Summary Programming and Software Development: Lecture (s) all

Programming and Software Development (University of Melbourne)

CHAPTER 1: INTRODUCTION

```
CLASS
public class Name { }

Main method
public static void main (string[] args){ }
```

CHAPTER 2: INPUT OUTPUT

PRINT

```
System.out.
println()               next line
print()                       same line
printf("Start%8.2fEnd", value)      format specifier
      System.out.printf("$%6.2f for each %s.", price, name);
      System.out.printf("$%6.2f for each %s.%n", price, name); line break

%5d         %d    decimal integer
%6.2f       %f    floating point
%8.3e       %e    e notation
%8.3g       %g    java decides e notation or not
%12s        %s    String
%2c         %c    character
            %n    line break

Common used %.2f


String methods:
```

Method	Use
s.length()	The number of characters in the string s
s.equals(s1)	Are the strings identical
s.compareTo(s1)	Negative if s < s1, zero if s = s1, positive if s > s1
equalsIgnoreCase, compareToIgnoreCase	case-insensitive versions of equals and compareTo

Method	Use
s.trim()	s with leading/trailing whitespace removed
s.charAt(n)	The character at position n (0 origin)
s.toLowerCase() s.toUpperCase()	All lower (upper) case version of string
s.Substring(n,n1)	Substring of s from character n up to but *not* including character n1
s.indexOf(s1)	Position of first appearance of s1 in s

Money formats (number to String)

```
import java.text.NumberFormat

...
NumberFormat moneyFormater = NumberFormat.getCurrencyInstance();
    System.out.println(moneyFormater.format(19.8));
    System.out.println(moneyFormater.format(19.81111));

Output:
$19.80
$19.81
```

```
import java.util.Locale;
NumberFormat.getCurrencyInstance(Locale.US)
```

IMPORT

– Classes in **java.lang** package are imported automatically

Others

```
import java.util.Scanner;     // import the Scanner class only
import java.text.*;           //import all the classes in package java.text
```

DecimalFormat Class

```
Import java.text.DecimalFormat;
...
DecimalFormat pattern00dot000 = new DecimalFormat("00.000");
DecimalFormat pattern0dot00 = new DecimalFormat("0.00");


Double d = 12.3456789;
System.out.println("Pattern 00.000");
System.out.println(pattern00dot000.format(d));
System.out.println("Pattern 0.00");
System.out.println(pattern0dot00.format(d));
```

Output

```
Pattern 00.000
12.346
Pattern 0.00
12.35
```

Other patterns (formats)

"0.00%" "#0.###E0" "#00.###E0"

Sccanner Class

```
import java.util.Scanner
...
Scanner keyboard = new Scanner(System.in);
```

Methods:

```
int numberOfPods = keyboard.nextInt();
double d1 = keyboard.nextDouble();
String word1 = keyboard.next();
String line = keyboard.nextLine();              until '\n' character but not included
```

Dealing with the '\n' line terminator

Extra String junk = keyboard.nextLine(); is required after an (next, nextInt, ...) to read another input.

Display 2.8 Methods of the Scanner Class

Scannner_Object_Name.nextLong()

Returns the next value of type long that is typed on the keyboard.

Scannner_Object_Name.nextByte()

Returns the next value of type byte that is typed on the keyboard.

Scannner_Object_Name.nextShort()

Returns the next value of type short that is typed on the keyboard.

Scannner_Object_Name.nextDouble()

Returns the next value of type double that is typed on the keyboard.

Scannner_Object_Name.nextFloat()

Returns the next value of type float that is typed on the keyboard.

(continued)

Display 2.8 Methods of the Scanner Class

Scannner_Object_Name.next()

Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

Scannner_Object_Name.nextBoolean()

Returns the next value of type boolean that is typed on the keyboard. The values of true and false are entered as the strings "true" and "false". Any combination of upper- and/or lowercase letters is allowed in spelling "true" and "false".

Scanner_Object_Name.nextLine()

Reads the rest of the current keyboard input line and returns the characters read as a value of type String. Note that the line terminator '\n' is read and discarded; it is not included in the string returned.

Scanner_Object_Name.useDelimiter(*New_Delimiter*);

Changes the delimiter for keyboard input with *Scanner_Object_Name*. The *New_Delimiter* is a value of type String. After this statement is executed, *New_Delimiter* is the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" for details.

Other input delimiters than WHITESPACES (tab, space, enter, line)

```
Scanner keyboard2 = new Scanner(System.in);
Keyboard2.useDelimiter("##");
```

CHAPTER 3: FLOW CONTROL

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	x + 7 == 2*y answer == 'y'
≠	Not equal to	!=	score != 0 answer != 'y'
>	Greater than	>	time > limit
≥	Greater than or equal to	>=	age >= 21
<	Less than	<	pressure < max
≤	Less than or equal to	<=	time <= limit

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

- all uppercase letters come before lowercase letters
- Use **(min < result) && (result < max)** rather than **min < result < max**

- In this case, use the **&** and **|** for complete evaluation
- operators instead of **&&** and **||** for short circuit evaluation (can avoid 2nd st errors)

Display 3.6 **Precedence and Associativity Rules**

	Precedence From highest at top to lowest at bottom. Operators in the same group have equal precedence.	Associativity
Highest Precedence (Grouped First) ↓	Dot operator, array indexing, and method invocation ., [ ], ( )	Left to right
	++ (postfix, as in x++), −− (postfix)	Right to left
	The unary operators: +, −, ++ (prefix, as in ++x), −− (prefix), and !	Right to left
	Type casts (*Type*)	Right to left
	The binary operators *, /, %	Left to right
	The binary operators +, −	Left to right
	The binary operators <, >, <=, >=	Left to right
	The binary operators ==, !=	Left to right
	The binary operator &	Left to right
	The binary operator |	Left to right
	The binary operator &&	Left to right
	The binary operator ||	Left to right
	The ternary operator (conditional operator) ?:	Right to left
Lowest Precedence (Grouped Last)	The assignment operators: =, *=, /=, %=, +=, −=, &=, |=	Right to left

