

# CSE 1325: Object-Oriented Programming

## Lecture 08

# Packages, Interfaces, and JavaDoc

**Mr. George F. Rice**

**[george.rice@uta.edu](mailto:george.rice@uta.edu)**

**Office Hours:**

**Prof Rice 12:30 Tuesday and  
Thursday in ERB 336**

**For TAs [see this web page](#)**

A chicken crossed the road. It was poultry in motion.



# Overview: Interfaces and Packages

- Packages and Package-private
- Multiple Inheritance
- Interfaces
  - Writing
  - Implementing
  - Using as a Type
- Package Documentation
  - Tags
  - Examples



# Packages

- Packages are used to organize code in Java
  - Code within a package can have enhanced access to other code in the same package
  - The entire package can be “imported” elsewhere

```
package myPackage; ← The code in this file will be in “myPackage”
```

```
public class MyClass {  
    public static void main(String[] args) {  
        System.out.println("Running myPackage.MyClass.main()");  
    }  
}
```

Java NEVER needs C-like guards (#ifndef / #define / #endif)  
Java imports are *semantic* rather than *lexical*

**Package** – A grouping of related types providing access protection and namespace management





# Package corresponds to Directory

- A package is also a physical directory on disk
  - A class within a package has its source code file within a directory of the same name
- **Compile and run from the top-level directory!**
  - The source file is compiled as package/Source.java
  - The class file is executed as package.Source

```
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$ ls myPackage/
Permissions Size User   Date Modified Name
drwxrwxr-x   - ricegf 01-17 18:44  myPackage
.rw-rw-r--  498 ricegf 01-17 18:43  └─ MyClass.java
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$ javac myPackage/MyClass.java
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$ java myPackage.MyClass
Running myPackage.MyClass.main()
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$
```

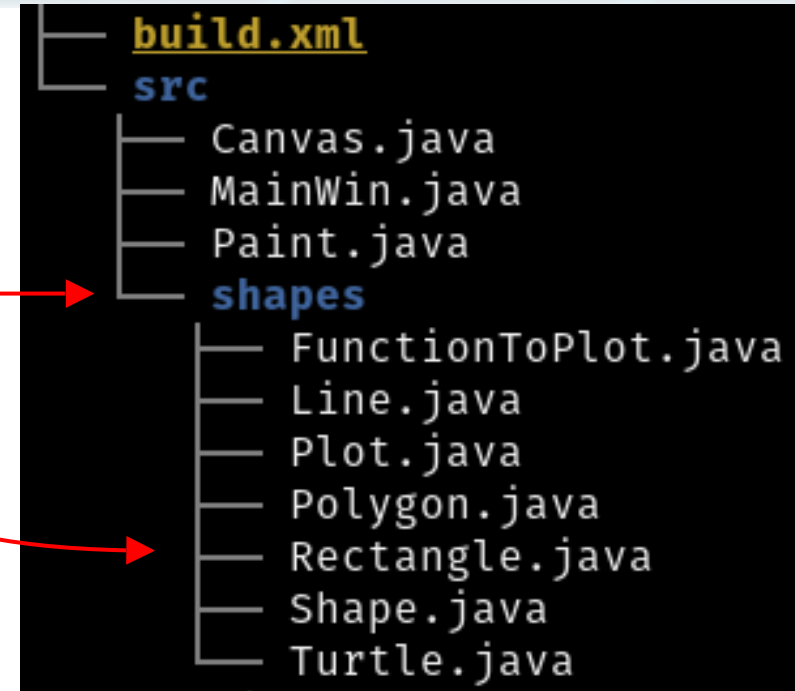
# A Point about Packages

Don't import classes with the same name!

```
import javax.swing.JPanel;
import java.awt.Dimension;
// import java.awt.Rectangle;
import java.awt.Graphics;
import shapes.Rectangle;

public class Canvas extends JPanel {
    private final static int border = 25;

    public Canvas() {
        // Continues
    }
}
```



Packages help avoid name conflicts for large projects and simplify reuse.

Be careful with imports. If you import two classes with the same name, the Java compiler will report an error. This is all too common with those who `import java.util.*` and the like!

The **fully-qualified** class name used for importing is *constant* – Import `shapes.Rectangle` with `import shapes.Rectangle;` from every class in every package in the system. **Imports are never relative!**

Classes in the same package never need to import each other, however.



# Package-Private Access Modifier

(Sometimes called simply Package)

- Package-private access permits *any* code within the *current package only* to access the class member
  - A **class** with no modifier is visible only within its package
  - A **method** with no modifier may be called by methods in any other class within its package
  - A **field** with no modifier is accessible and *may be modified* by any code within the current package

Package-private in Java is the *default* and thus requires/permits no modifier!

(Yes, this bothers me.)

```
package edu.uta.cse1325.grocery;

public class Fruit {
    private String name;
    protected Color color;
    Firmness firmness;
    public final static boolean hasSeeds = true;

    public Fruit(String name, Color color) {
        This.name = name;
        This.color = color;
        Firmness = Firmness.ETDM;
    }
}
```



# Thoughts on Package-Private

- Used only within *named* packages
  - Do NOT use in the default package / top directory of your program
  - Do NOT use just to allow your class and filename to differ
- Intended for collaboration across classes within a given package
  - The classes within the same package can thus communicate through a channel that is “hidden” from the other packages in the system
- In *practice*, package-private is not very common because units of functionality tend to span *multiple* packages
  - In other words, the package isn’t as useful a form of encapsulation for units of functionality greater than class as was expected
  - Some other languages such as C++ don’t even *support* package-private
  - Java 9 adds *module* support, which is much more flexible and useful. We won’t cover modules in this class due to limited time.

