEST:  result = 270; break;
    default:    throw new RuntimeException("invalid compass
point");
    }
    return result;
}
```

The cases of a switch must be terminated by **break**, or they will fall through to the next case!

default is like the else clause of a switch – it's good practice to always include one. Often it just throws an exception.

© Robert Miller 2007

Summary

Subclassing

- Inheritance of fields and methods
- Declared type vs. actual type
- Overriding and method selection
- Downcasting

Interfaces

- An interface captures the essence of a class: its method specifications

Packages

- Packages define separate namespaces for classes
- Namespaces are a useful organizing principle for large systems

Collections

- List<*Type*> is a list of *Type* objects
- Map<*Key,Value*> is a set of <Key,Value> pairs
- Generic types like List and Map take a type parameter <*Type*>

© Robert Miller 2007