IF-ELSE

```
if (Boolean_Expression)
   Statement_1
else if (Boolean_Expression)
   Statement_2
          ...
else if (Boolean_Expression_n)
   Statement_n
else
  Statement_For_All_Other_Possibilities
```

SWITCH The controlling expression must evaluate to a char, int, short, or byte

- **break** statement omitted will not issue a **compiling error**.

```
switch (Controlling_Expression)
{
  case Case_Label_1:                    Controlling_Expression must match a Case_label_#
          Statement_Sequence_1;
          break;
  case Case_Label_2:
```

```
            Statement_Sequence_2;
            break;
 ...
  case Case_Label_n:
            Statement_Sequence_n;
            break;
  default:
            Default_Statement Sequence;
            break;
}
```

The Conditional Operator

```
if (n1 > n2)
      max = n1;
else
      max = n2;
```

THE SAME AS:

```
max = (n1 > n2) ? n1 : n2;
```

LOOPS

- The code that is repeated in a loop is called the ***body*** of the loop
- Each repetition of the loop body is called an ***iteration*** of the loop

WHILE

```
while (Boolean_Expression)
{
   Statement_1;
   Statement_2;
   ...
   Statement_Last;
}
```

DO - WHILE

```
do
```

```
{
   Statement_1;
   Statement_2;
   Statement_Last;
}  while (Boolean_Expression);
```

FOR

```
for (Initialization; Boolean_Expression; Update)
    Body
```

EXAMPLE:

```
for (number = 100; number >= 0; number--)
    System.out.println(number
         + " bottles of beer on the shelf.");
```

break;	finish, end	inner most loop or switch
continue;	in loops, end iteration	inner most loop

Label a loop:

someIdentifier: for(xxx;xxx;xxx)

Labeled break statement

break someIdentifier; end any labeled loop

End a program

System.exit(0);

- zero argument is used to indicate a normal ending of the program
- "1" argument indicate error and ending

- **An off-by-one error is when a loop repeats the loop body one too many or one too few times**

Tracing Variables System.out.println("n = " + n); // Tracing n

Assertion Checks assert Boolean_Expression; //if false the program ends

To run with assertions on java –enableassertions ProgramName

- An assertion must be either true or false, and **should be true if a program is working properly**

- Assertions can be placed in a program as comments

CHAPTER 4: DEFINING CLASSES I

A Class is a Type

- If **A is a class**, then the phrases "bla is of **type A**," "bla is an object of the **class A**," and "bla is an instance of the **class A**" mean the same thing

A **Class** definition contains — **instance variables** (declared and initialized)

ClassName — **methods** (definitions)

CREATE OBJECTS (in other outer classes)

Declaration: **ClassName classVar;**

Creation: **classVar = new ClassName();**

ClassName classVar = new ClassName();

Declaring instance variables: (should always be private)_

```
private String instanceVar1;
private int instanceVar2;
```

private it **is enough** for **primitive types**

not for **class types** instance variable

Refering instance variables: classObject.instanceVar

```
classVar.instanceVar1
classVar.instanceVar2
```

Method definitions

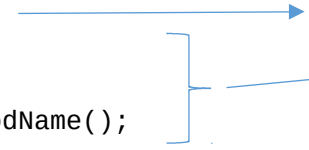
```
public void methodName()
 {
    code  to perform some action
    and/or compute a value
 }
```

Heading

Body

Methods invocation or calling

```
classVar.methodName();
typeReturned tRVariable;
tRVariable = classVar.methodName();
```



void method

return value and assign to a variable

public typeReturned methodName(paramList)

(not necessary to create a variable)

Two kinds of methods **public <void *or* typeReturned> myMethod()**

void methods Perform an action

public void methodName(paramList) { }

return; only if early end is required

return methods Return a value

public typeReturned methodName(paramList){

return Expression; (of typeReturned)

}

PARAM LIST (in definition)

public double myMethod(int p1, int p2, double p3)

int a=1,b=2,c=3;

double result = myMethod(a,b,c); ***call-by-value mechanism***

c is cast to a **double**

type cast **byte→short→int→long→float→double**

char

- Formal parameters can be local variables and change within method but **don't change the argument value**. (for **primitive types parameters**)
- **Class type parameters** appear to behave differently from primitive type parameters
  - They appear to behave in a way similar to parameters in languages that have the ***call-by-reference*** parameter passing mechanism
  - Any change made to the object named by the **parameter** (i.e., **changes made to the values of its instance variables**) will be made to the object named by the **argument**, because they are the same object

THIS parameter (hidden) classVar.myMethod(int someVariable, ClassName this)

`private int someVariable = 5` this = classVar **hidden**

int someVariable = this.someVariable

local **instance**

- **if-else, while** and **for-loops** can invoke a **method (boolean type returned)** as ***Boolean_Expression***.

equals and toString Methods expected in almost all classes

public boolean equals(ClassName objectName) {...}

to **compare two objects** of the class: true or false

The **==** operator only checks that two class type variables have the **same memory address**

Two objects in **two different locations** whose **instance variables have exactly the same values** would still test as being "**not equal**" ("==" = false)

To check if an object has "no real" values (null)

Variable initialized in null

YourClass yourObject = null

- Don't use **EQUALS** because it is checking the reference, no the object

  Method **invoking a null object** gives a "**Null Pointer Exception**" **error** message