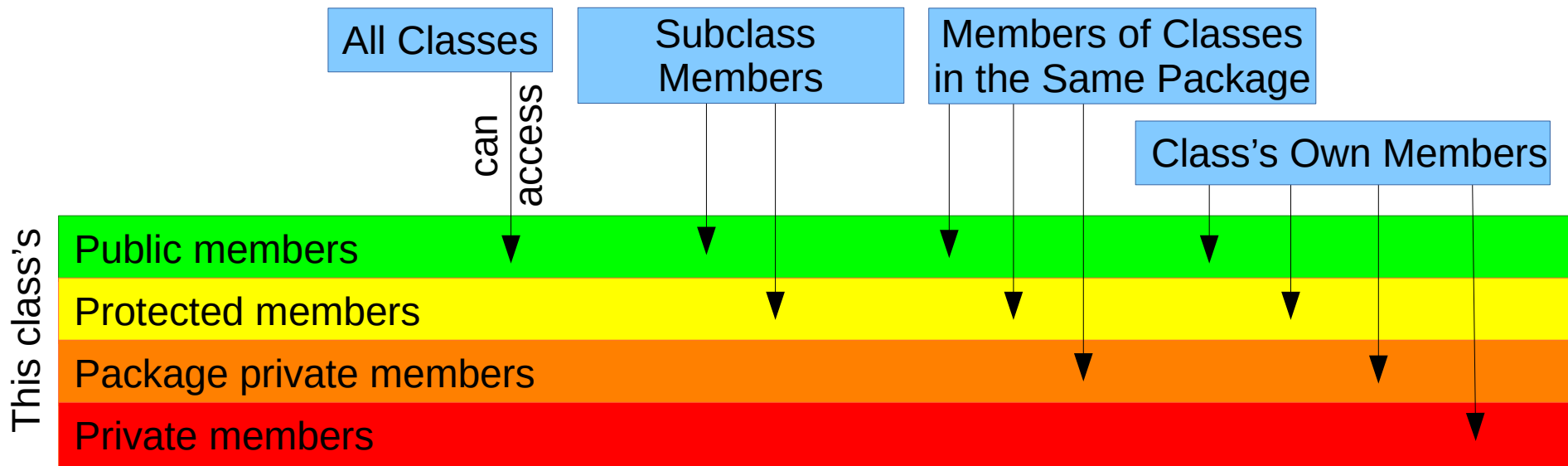


# Secrets of Protected

- Unlike other OO languages I know, Java allows other classes in the same package to access any *protected* field
  - **Private** fields are accessible only in the same class
  - **Protected** fields are accessible by any class within the same package AND any subclasses in *other* packages
  - **Package** fields are accessible by any class within the same package
  - **Public** fields are accessible by any class at all
- This is odd and inconsistent with other languages I know
  - I treat Java protected fields as accessible only within the class hierarchy for consistency
  - We will not deduct points for accessing protected fields, though



# Our Java Access Model is Complete at Last!



- A class or a class member (field, method, or class) can be
  - **Public** – **Anyone** can access this member
  - **Protected** – **Only class members and subclass members** (and members on the same package in Java) can access this member
  - **Package Private** (no modifier) – **Only class members within the same package** can access this member
  - **Private** – **Only class members** can access this member



# Thoughts on Packages

- Use packages where specified in the assignment
  - They are part of P04!
- Do NOT allow your editor to add packages to your homework assignments!
  - It may make assumptions that conflict with learning Java from scratch, e.g., that this entire semester is about writing a single, weirdly eclectic application
  - The resulting code won't run from the command line as specified in the requirements and is hard to grade
  - You will get low homework grades

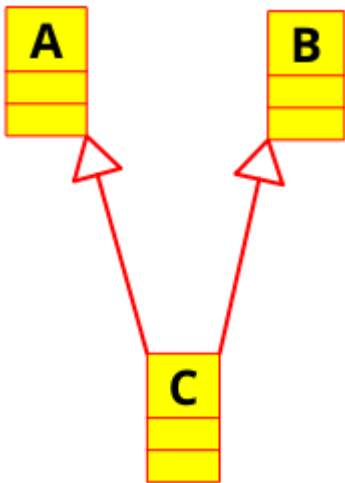






# Multiple Inheritance in UML and C++

- In UML, just use multiple arrows
- In C++, just list multiple base classes



```
#include<iostream>

class A {
    public: A() { std::cout << "A's constructor called" <<
std::endl;}
};

class B {
    public: B() {std::cout << "B's constructor called" <<
std::endl;}
};

class C: public A, public B {
    public: C() {std::cout << "C's constructor called" <<
std::endl;}
};

int main() {
    C c;
}
```

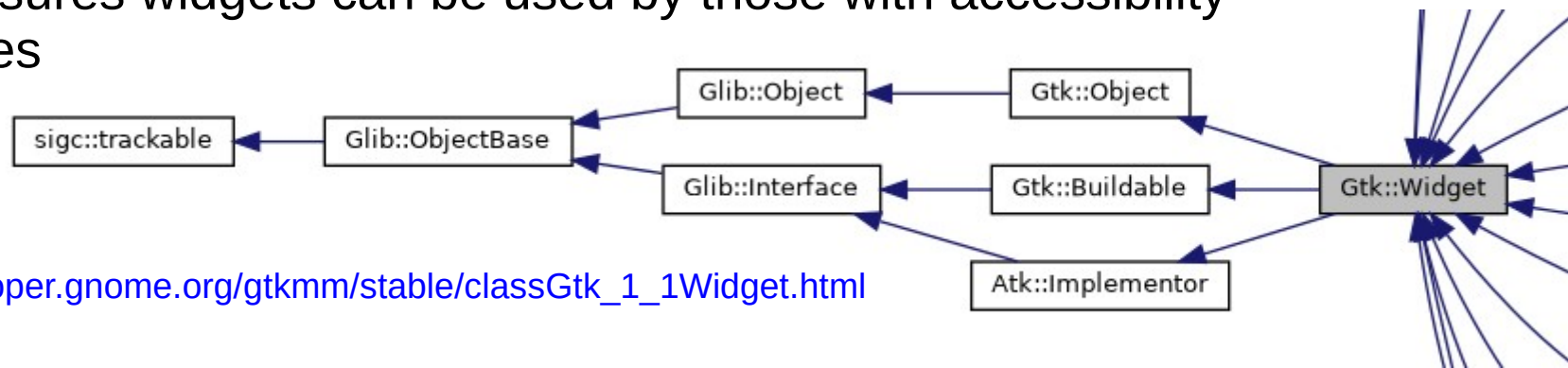
Constructors are called  
in the order listed.  
Destructors are called  
in the reverse order listed.

```
student@cse1325:/media/sf_dev/07$ make mi
g++ --std=c++17 -c mi.cpp
g++ --std=c++17 -o mi mi.o
student@cse1325:/media/sf_dev/07$ ./mi
A's constructor called
B's constructor called
C's constructor called
student@cse1325:/media/sf_dev/07$
```



