Packages

- A *package* allows classes to be grouped together into a single unit which also acts as a scope.
- A class indicates that it is part of a package using the `package` statement (must be the first statement in a source file)
  - **`package`** *packageName* **`;`**
- Packages are imported using an import statement
  - **`import`** *packageName.className* **`;`**
  - **`import`** *packageName*`.*;`

Packages

- It is also possible to specify the name of a zip file in the CLASSPATH as well as directories containing class files.
- Any class from another package must be imported. for all Java classes this must be done explicitly with the exception of `java.lang`.
- If a class is not defined as part of a package, then it is considered to be part of the unnamed default package.

Mapping Packages to Files

- Package names map to directory names. Each directory contains all the `.class` files for a given package
  - `cs1.examples.stack` would map to `cs1/examples/stack`
  - the relative pathname is then appended to each entry in the CLASSPATH variable to create a full pathname
- Sun's recommended naming convention:
  - `edu.rit.cs.ptt.classes.cs1.examples`

Inner Classes

- Inner, or Nested, classes are standard classes declared within the scope of a standard top-level class.
- There are different kinds of inner class
  - nested top-level class
  - member class
  - local class
  - anonymous class

Nested Top-Level Classes

```
class outer {
  private static class NestedTopLevel {
    normal class stuff
  }
  normal class stuff
}
```

- Nested top-level classes are declared static within a top-level class (sort of like a *class* member).
- They follow the same rules as standard classes
  - `private static` classes cannot be seen outside the enclosing class
  - `public static` allows the class to be seen outside

LinkedStack2

```
public class LinkedStack2 {
  private StackNode tos = null;

  private static class StackNode {
    private Object data; private StackNode next, prev;

    public StackNode( Object o ) { this( o, null ); }
    public StackNode( Object o, StackNode n ) {
      data = o; next = n;
    }

    public StackNode getNext() { return next; }
    public Object getData() { return data; }
  }

  public boolean isEmpty() { return tos == null; }
  public boolean isFull() { return false; }
  public void push( Object o ) { tos = new StackNode( o, tos ); }
  public void pop() { tos = tos.getNext(); }
  public Object top() { return tos.getData(); }
}
```

Member Classes

- A member class is a nested top-level class that is ***not*** declared static.
- This means the member class has a `this` reference which refers to the enclosing class object.
- Member classes cannot declare static variables, methods or nested top-level classes.
- Member objects are used to create data structures that need to know about the object they are contained in.

Class5

```
class Class5 {

  private class Member {
    public void test() {
     i = i + 10;
     System.out.println( i );
     System.out.println( s );
    }
  }

  public void test() {
    Member n = new Member();
    n.test();
  }

  private int i = 10;
  private String "Hello";
}
```


`this` Revisited

- To support member classes several extra kinds of expressions are provided
  - `x = this.dataMember` is valid only if `dataMember` is an instance variable declared by the member class, not if `dataMember` belongs to the enclosing class.
  - `x = EnclosingClass.this.dataMember` allows access to `dataMember` that belongs to the enclosing class.
- Inner classes can be nested to any depth and the `this` mechanism can be used with nesting.

`this` and Member Classes

```
public class EnclosingClass {
  private int i,j;

  private class MemberClass {
    private int i;  public int j;

    public void aMethod( int i ) {
      int a = i;                      // Assign param to a
      int b = this.i;                 // Assign member's i to b
      int c = EnclosingClass.this.i;  // Assign top-level's i to c
      int d = j;                      // Assign top-level's j to d
    }  }

  public void aMethod() {
    MemberClass mem = new MemberClass();
    mem.aMethod( 10 );
    System.out.println( mem.i + mem.j );  // is this a bug?
  }}
```


`new` Revisited

- Member class objects can only be created if they have access to an enclosing class object.
- This happens by default if the member class object is created by an instance method belonging to its enclosing class.
- Otherwise it is possible to specify an enclosing class object using the `new` operator as follows
  - `MemberClass b = anEnclosingClass.new MemberClass();`

Local Classes

- A local class is a class declared within the scope of a compound statement, like a local variable.
- A local class is a member class, but cannot include static variables, methods or classes. Additionally they cannot be declared `public`, `protected`, `private` or `static`.
- A local class has the ability to access *final* variables and parameters in the enclosing scope.

Local Class Example

```
public class EnclosingClass {
  String name = "Local class example";

  public void aMethod( final int h, int w ) {
    int j = 20; final int k = 30;

    class LocalClass {
      public void aMethod() {
        System.out.println( h );
        // System.out.println( w ); ERROR w is not final
        // System.out.println( j ); ERROR j is not final
         System.out.println( k );
         // System.out.println( i ); ERROR i is not declared yet
         System.out.println( name); // normal member access
    }}

    LocalClass l = new LocalClass(); l.aMethod();
    final int i = 10; }

    public static void main() {
      EnclosingClass c = new EnclosingClass(); c.aMethod( 10, 50 ); }}
```


Anonymous Classes

- An anonymous class is a local class that does not have a name.
- An anonymous class allows an object to be created using an expression that combines object creation with the declaration of the class.
- This avoids naming a class, at the cost of only ever being able to create one instance of that anonymous class.
- This is handy in the AWT.

Anonymous Class Syntax

- An anonymous class is defined as part of a new expression and *must* be a subclass or implement an interface

```
new className( argumentList ) { classBody }
new interfaceName() { classBody }
```

- The class body can define methods but cannot define any constructors.
- The restrictions imposed on local classes also apply

Using Anonymous Classes

```
import java.awt.*; import java.awt.event.*;import javax.swing.*;

public class MainProg {
  JFrame win;

  public MainProg( String title ) {
    win = new JFrame( title );

    win.addWindowListener(
      new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
          System.exit( 0 );
       }});
  }

  public static void main( String args[] ) {
       MainProg x = new MainProg( "Simple Example" );
    }}
```