Complete example with different equals methods: (in person class)

```
public boolean equals(Person otherPerson)
{
  if (otherPerson == null)
    return false;
  else
    return (name.equals(otherPerson.name) &&
            born.equals(otherPerson.born) &&
            datesMatch(died, otherPerson.died));      (no equals because can be null)
}
```

public String toString() {...}

to return a **String** value that represents the data in the object

Complete example with different toString methods: (in person class)

```
public String toString( )
{
  String diedString;
  if (died == null)
    diedString = ""; //Empty string
  else
    diedString = died.toString( );
  return (name + ", " + born + "-" + diedString);
}
    (string.toString)     (date.toString)
```

TESTING METHODS

- Each method should be tested in a program in which it is the only untested program
  - A **program** whose only purpose is **to test a method** is called a ***driver program***
- One method often invokes other methods, so one way to do this is to **first test all the methods invoked by that method**, **and then test the method itself**
  - This is called ***bottom-up testing***
- Sometimes it is necessary to test a method before another method it depends on is finished or tested
  - In this case, use a **simplified version of the method,** called a ***stub***, to return a value for testing

PUBLIC AND PRIVATE MODIFIERS

- It is considered good programming practice to make all **instance variables private**
- Methods to be **used by other classes** must be made **public**
- Methods **not needed outside the class** should be made **private**

ACCESSOR AND MUTATOR METHODS

```
public int getDay( )
    {return day;}
public int getYear( )
    {return year;}
```

Accessors

```
public void setDay(int day)
    {if ((day <= 0) || (day > 31))
        {   System.out.println("Fatal Error");
            System.exit(0);
        }
     Else
     this.day = day;
    }
```

Mutator (some cases is better return a boolean value when change doesn't make sense)

AVOID PRIVACY LEAKS CREATING COPIES OF CLASS TYPE VARIABLES USING COPY CONSTRUCTORS

```
public Date getBirthDate()    {  return born;   }          //dangerous
public Date getBirthDate()    {  return new Date(born); }  //correct
```

MUTABLE AND IMMUTABLE CLASSES

- A class that contains **no methods (other than constructors)** that **change any of the data** in an object of the class is called an ***immutable class***
- **No Mutators**

Only with immutable classes is possible return a reference with no privacy leaks:

```
public Date getBirthDate()    {  return born;   }
```

DEEP AND SHALLOW COPY

- A ***deep copy*** of an object is a copy that, with one exception, has no references in common with the original
  - Exception: **References to immutable objects** are allowed to be shared
- Any copy that is not a deep copy is called a *shallow copy* (dangerous privacy leaks)

Overloading a method

- two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have **different *signatures***
  - A **signature** consists of the name of a method together with its parameter list
- If Java cannot find a method signature that exactly matches a method invocation, it will try to use **automatic type conversion**
  - **Ambiguous method invocations will produce an error in Java**
- Java does **not permit** methods with the **same name and different return types** in the same class

Constructors

- special kind of method that is designed to initialize the instance variables for an object:

```
public class ClassName {...
public ClassName(anyParameters){code}          (no type returned, not even void)
...}                                           (also has a this parameter)
```

- Constructors are typically **overloaded** (several constructors in a Class)

EXAMPLE

public Person(String initialName, Date birthDate, Date deathDate)

COPY CONSTRUCTOR

```
public Date(Date aDate){                    (Class with primitive types instances variables)
  if (aDate == null)                        //Not a real date.
  {
    System.out.println("Fatal Error.");
    System.exit(0);
  }
  month = aDate.month;
  day = aDate.day;
  year = aDate.year;
}
```

public Person(Person original) {code} (Class with Class types instances variables)

DANGEROUS: it doesn't create an independent copy

```
born = original.born //dangerous
died = original.died //dangerous
```

CORRECT:

```
born = new Date(original.born);     (Use an already correct copy constructor)
died = new Date(original.died);
```

Complete code:

```
public Person(Person original)
{
  if (original == null)
  {
    System.out.println("Fatal error.");
    System.exit(0);
  }
  name = original.name;                     //no problem because is a String type (IMMUTABLE)
  born = new Date(original.born);
  if (original.died == null)
    died = null;
  else
    died = new Date(original.died);
}
```

Invoking constructors (outside the class)

```
ClassName objectName = new ClassName(anyArgs);
```

- If a constructor is invoked again (using **new**), the **first object** is discarded and an entirely **new object** is created

  - If you need to **change the values** of instance variables of the **object**, use **mutator methods** instead

- Always provide a No-argument construct to set the instance variables to default values, if none construct is provided, java provide one.

CHECK this AND super CONSTRUCTORS

Default variable initializations

- Instance variables are automatically initialized in Java
  - **boolean** types are initialized to false
  - **Other primitives** are initialized to the zero of their type
  - **Class** types are initialized to null
- However, it is a **better practice to explicitly initialize instance variables** in a constructor
- Note: **Local variables** are **not automatically initialized**

STATIC VARIABLES (*class variable*)

private static int myStaticVariable = 0; (or just declared)

- A *static variable* is a variable that belongs to the class as a whole, and not just to one object
  - There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
- All objects of the class can read and change a static variable
- Should always be defined as **private**

CONSTANTS (statics but with final, can be publics)

public static final int DAYS_PER_WEEK = 7;

Referring a constant (outside the class) (The Class despite of the Object)

int year = MyClass.DAYS_PER_WEEK;

STATIC METHODS

- A *static method* is one that can be used **without a calling object**
- Cannot refer to an instance variable, don't have **this** parameter and cannot invoke a non-static method, **just can** invoke another **static method**.

public static returnedType myMethod(parameters) { . . . }

Invoking a static method (outside the class) (The Class despite of the Object)

returnedValue = MyClass.myMethod(arguments);

- Any regular class can contain a **main method** (useful to diagnostic)

The Math Class (included in java.lang package)

area = Math.PI * radius * radius;

PI is a constant of Math Class

All methods of Math Class are statics:

```
public static double pow(double base, double exponent)
```

Returns base to the power exponent.