# Is Multiple Inheritance Useful?

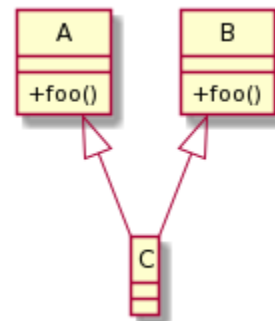
- Any problem that can be solved with multiple inheritance can also be solved with single inheritance
  - But multiple inheritance is *sometimes* helpful
  - One such case is graphical tool kits for building desktop applications
- A *widget* in gtkmm is a manipulable on-screen object, e.g., Button
  - Widget inherits from **Object**, which provides all members common to all gtkmm objects
  - Widget also inherits from **Buildable**, which support using a tool to design the layout of your windows
  - Widget *also* inherits from the **Accessibility Toolkit Implementor** (sic), which ensures widgets can be used by those with accessibility challenges



# Is Multiple Inheritance Problematic?

- Definitely! Sometimes...

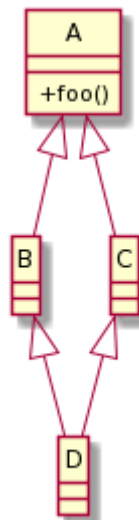
- In the hierarchy to the right, which `foo()` is `C.foo()` exactly?



- C++ compilers give an “ambiguous override” error
- Python calls the first method found in the method resolution order – although the method resolution order can be defined ambiguously if you’re clever

- In the hierarchy to the right, B and C both include an instance of A. So which `foo()` is `D.foo()`?

This is the “virtual inheritance” problem



Not on  
(this) exam!

Not on  
(this) exam!

- C++ allows you to define a “virtual A” pointed to by B and C – so B and C share the same A instance





# Multiple Inheritance in Java Interfaces

- Unlike C++ & Python, **Java has no *class* multiple inheritance**
  - This avoids the diamond problem and virtual inheritance issues
- In its place, Java provides first-class support **and multiple inheritance** for pure abstract classes called “interfaces”
  - Interface methods are *public abstract* by default
    - *Public static* methods with implementation, available to each implementer of the interface, are also permitted to provide “helpers”
    - *Public default* methods with implementation are permitted primarily to provide backward compatibility for updates that add new abstract methods
    - Fields are discouraged, but if included are *public static final* (i.e., const)
  - A class may *extend* 0 or 1 classes, but may *implement* any number of comma-separated interfaces
    - **A interface type may be used as a parameter or return type** – any object of any class that implements that interface is type-compatible


# Java Interface Definition

- A Java interface is a reference type, similar to a class, containing only method signatures, default methods, static methods, constants, and nested types
  - Like classes, interfaces may be extended (i.e., they support inheritance from other interfaces)
  - Unlike classes, interfaces may extend multiple other interfaces (i.e., they support *multiple* inheritance)
  - Unlike classes, interfaces are *never* instantiated, though they act as variable, parameter, or return types
    - You may then call only the methods specified by the interface





# Shushing our Barnyard Animals

- Let's say we want to sleep at night on the farm
  - We want to shush our animals so they are quiet
  - We want their glorious cacophony the next morning
- We'll define an interface Shushable
  - Any class that implements Shushable can be shushed by our main method
- We require 3 methods That's the  entire interface!
  - **shush()** will silence the animals
  - **unshush()** will allow them to speak again
  - **isShushed()** tells us if they are currently shushed or not

```
interface Shushable {  
    boolean isShushed();  
    void shush();  
    void unshush();  
}
```

# Rethinking Critter as Shushable

```
abstract class Critter implements Shushable {  
    public Critter(int frequency) {  
        this.frequency = frequency;  
        this.timer = 0;  
        shushed = false;  
    }  
    public void count() {if (++timer > frequency) timer = 0;}  
    public abstract void speak();  
    protected final String speakCmd = "espeak ";  
    protected void say(String s) { // omitted - see code at cse1325-prof  
    }  
    protected int frequency;  
    protected int timer;  
  
    // ///////////////////////////////////////////  
    // Implementing the Shushable interface  
    private boolean shushed;  
  
    public void shush() {shushed = true;}  
    public void unshush() {shushed = false;}  
    public boolean isShushed() {return shushed;}  
}
```

Our implementation is quite simple – a Boolean field that is set true or false or returned for our 3 methods.



# Shushing our Derived Critters

```
class Cow extends Critter {  
    public Cow(int frequency) {super(frequency);}  
    @Override  
    public void speak() {if (timer == 0  
                            && !isShushed()) System.out.println("Moo! Mooooo!"); }  
}
```

```
class Chicken extends Critter {  
    public Chicken(int frequency) {super(frequency);}  
    @Override  
    public void speak() {if (timer == 0  
                            && !isShushed()) System.out.println("Cluck! Cluck!"); }  
}
```

```
class Dog extends Critter {  
    public Dog(int frequency) {super(frequency);}  
    @Override  
    public void speak() {if (timer == 0  
                            && !isShushed()) System.out.println("Woof! Woof!"); }  
}
```

# Sleeping on a Farm with *Shushable*

```
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.Arrays;

public class Farm {
    public static void main(String[] args) {
        ArrayList<Critter> critters = new ArrayList<>(
            Arrays.asList(new Cow(13), new Dog(11), new Dog(9), new Cow(7),
                new Chicken(5), new Dog(3), new Chicken(2)));
        TimeUnit ms = TimeUnit.MILLISECONDS;

        System.out.println("W E L C O M E   T O   T H E   B A R N Y A R D !");
        for (int i=0; i<120; ++i) {
            for (Critter c: critters) { c.count(); c.speak(); }

            if(i%20 == 0) {
                System.out.println("==> Unshushing the barnyard animals! Earplugs in!");
                for (Critter c: critters) { c.unshush(); }
            } else if ((i+10)%20 == 0) {
                System.out.println("==> Shushing the barnyard animals... zzz...");
                for (Critter c: critters) { c.shush(); }
            }
            try {ms.sleep(200L + (long) (200L * Math.random()));}
            catch (InterruptedException e) { }
        }
    }
}
```

We'll alternately shush and unshush our critters every 10 counts.