EXAMPLE

Math.pow(2.4, 3.2) returns 18.6.

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the argument. (The method name abs is overloaded to produce four similar methods.)

EXAMPLE

Math.abs(−6) and Math.abs(6) both return 6. Math.abs(−5.5) and Math.abs(5.5) both return 5.5.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

EXAMPLE

Math.min(3, 2) returns 2.

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

EXAMPLE

Math.max(3, 2) returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

Rounds its argument.

EXAMPLE

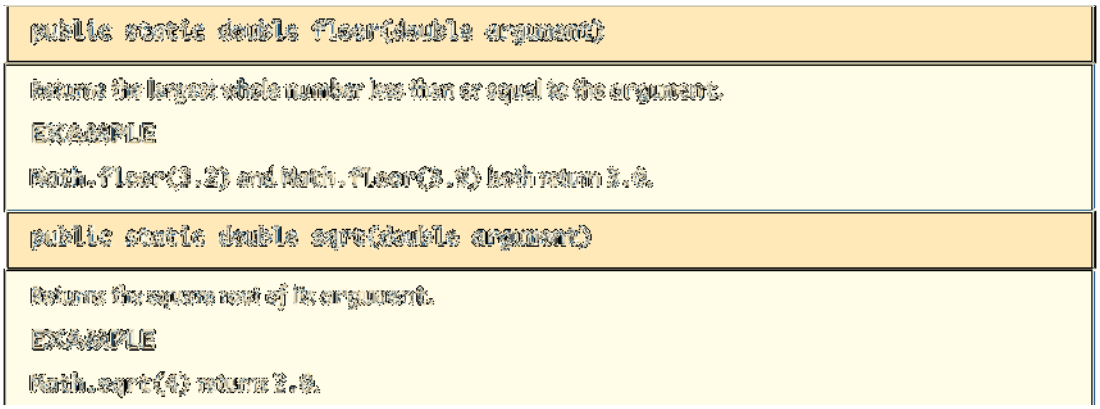
Math.round(3.2) returns 3; Math.round(3.5) returns 4.

```
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the argument.

EXAMPLE

Math.ceil(3.2) and Math.ceil(3.9) both return 4.0.



WRAPPER CLASSES (also contain CONSTANTS and STATIC methods, but don't support +, -, *, / operations)

the primitive types **boolean**, **byte**, **short**, **int**, **long**, **float**, **double**, and **char**

The wrapper classes **Boolean**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, and **Character**

BOXING

- from a **value of a primitive type** to an **object of its wrapper class.**

```
Integer integerObject = new Integer(42);
Integer integerObject = 42;              (java 5.0)
```

UNBOXING

- from an **object of a wrapper class** to the corresponding **value of a primitive type.**

Wrapper classes: **Boolean, Byte, Short, Integer, Long, Float, Double,** and **Character**

The methods: **booleanValue, byteValue, shortValue, intValue, longValue, floatValue, doubleValue,** and **charValue**

```
int i = integerObject.intValue();
int i = integerObject;                   (java 5.0)
```

Some Static Methods

WrapperClass.parse*(String string) (String to primitiveType)

String.valueOf(PrimitiveType type) (primitiveType to String)

```
int x = Integer.parseInt("34");          // x=34
```

```
double y = Double.parseDouble("34.7");    // y =34.7

String s1 = String.valueOf('a');          // s1="a"

String s2 = String.valueOf(true);         // s2="true"

(Another option)

String s3 = Integer.toString(60);         // s3="60"
```

THE CLASS INVARIANT (A statement that **is always true for every object of the class**)

public Person(String initialName, Date birthDate, Date deathDate) **(CONSTRUCTOR)**

```
{
  if (consistent(birthDate, deathDate))
  { name = initialName;
    born = new Date(birthDate);
    if (deathDate == null)
      died = null;
    else
      died = new Date(deathDate);
  }
  else
  { System.out.println("Inconsistent dates.");
    System.exit(0);
  }
}

/**  Class invariant: A Person always has a date of birth,
     and if the Person has a date of death, then the date of
     death is equal to or later than the date of birth.
     To be consistent, birthDate must not be null. If there
     is no date of death (deathDate == null), that is
     consistent with any birthDate. Otherwise, the birthDate
     must come before or be equal to the deathDate.
*/
private static boolean consistent(Date birthDate, Date
                                                  deathDate)
{
    if (birthDate == null)  return false;
    else if (deathDate == null)  return true;
    else  return (birthDate.precedes(deathDate) ||
                  birthDate.equals(deathDate));
 }
```

CLASS	public Class MyClass{
INSTANCE VARIABLES	private ClassType myVar;
STATIC VARIABLES	
CONSTANTS	public static final int IVA = 12

```
      OTHERS (PRIVATE)        private static int other = 15 (or empty)
CONSTRUCTORS
      DEFAULT                 MyClass()
      COMPLETE                MyClass(ClassType aVar)
      COPY                    MyClass(MyClass original)
METHODS (public, private)
      STATIC      (helpers)   public static void ..., public static typeReturn myStaMet(par)
      NON STATIC              public void myMethod(para), public typeReturn myMethod2(para)
                              Mutators(setMyVar)            Accessors(getMyVar(no args))
            EQUALS            public boolean equals(MyClass otherObject)
            toString          public String toString()
      MAIN  (to diagnostic)   public static void main(String[] args)

MAIN CLASS                    public class MainClass {
                              public static void main(String[] args) {
OBJECTS CREATION              MyClass newObject =     new MyClass()
                                                      new MyClass(ClassType aValue)
                              MyClass copyObject=     new MyClass(newObject)
INVOKATIONS
      METHODS
            NON STATIC
                  void              newObject.myMethod(---)
                  typeReturn        typeReturn x = newObject.myMethod2(---)
            STATIC            MyClass. myStaMet(---)
PRINT                         System.out.println(newObject)
                              System.out.println(newObject.myVar)
```

ARRAYS *BaseType*[] *ArrayName* = new *BaseType*[*size*];

holds a reference Uncommon constructor

Declaration and Creation of an array of five "scores"

double[] score = new double[5]; **[int]** **array of doubles**

- int variable

- **indexed variables** (subscripted variables or elements): (must be of the same type, called the ***base type*** of the array)

score[0], score[1], score[2], score[3], score[4] **ArrayName[index]**

max = score[0]; - expression

score[0] = 100; //indexed variable initialized - variable

//not common

Complete initialization

int[] age = {2, 12, 1};// age.length = 3

- It can exist Partially Filled Arrays, declared to be of the largest size that the program could possibly need

Manipulations (for-loop)

For (index = 0; index < 5; index++) {...}

AN **ARRAY** (as an object) HAS JUST **ONE INSTANCE VARIABLE**: **length** (cannot be changed)

- An **out of bounds index** will cause a program to terminate with a ***run-time error message***

An Array of characters is not a String

char[] a = {'A', 'B', 'C'};

String s = a; **//Illegal!**

Instead use some constructors of **String** Class:
String s = new String(a);
String s2 = new String(a,0,2);

System.out.println(s); "ABC"

System.out.println(s2); "AB"

System.out.println(a); "ABC" //same output

- **Like class types,** a variable of an **array type holds a *reference***
- **Array types** are (usually) considered to be **object types**

Array parameters (methods does can change the argument)

```
public class SampleClass {
      public static void doubleElements(double[] a)
      {
      for (int i = 0; i < a.length; i++)
            a[i] = a[i]*2;                    //change the values of a[i]
      }
}
```

- This method *does* change the content of the array it is passed

Given:
```
double[] a = new double[10];
double[] b = new double[30];
```

Invokations like: (outside the SampleClass)
```
SampleClass.doubleElements(a);          //Note doubleElements is a static method
SampleClass.doubleElements(b);          //Change the arrays
```

= and ==

- the assignment operator (**=**) only copies this memory address.
- **b = a;**
- The memory address in **a** is now **the same as** the memory address in **b**: They reference the same array
- For the same reason, **(a == b)** will be **true** if **a** and **b** share the same memory address

equalsArray (in the same way that equals for Class types)

```
public static boolean equalsArray(int[] a, int[] b)
{
  if (a.length != b.length)  return false;
  else
  {
    int i = 0;
    while (i < a.length)
    {
      if (a[i] != b[i])
        return false;
      i++;
    }
  }
  return true;
}
```

ACCESSOR METHODS (can cause privacy leaks)

private String[] s instance variable

A. Write the accessor to take an index i as input and return only a[i] (CORRECT)

B. Make a new array, copy contents of s into it, and return the copy (CORRECT)

```
public double[] getArray()
{
  return anArray;//BAD!
}
```

CORRECT

```
public double[] getArray()
{
  double[] temp = new double[count];            //count is an inst var for size
  for (int i = 0; i < count; i++)
    temp[i] = a[i];                             //a is the array inst var
  return temp
}
```

```
public ClassType[] getArray()
{
  ClassType[] temp = new ClassType[count];
  for (int i = 0; i < count; i++)
    temp[i] = new ClassType(someArray[i]);      //copy constructor of ClassType
  return temp;
}
```

MULTIDIMENSIONAL ARRAYS

```
double[][]table = new double[100][10];

int[][][] figure = new int[10][20][30];

Person[][] = new Person[10][100];
```

Two-dimensional array: the **first index** giving the **row**,
and the **second index** giving the **column**

```
char[][] a = new char[5][12];
a.length       equals    5
a[0].length    equals    12
   i
```

SELECTION SORT

```
for (int index = 0; index < count; index++)

  Place the indexth smallest element in a[index]
```

FOR EACH LOOP

- *for-each loop* or *enhanced for loop for a* ***collection of values not indexed. (not arrays)***

```
for (ArrayBaseType VariableName : ArrayName)
```

Statement

ENUMERATED TYPES

- The definition of an enumerated type is normally placed **outside of all methods** in the **same place that named constants** are defined:

enum TypeName {VALUE_1, VALUE_2, …, VALUE_N};

enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};

WorkDay meetingDay, availableDay; **//declaration**

meetingDay = WorkDay.THURSDAY; **//set or initialization**

availableDay = null;

WorkDay meetingDay = WorkDay.THURSDAY;

CA BE PRINT DIRECTLY

System.out.println(**meetingDay**); output: THURSDAY

System.out.println(**WorkDay.THURSDAY**); output: THURSDAY

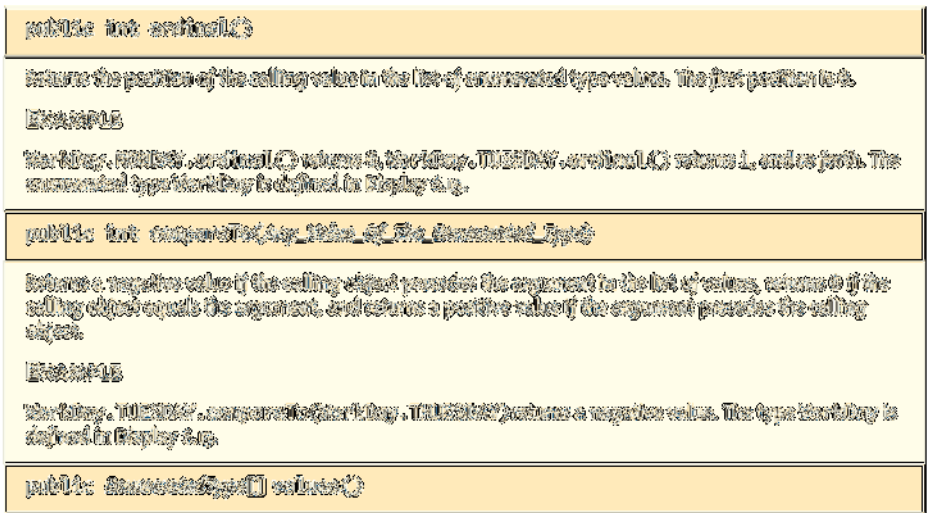
- may look like **String** values, but they don't.
- However, they **can be used for tasks** which could be done by **String values** and, in some cases, **work better**

Two Enumerated types can be compared with (==) or equals. == is better