# Sleeping on a Farm with *Shushable*

```
ricegf@pluto:~/dev/202008/08/java/shushable@ javac Critter.java
ricegf@pluto:~/dev/202008/08/java/shushable$ java Critter
WELCOME TO THE BARNYARD!
==> Unshushing the barnyard animals! Earplugs in!
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Moo! Mooooo!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
==> Shushing the barnyard animals... zzz...
==> Unshushing the barnyard animals! Earplugs in!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
```



# ArrayList of Shushables

- A parameter of Shushables, e.g., for shush() and unshush() methods, is permissible
  - NOT as ArrayList<Shushable>, which would be simpler
  - BUT as ArrayList<? extends Shushable>
- The methods shush() and unshush() can *only* use
  - > methods defined in Shushable,
  - > NOT methods defined in Critters,since the interface does NOT require Critter objects
- ArrayList of other types that implement Shushable could also be passed to these methods



# Shushable Methods

```
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
```

```
class Shushables {
```

These methods will accept an ArrayList of any type as long as that type implements Shushable (or a subclass thereof)

We can now shush and unshush an entire ArrayList with a single call!

```
private static void shush(ArrayList<? extends Shushable> list) {
    for(var e : list) e.shush();
}
```

```
private static void unshush(ArrayList<? extends Shushable> list) {
    for(var e : list) e.unshush();
}
```

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

```
    ArrayList<Critter> critters = new ArrayList<>(
        Arrays.asList(new Cow(13),    new Dog(11), new Dog(9),  new Cow(7),
                      new Chicken(5), new Dog(3),  new Chicken(2)));
```

```
    ... CODE OMITTED HERE
```

```
    System.out.println("W E L C O M E   T O   T H E   B A R N Y A R D !");
```

```
    for (int i=0; i<120; ++i) {
```

```
        for (Critter c: critters) { c.count(); c.speak(); }
```

```
        if(i%20 == 0) {
```

```
            System.out.println("==> Unshushing the barnyard animals! Earplugs in!");
            unshush(critters);
```

```
        } else if ((i+10)%20 == 0) {
```

```
            System.out.println("==> Shushing the barnyard animals... zzz...");
            shush(critters);
```

```
        }
```

```
    } try {ms.sleep(50L);} catch (InterruptedException e) { }
```

Critters is an ArrayList of Critter.  
Is it also an ArrayList of Shushable?

# Interface-type Parameters

- This simple (and largely non-nonsensical) additional shush method demonstrates using an interface type for a parameter
  - Any class that implements the interface may be passed
  - Only methods specified in the interface may be called in the method body

```
class Shushables {  
  
    private static void shush(Shushable e) {e.shush();}  
  
    private static void shush(ArrayList<? extends Shushable> list) {  
        for(var e : list) shush(e); // yes, e.shush() would work, too :-)  
    }  
  
    // Additional code omitted
```





# A Few Good Interfaces

(Though You May Write Your Own!)

- **List<E>** - An ordered object collection that permits duplicate values (implemented by `ArrayList`!)
- **Iterable** - Allows a container to participate in a for-each loop (also implemented by `ArrayList`!)
- **Comparable** – Requires the useful `compareTo` method
- **Serializable** – Enables streaming objects between systems
- **Readable** – Permits instances to be read via a `CharBuffer`
- **Appendable** – Permits chars to be appended to the object
- **Closeable** – Permits the object to be closed via `close()`
  - **AutoCloseable** – Permits a try-with-resources block to `close()` the object
- **Runnable** – Permits the object to be `run()` (as a `Thread`\*)
  - **Callable** – Permits the object to be run (as a `Thread`) and to return a result

\* But we'll use **Runnable** to implement Menu-Driven Interfaces, too!

# Example – Interface List Implemented by ArrayList

Module java.base

Package java.util

## Interface List<E>

Modifier and Type	Method	Description
void	<b>add</b> (int index, E element)	Inserts the specified element at the specified position in this list (optional operation).
boolean	<b>add</b> (E e)	Appends the specified element to the end of this list (optional operation).

Module java.base

Package java.util

## Class ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Modifier and Type	Method	Description
void	<b>add</b> (int index, E element)	Inserts the specified element at the specified position in this list.
boolean	<b>add</b> (E e)	Appends the specified element to the end of this list.



# Example – Interface List

## Also Implemented by LinkedList

Module java.base

Package java.util

### Interface List<E>

Modifier and Type	Method	Description
void	<b>add</b> (int index, E element)	Inserts the specified element at the specified position in this list (optional operation).
boolean	<b>add</b> (E e)	Appends the specified element to the end of this list (optional operation).

Module java.base

Package java.util

### Class LinkedList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

Modifier and Type	Method	Description
void	<b>add</b> (int index, E element)	Inserts the specified element at the specified position in this list.
boolean	<b>add</b> (E e)	Appends the specified element to the end of this list.

# Interchangeable Lists!

This is VERY idiomatic Java:

```
List<X> list = new ArrayList<>();
```

```
import java.util.List;           // The interface
import java.util.ArrayList;     // Implements List
import java.util.LinkedList;    // Implements List

public class UsingList {
    private static void fillList(List<Double> list) {
        for(int i=0; i<20; ++i)
            list.add(Math.random() * 100);
    }
    public static void main(String[] args) {
        List<Double> anyList;

        anyList = new ArrayList<>();
        fillList(anyList);
        System.out.println("As ArrayList: " + anyList);

        anyList = new LinkedList<>();
        fillList(anyList);
        System.out.println("\nAs LinkedList: " + anyList);
    }
}
```

## IMPORTANT

anyList is limited  
to methods defined  
in the List interface









# Javadoc

- Java includes the ability to generate API documentation
  - The documentation is built from the source code via the javac compiler
    - thus, the documentation *always* matches the source code
  - The documentation can be augmented with explanatory text embedded in the source code which *might* match the source code
  - The documentation is internally hyperlinked, and external hyperlinks can be added via code comments
  - Output can be HTML, XML, MIF, RTF, PDF, ... or you can define your own format
- **Javadoc** works with individual files or (more usefully) packages
- The standard documentation we've been using was generated by... well, you can guess :-D
- See the tutorial at <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html>



# Javadoc for a Class

- Javadoc is just a comment with a spare \*

```
/**  
 * Models a Complex number.  
 *  
 * @author          Professor George F. Rice  
 * @version         1.0  
 * @since           1.0  
 * @license.agreement Gnu General Public License 3.0  
 */  
public class Complex {
```

Package complex

## Class Complex

java.lang.Object  
complex.Complex

public class **Complex**  
extends java.lang.Object

Models a Complex number.

**Since:**

1.0

**Version:**

1.0

**Author:**

Professor George F. Rice

**Licensed under:**

Gnu General Public License 3.0

- The @ words are *tags* for javadoc to format
- Include Javadoc for all *public* classes and members
  - Javadoc will ignore non-public classes and members and their Javadoc markup

# Javadoc for a Class Member

```
/**
 * Creates a Complex instance.
 *
 * @param a          the real component of the Complex number
 * @param b          the imaginary component of the Complex number
 * @since 1.0
 */
public Complex(double a, double b) {
```

## Complex

```
public Complex(double a,
               double b)
```

Creates a Complex instance.

### Parameters:

a - the real component of the Complex number

b - the imaginary component of the Complex number

### Since:

1.0



# Javadoc “See Also”

```
/**
 * Creates a default Complex instance.
 *
 * @since      1.0
 * @see        #Complex(double,double)
 */
public Complex() {
    this(0,0);    // Chain to the first constructor
}
```

## Complex

```
public Complex()
```

Creates a default Complex instance.

### Since:

1.0

### See Also:

Complex(double,double)

To link to a member in another class (or even package)  
**@see Package.Class#member**

# Javadoc Common Tags

Tag & Parameter	Usage	Applies to
<b>@author</b> <i>John Smith</i>	Describes an author.	Class, Interface, Enum
<b>{@docRoot}</b>	Represents the relative path to the generated document's root directory from any generated page.	Class, Interface, Enum, Field, Method
<b>@version</b> <i>version</i>	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum
<b>@since</b> <i>since-text</i>	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method
<b>@see</b> <i>reference</i>	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method
<b>@param</b> <i>name description</i>	Describes a method parameter.	Method
<b>@return</b> <i>description</i>	Describes the return value.	Method
<b>@exception</b> <i>classname description</i> <b>@throws</b> <i>classname description</i>	Describes an exception that may be thrown from this method.	Method
<b>@deprecated</b> <i>description</i>	Describes an outdated method.	Class, Interface, Enum, Field, Method

Oddly missing is @license... go figure.

Table copied from Wikipedia.org, [https://en.wikipedia.org/wiki/Javadoc#Table\\_of\\_Javadoc\\_tags](https://en.wikipedia.org/wiki/Javadoc#Table_of_Javadoc_tags)  
Licensed under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#)





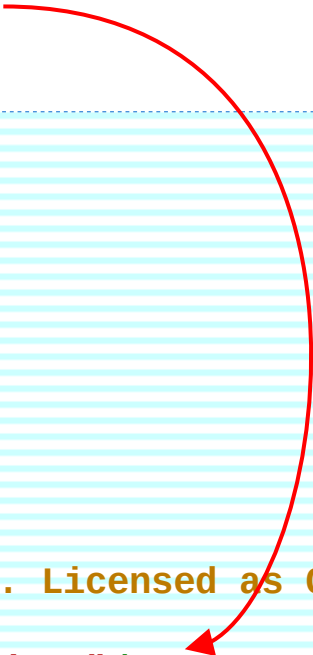
# Generating the Documentation

- OpenJDK includes the `javadoc` tool for manual builds
  - `-d dir` specifies the directory to receive the HTML
  - The arguments list the packages first, then additional files containing source code with JavaDoc mark up
- `javadoc -h` provides a complete list of options
- But `ant` is (almost always) much simpler!

# Example Ant Specification

- The javadoc tags specifies package names
  - Most of the tags are self-explanatory
  - <tag ...> permits creating custom tags

```
<javadoc packagenames="complex.*"
        sourcepath="."
        excludepackagenames="com.dummy.test.doc-files.*"
        defaultexcludes="yes"
        destdir="docs/api"
        author="true"
        version="true"
        use="true"
        windowtitle="Complex API">
  <doctitle><![CDATA[<h1>Complex</h1>]]></doctitle>
  <bottom><![CDATA[<i>Copyright &#169; 2023 Professor George F. Rice. Licensed as CC BY-SA
International 4.0</i>]]></bottom>
  <tag name="license.agreement" scope="all" description="Licensed under:"/>
</javadoc>
```





# Building with Ant

```
riceg@antares:~/dev/202108/08/code_from_slides/complex$ ant
Buildfile: /home/ricg/dev/202108/08/code_from_slides/complex/build.xml
Trying to override old definition of task javac
[javadoc] Generating Javadoc
[javadoc] Javadoc execution
[javadoc] Loading source files for package complex...
[javadoc] Constructing Javadoc information...
[javadoc] Standard Doclet version 14.0.2
[javadoc] Building tree for all the packages and classes...
[javadoc] Building index for all the packages and classes...
[javadoc] Building index for all classes...

build:
[javac] Compiling 1 source file

BUILD SUCCESSFUL
Total time: 1 second
riceg@antares:~/dev/202108/08/code_from_slides/complex$
```

Note: `ant clean` will also rebuild the documentation

# Package Documentation

**PACKAGE** CLASS USE TREE DEPRECATED INDEX HELP

SEARCH:

## Package complex

### Class Summary

Class	Description
<b>Complex</b>	Models a Complex number.