```
if (meetingDay == availableDay)
      statement
```

METHODS OF ENUMERATED TYPES

Obviously:	equals and	toString
	Value1.equals(Value2)	workday.MONDAY.toString()
	public boolean equals(...)	public String toString()



SEE BELOW (static??)

Returns an array whose elements are the values of the enumerated type in the order in which they are listed in the definition of the enumerated type.

EXAMPLE

See Display 6.15.

`public static EnumeratedType valueOf(String name)`

Returns the enumerated type value with the specified name. The string `name` must be an exact match.

EXAMPLE

`WorkDay.valueOf("THURSDAY")` returns `WorkDay.THURSDAY`. The type `WorkDay` is defined in Display 6.13.

The values METHOD

- It returns an **array** whose elements are the values of the **enumerated type given in the order** in which the elements are listed in the definition of the enumerated type
- The **base type** of the array that is returned is the **enumerated type**

```
WorkDay day = WorkDay.values();          //day is an array of all days in order
                                         //so, day.length can be used
```

SWITCH

Display 6.16 Enumerated Type in a `switch` Statement

```
 1  import java.util.Scanner;
 2
 3  public class EnumSwitchDemo
 4  {
 5      enum Flavor {VANILLA, CHOCOLATE, STRAWBERRY};

 6      public static void main(String[] args)
 7      {
 8          Flavor favorite = null;
 9          Scanner keyboard = new Scanner(System.in);

10          System.out.println("What is your favorite flavor?");
11          String answer = keyboard.next();
12          answer = answer.toUpperCase();
13          favorite = Flavor.valueOf(answer);

                                        The case labels must have just the name of
                                        the value without the type name and dot.
14          switch (favorite)
15          {
16              case VANILLA:
17                  System.out.println("Classic");
18                  break;
19              case CHOCOLATE:
20                  System.out.println("Rich");
21                  break;
22              default:
23                  System.out.println("I bet you said STRAWBERRY.");
24                  break;
25          }
26      }
27  }
```

Gives error if input is not in the Flavor enumerated types

INHERITANCE

- A derived class automatically **has all the instance variables and methods that the base class** has, and **it can have additional methods and/or instance variables** as well
- Inheritance is especially advantageous because it allows code to be ***reused***, without having to copy it into the definitions of the derived classes

    - The original class is called the ***base/parent/super*** *class*
        - ***Employee is the BaseClass***
        - ***public class Employee {***
            - instance variables **name** and **hireDate**

    - The new class is called a ***derived/child/sub*** *class*
        - **extends BaseClass**
        - **public class HourlyEmployee extends Employee {**
            - additional instance variables **wageRate** and **hours**
            - and inherits **name** and **hireDate**
            - inherits all the **public** methods and **all the static variables** from the base class. Can **add more**.
            - Also **can change or *override* an inherited method** if necessary
            - Changing **returning class type** to a its **descendant class type** (*covariant return type*)
            - An **object of a derived class** has the **type of every one of its ancestor classes**
            - Therefore, an **object of a derived class** can be **assigned to a variable of any ancestor type,** and **can be used anyplace** that an object of any **of its ancestor types** can be used

OVERRIDING METHODS

CODE

BASE CLASS

```
public class BaseClass
      { . . .
      public Employee getSomeone(int someKey)
      . . .
```

DERIVED CLASS

```
public class DerivedClass extends BaseClass
{ . . .
public HourlyEmployee getSomeone(int someKey)
. . .
```

ACCESS PERMISSION

METHOD IN BASE CLASS	CHANGE	OVERRIDING METHOD IN DERIVED CLASS
private	**can**	to public (more accessible)
public	**cannot**	to private (more restrictive)

OVERRIDING IS NOT THE SAME THAT OVERLOADING

- OVERRINDING: The new method has the **exact same number and types of parameters** as in the **base class**
- OVERLOADING: The method is just overloading (**has different signature**), because the **other method still is inherited form base class**

The final modifier

final myMethod	myMethod may not be redefined in a derived class
final MyClass	MyClass may not be used as a base class to derive other classes

The super Constructor

- A **derived class** uses a **constructor from the base class** to initialize all the data inherited from the base class

INVOKATION in a derived class constructor

```
public derivedClass(int p1, int p2, double p3)
  {
    super(p1, p2);                    //calls the base class constructor
    instanceVariable = p3;            //Inst Var from the derived class
  }
```

- Always the first action in a derived class constructor
- **Never** use an instance variable as argument
- If there is not a **super** invocation, then automatically: **super()**

INVOKING BASE CLASS VERSION OF OVERRIDING METHOD

INSIDE A DERIVED CLASS

```
public String toString()
{
     return (super.toString() + "$" + wageRate);
}
```

USES super ONLY FOR DIRECT PARENTS

The this Constructor

same **super** constructor rules

- Within the **definition of a constructor**, this can be used for invoking **another constructor in the same class**
- If **super** and **this** is needed, so **first call this** and then inside this must be a **super** constructor
  - No-argument constructor (invokes explicit-value constructor using **this** and default arguments):

```
public ClassName()
{
     this(argument1, argument2);
}
```

  - Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2)
{
     . . .
}
```