### Enum Summary

Enum	Description
<b>Complex.Form</b>	

**PACKAGE** CLASS USE TREE DEPRECATED INDEX HELP



# Class Documentation

PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD    DETAIL: FIELD | CONSTR | METHOD

SEARCH:

Package **complex**

## Class **Complex**

java.lang.Object  
complex.Complex

```
public class Complex
  extends java.lang.Object
```

Models a Complex number.

**Since:**

1.0

**Version:**

1.0

**Author:**

Professor George F. Rice

**Licensed under:**

Gnu General Public License 3.0

### ***Nested Class Summary***

#### **Nested Classes**

Modifier and Type	Class	Description
static class	<a href="#">Complex.Form</a>	

### ***Constructor Summary***

#### **Constructors**

Constructor	Description
<a href="#">Complex()</a>	Creates a default Complex instance.
<a href="#">Complex(double a, double b)</a>	Creates a Complex instance.

### ***Methods***

# Method Summary

## Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description	
<b>Complex</b>	<b>add</b> ( <b>Complex</b> rhs)	Adds this Complex instance to another, returning the result.	
boolean	<b>equals</b> (java.lang.Object o)	Returns true if the parameter is equal to the current Complex number.	
static <b>Complex.Form</b>	<b>getForm</b> ()	Returns the current form for all Complex I/O operations.	
double	<b>magnitude</b> ()	Calculates the magnitude of this complex number.	
static void	<b>setForm</b> ( <b>Complex.Form</b> form)	Specifies the form for future Complex I/O operations for all instances.	
java.lang.String	<b>toString</b> ()	Returns the Cartesian or polar form of the Complex number, depending on the current form.	

### Methods inherited from class java.lang.Object

clone, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait



# Method Details

## **Method Details**

### **magnitude**

```
public double magnitude()
```

Calculates the magnitude of this complex number.

**Since:**

1.0

### **add**

```
public Complex add(Complex rhs)
```

Adds this Complex instance to another, returning the result.

**Parameters:**

rhs - the Complex number to be added to this instance

**Returns:**

the resulting Complex number

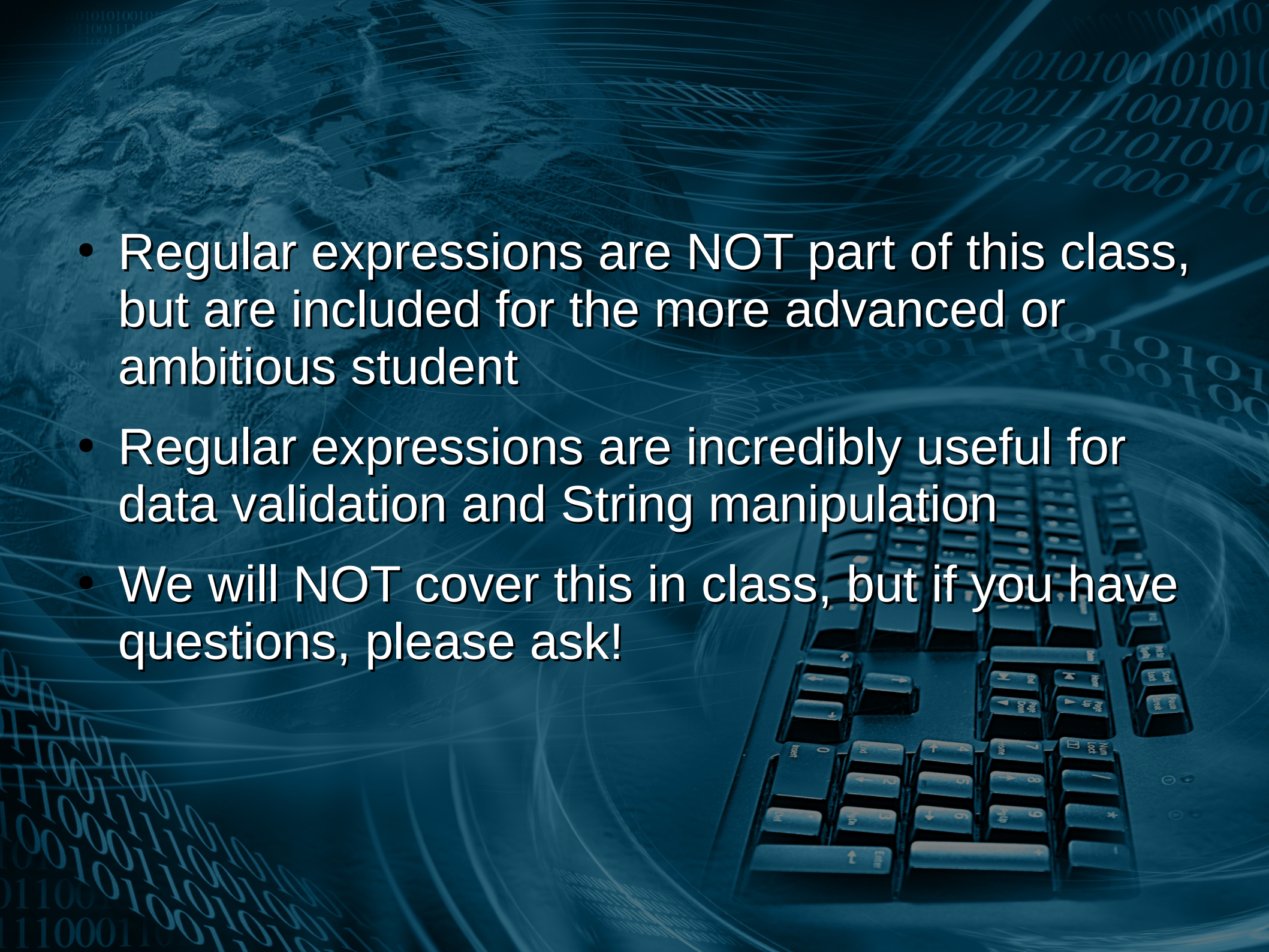
**Since:**

1.0



The End



- 
- The background of the slide is a dark blue composition. On the left, a portion of a globe is visible, showing continents. Overlaid on the globe and the rest of the background are various patterns of binary code (0s and 1s) in a lighter blue color. In the bottom right corner, there is a close-up, slightly angled view of a computer keyboard, also in shades of blue, with some keys like 'Enter', 'Page Up', and 'Page Down' clearly visible.
- Regular expressions are NOT part of this class, but are included for the more advanced or ambitious student
  - Regular expressions are incredibly useful for data validation and String manipulation
  - We will NOT cover this in class, but if you have questions, please ask!



# Optional More Regular Expressions (regex)

- A **regular expression** (regex) is a string that defines a search or search-and-replace pattern for other strings
- Two standards exist, POSIX and Perl – Java supports Perl
  - The Perl standard\* dates back to the 1950s, first proposed by American mathematician Stephen Cole Kleene, implemented in 1968 in the original Unix text editor QED, and enhanced in 1986 for Perl
  - The POSIX standard with more verbose but flexible syntax was adopted in 1992
- A regex is often an excellent choice for text manipulation, command or data parsing, or input validation
  - Regex are built into Perl and Javascript, and available as standard classes in C++, C#, Java, and Python. They are not part of ANSI C.
- The regex syntax is terse, cryptic, and borderline write-only, yet exceeding useful – learn it!

\* Actually, language implementations vary significantly – it's more of a *pseudo*-standard



# Simple regex Example

- Java provides “regex”, excellent for data validation
  - Lets you define the “look” of valid data
  - The match method will let you know if the data “looks” valid

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Ints {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Pattern pattern = Pattern.compile("^-?\\d+$");
        System.out.println("Enter some integers:");
        while(in.hasNextLine()) {
            Matcher matcher = pattern.matcher(in.nextLine());
            if(matcher.find())
                System.out.println("That's an int!");
            else
                System.out.println("### INVALID INPUT ###");
        }
    }
}
```

```
ricegf@antares:~/dev/202108/0
ricegf@antares:~/dev/202108/0
Enter some integers:
3.14
### INVALID INPUT ###
314
That's an int!
34785930438572034875
That's an int!
twelve
### INVALID INPUT ###
0x10
### INVALID INPUT ###
-365
That's an int!
19h
### INVALID INPUT ###
ricegf@antares:~/dev/202108/0
```

**Ints.java**

# (Very) Basic Regex Rules

- Most characters match themselves
- A special character matches one of a group
  - \s matches any whitespace except newline
  - \w matches a “word” character – A-Z, a-z, 0-9, or \_
  - \d matches a “digit” – 0-9
  - \S, \W, and \D match the opposite of their lowercase version
- Repeaters compactly express multiple characters
  - \d? represents either 0 or 1 digits
  - \d\* represents 0 or more digits
  - \d+ represents 1 or more digits
- Parentheses work as expected
  - (ha)+ represents ha, haha, hahahahaha, etc.

regex integer{"-?\d+"};

Matches itself

Matches “-” 0 or 1 times

Matches a digit

Matches the digit 1 or more times





# What Regex Would Validate These?

- “Professor Rice” or “Professor George Rice”
- A C++ name, e.g., “\_counter3”
- A negative integer, e.g., “-108”
- Adding two positive integers, e.g., “42+38”
- Exactly 3 words, e.g., “Go Mavs Go”
- One or more words, e.g., “Hello” or “Bye Bye Baby Bye Bye”

# What Regex Would Validate These?

- “Professor Rice” or “Professor George Rice”  
`Professor( George)? Rice`
- A C++ name, e.g., “\_counter3” `\w+`
- A negative integer, e.g., “-108” `-\d+`
- Adding two positive integers, e.g., “42+38”  
`\d+\+\d+`
- Exactly 3 words, e.g., “Go Mavs Go”  
`\w+\s\w+\s\w+`
- One or more words, e.g., “Hello” or “Bye Bye Baby Bye Bye”  
`\w+(\s\w+)*`





# Some Java Regex-Related Classes and Methods

- The Pattern class compiles the regex
  - Constructed by providing your pattern to the `compile` static method (NO constructor)
  - If the regular expression has syntax errors, compile throws a `PatternSyntaxException`
  - Once instanced, the Pattern may be used in multiple matches and searches
- The Matcher class represents matches of a Pattern to specific text
  - Obtained by providing the text to Pattern's `matcher` method
  - Matcher provides numerous methods for examining matches



# Matcher Methods

- Finding a match
  - `matches()` returns true if the pattern matches the *entire text*
  - `lookingAt()` returns true if the pattern matches *the start of the text*
  - `find()` returns true if the pattern matches *any substring of the text*
    - Starts after the previous match (if any), so repeated calls find all matches
    - Accepts an optional starting index parameter to start at specific character
- Information about a match
  - `groupCount()` returns the number of substrings matched
  - `group()` returns the substring currently matched
    - Accepts optional group number between 0 and `GroupCount()-1`
  - `start()` & `end()` return start and end index of current match



# A Simple Regex Parser

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.regex.PatternSyntaxException;

public class Regex {
    public static void main(String[] args) {
        String text = args[0];    String regex = args[1];
        System.out.println(text); // We'll mark matches on the string itself
        int pos = 0;
        try {
            Pattern pattern = Pattern.compile(regex); // Compile the regex
            Matcher matcher = pattern.matcher(text); // Begin matching
            while(matcher.find()) { // Iterate over all matches
                int start = matcher.start(); // Index of 1st char of this match
                int end = matcher.end(); // Index+1 of last char of this match
                int len = end - start; // Number of characters matched
                System.out.print(" ".repeat(start - pos) + "^");
                if(len > 2) System.out.print("-".repeat(len-2));
                if(len > 1) System.out.print("^");
                pos = end;
            }
        } catch (PatternSyntaxException e) {
            System.err.println("Cannot match " + regex + " to this:\n"
                + text + "\n---\n" + e.getMessage());
        } finally {
            System.out.println();
        }
    }
}
```