IS-A versus HAS-A RELATIONSHIPS

SUPER CLASS	**DERIVED CLASS**	
SuperClassType	DerivedClassType	
DerivedClassType	SuperClassType	NO

DerivedClassType	**IS A**	SuperClassType
Complex		Simple

- For example, an **HourlyEmployee *"is an"* Employee**
- The **Employee** class contains an instance variable, **hireDate**, of the class **Date**, so therefore, an **Employee *"has a"* Date (composition)**
- **HourlyEmployee *"is an"* Employee** and ***"has a"* Date**

Encapsulation and Inheritance

- **Private instance variable** in a base class is **not accessible** *by name* in the **definition of a method** in any other class, **not** even in a **method definition of a derived class**

	Base	Derived
Class	Employee	HouryEmployee
Inst var	hireDate	

HourlyEmployee class **cannot access** the private instance variable **hireDate by name**, even though it is inherited from the Employee base class

Can only be accessed by the **public accessor and mutator**

  - An object of the **HourlyEmployee** class can use the **getHireDate** or **setHireDate** methods to access **hireDate**

- **Private methods** in a base class are **not** *directly* **available.**

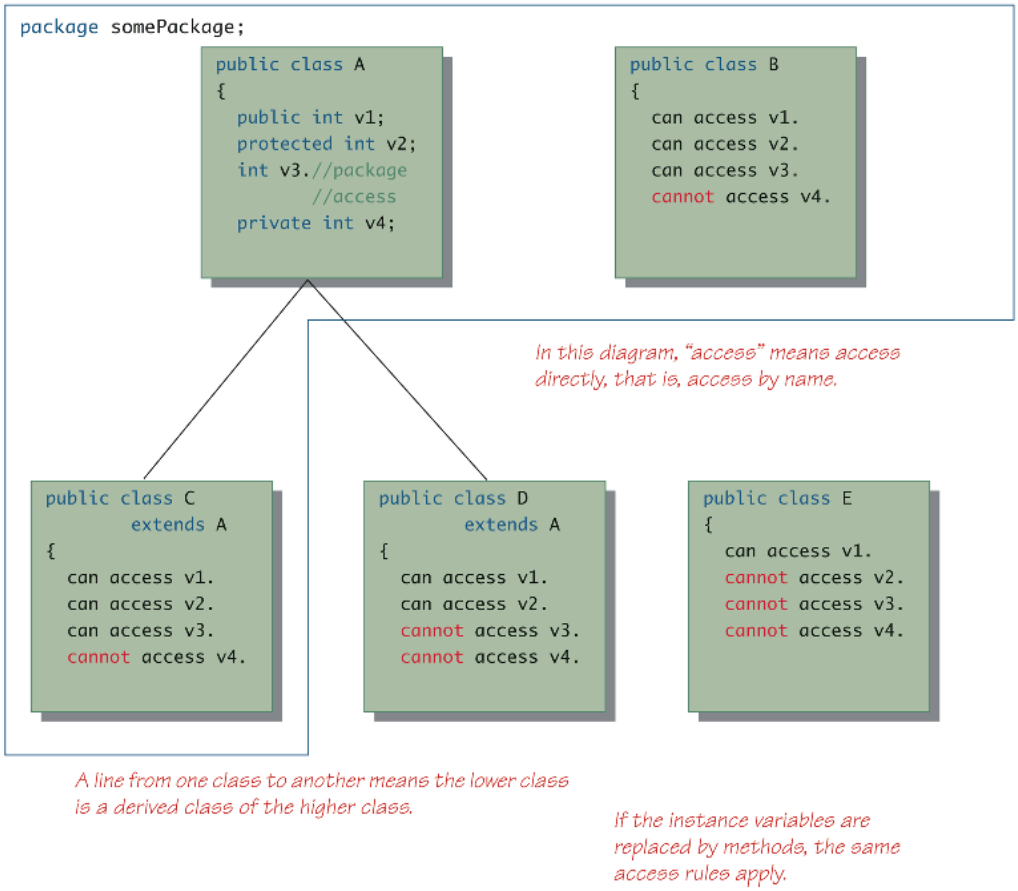
protected and **package** Access

Method or instance variable **protected** (rather than **public** or **private**), can be accessed by name.

  - Inside **its own class** definition
  - Inside **any class derived** from it
  - In the definition of **any class in the same package**

- An **instance variable or method** definition that is **not preceded** with a **modifier** has ***package access***
  - Package access is also known as ***default* or *friendly access***
  - can be accessed *by name* inside the definition of any class in the same package but cannot be accessed outside the package

Display 7.9 **Access Modifiers**



- **Static members in a base class** are **inherited** by any of its **derived classes**
- The modifiers **public**, **private**, and **protected**, and package access have the **same meaning for static members** as they do **for instance variables and methods**

THE CLASS Object

is in the package **java.lang**

Every class is a descendent of the class ***Object***

Every object is of **type Object**

- **Methods** to be written with a **parameter of type Object**
- Like The **equals** and **toString** methods (should be overridden in derived classes)

The instanceof Operator

- checks if an object is of the type given as its second argument

Object instanceof ClassName

True or false, if **Object** is a **type of ClassName** or not

True if **Object** is of a **derived class of ClassName**

The getClass() Method

marked as **final**, inherited from **Object Class**

- An invocation of **getClass()** on an object **returns** a **representation** *only* of the **class** that was used with **new** to create the object

Application: To check if two objects represents the exact same class

(object1.getClass() == object2.getClass())

getClass() is more exact than instanceof

The Right Way to Define equals

Not just overload

public boolean equals(Employee otherEmployee)

but Override

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;         //Type cast
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

TYPE CASTING

```
Object o = "str";
String str = (String)o;
```

Employee otherEmployee = (Employee)otherObject;