# regex Parser Output

```
ricegfa@antares:~/dev$ javac Regex.java
ricegfa@antares:~/dev$ java Regex
usage: java Regex [text] [regex]
ricegfa@antares:~/dev$ java Regex 'hello, world!' 'wor' # Match literal
hello, world!
      ^_^
ricegfa@antares:~/dev$ java Regex 'hello, world!' '..l' # . means any char
hello, world!
^_^      ^_^
ricegfa@antares:~/dev$ # \w is word (alpha) char, \s is whitespace, * is 0+, + is 1+
ricegfa@antares:~/dev$ java Regex 'We are CSE1325 students' '\w+\s*'
We are CSE1325 students
^_^^_^^^-----^^-----^
ricegfa@antares:~/dev$ # {3} is exactly 3 of preceding token, {3,4} is 3-4 of them
ricegfa@antares:~/dev$ java Regex 'We are CSE1325 students' '\w{3}\d{3,4}'
We are CSE1325 students
      ^-----^
ricegfa@antares:~/dev$
```





# Special and Escape Characters

- A non-special character matches itself, e.g., 'a' matches 'a'
- A special or escape character will match 0 or more characters
  - . matches any single character
    - \. matches a period – the backslash “escapes” the usual meaning
  - \s matches a whitespace character
    - \S matches anything except whitespace
  - \w matches a word char ([A-Za-z0-9\_])
    - \W matches anything else except whitespace
  - \d matches a digit ([0-9])
    - \D matches anything else except whitespace
  - \x represents a hexadecimal digit
    - \X matches anything else except whitespace
  - \n, \r, \t represent newline, carriage return, and tab, respectively

# Special Character Modifiers

- `{m,n}` matches the previous element at least m but no more than n times
  - `*` matches the previous element 0 or more times (same as `{0,}`)
  - `?` matches the previous element 0 or 1 times (same as `{0,1}`)
  - `+` matches the previous element 1 or more times (same as `{1,}`)
- `[ ]` matches any single character inside the brackets
  - `[^ ]` matches any character NOT inside the brackets
  - `[a-z]` matches any characters in the range a to z
  - `[cat|dog]` matches cat OR dog
- `^` matches the start and `$` the end of a line
- `( )` forms a sub-match for later reference
  - `\N` matches the N<sup>th</sup> sub-match's actual text (start at 1, NOT 0!)



# A More Useful Java Regex Example

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.Scanner;

public class Phone {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Pattern pattern = Pattern.compile("(\\d{3,3}-)?\\d{3,3}-\\d{4,4}");
        System.out.println("Enter 7- or 10-digit phone number:");
        while(in.hasNextLine()) {
            String phone = in.nextLine();
            if(pattern.matcher(phone).matches())
                System.out.println("Dialing " + phone);
            else
                System.err.println(phone + " is not a valid phone number!");
        }
    }
}
```

```
ricegfa@antares:~/dev$ java Phone
Enter 7- or 10-digit phone number:
817-924-1423
Dialing 817-924-1423
123-4567
Dialing 123-4567
1234567
1234567 is not a valid phone number!
817-787-123
817-787-123 is not a valid phone number!
(612)555-1212
(612)555-1212 is not a valid phone number!
ricegfa@antares:~/dev$
```

# More Character Matching Examples

- **a.c** matches abc, acc, a/c
- **r[au]n** matches ran or run
- **[^2-9][0-9]** matches 17 or 03 or K9, but not 21 or 99
- **3\.3\*** matches 3., 3.3, 3.33, or 3.33333333333
- **([0-9]+) = \1** matches 3 = 3 and 42 = 42, but not 21 = 39
- **^[hH]ello** matches hello or Hello, but only at the start of a line
- **Sincerely( yours)?,\$** matches Sincerely, or Sincerely yours, but only at the end of a line
- **Subject: (F[Ww]:|R[Ee]:)?(.\*)** matches an email subject line for forwards and replies only
- **[a-zA-Z][a-zA-Z\_0-9]\*** matches a C++ identifier



# More Character Matching Examples

- `Xa{2,3}` matches Xaa Xaaa
- `Xb{2}` matches Xbb
- `Xc{2,}` matches Xcc Xccc Xcccc Xcccc
- `int\s+x\s*=\s*\d+;` matches an integer definition
- `CSE\d{4}` matches a CSE class
- `\w{2}-\d{4,5}` matches 2 letters and 4 or 5 digits
- `\d{5}(-\d{4})?` matches a 5 or 9 digit zip code
- `[^aeiouy]` matches not an English vowel



# Regex and Beyond

- Regex are extremely powerful text manipulators
  - Excellent for validating input from dialogs (next section)
  - Make quick work of complex search-and-replace operations
- Regex are widely used outside Java
  - The gedit text editor can search (and replace) using regex
  - Command line tools like grep (literally “global regular expression print”), find, and awk use regex
  - Perl famously extends regex power to ludicrous (and incredibly useful!) extremes
- If you deal with much text, you need to learn regex!





# Regex: Learning More

- We won't focus on regex this semester
  - Even though they are *incredibly* useful
  - Moreso when writing dialog boxes next section!
- Tutorials are available for further understanding
  - Jenkov:  
<http://tutorials.jenkov.com/java-regex/index.html>
  - Oracle:  
<https://docs.oracle.com/javase/tutorial/essential/regex/>