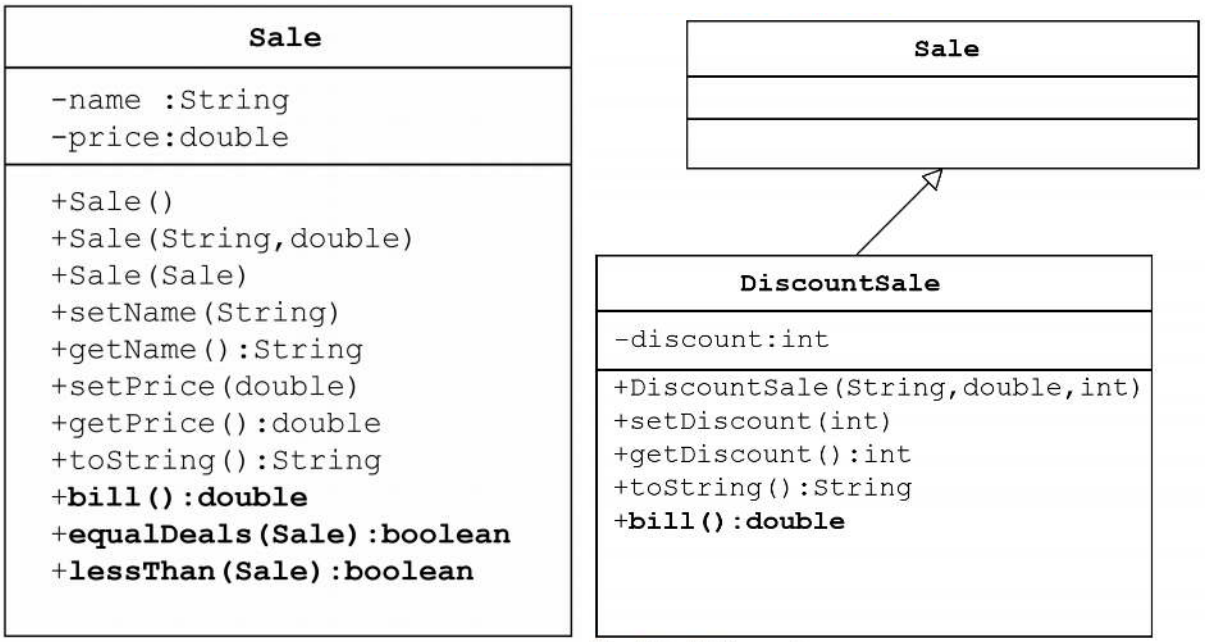
POLYMORPHISM

Through late binding or dynamic binding (in run time method invokations)

- Java uses late binding for all methods (except **private**, **final**, and **static methods**)

BaseClass Sale

DerivedClass DiscountSale



- The **Sale** class **lessThan** method
  - Note the **bill()** method invocations:

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
```

```
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

- The **Sale** class **bill()** method:

```
public double bill( )
{
  return price;
}
```

- The **DiscountSale** class **bill()** method:

```
public double bill( )
{
  double fraction = discount/100;
  return (1 - fraction) * getPrice( );
}
```

  - Decision of which **bill()** is invoked is made based on the *type of the variable naming the object*
  - The same with **toString()** method.

- Java uses static binding with **private**, **final**, and **static** methods
  - In the case of **private** and **final** methods, late binding would **serve no purpose**
  - However, in the case of a **static method** invoked using a calling object, it does make a difference

UPCASTING AND DOWNCASTING

- **UPCASTING**: When an object of a **derived class** is assigned to a variable of a **base class (ancestor)**

```
Sale saleVariable;                                          //Base class
DiscountSale discountVariable = new DiscountSale("paint", 15,10);  //Derived class
saleVariable = discountVariable;                           //Upcasting
System.out.println(saleVariable.toString());
```

toString above uses the definition given in the **DiscountSale class**

- **DOWNCASTING**: When an object of a **ancestor class** is assigned to a variable of a **derived class**
- has to be done very carefully
- In many cases it doesn't make sense, or is illegal:

```
discountVariable = (DiscountSale)saleVariable;        //will produce
                                                      //run-time error
discountVariable = saleVariable                  //will produce
                                                 //compiler error
```

But Sometimes it is necessary:

- i.e. inside the **equals** method for a class:

```
Sale otherSale = (Sale)otherObject;              //downcasting
```

Checking to See if Downcasting is Legitimate

object* instanceof *ClassName

ABSTRACT CLASSES

- An abstract method **has a heading, but no method body**
- The body of the method **is defined in the derived classes**
- Add the modifier **abstract**
- It **cannot be private**
- It **has no method body**, and ends with a **semicolon** in place of its body

```
public abstract class Employee
{
     private instanceVariables;
     . . .
     public abstract double getPay();
     public abstract void doIt(int count);
  . . .
}
```

- A class that has **at least one abstract method** is called an ***abstract class***
- A class that has **no abstract methods** is called a ***concrete class***

- You Cannot Create Instances of an Abstract Class
- An **abstract class constructor cannot** be used to **create an object** of the **abstract** class
- However, a **derived class constructor** will include **an invocation of the abstract class constructor** in the form of **super**
- It is perfectly fine to **have a parameter of an abstract class type**
- This makes it possible to **plug in** an object of any of its **descendent** classes
- It is also fine to **use a variable of an abstract class type**, as long is **it names objects of its concrete descendent** classes only

INTERFACES

- An *interface* is something like an **extreme case** of an **abstract class**
  - However, *an interface* ***is not a class***
  - *It is a* ***type*** *that* ***can be satisfied*** *by* ***any class*** *that* ***implements the interface***
  - **Java's way of approximating multiple inheritance, because Some languages allow one class to be derived from two or more different base classes**

- The syntax for defining an interface is similar to that of defining a class
  - Except the word **interface** is used in place of **class**
- An interface specifies a **set of methods** that **any class that implements** the interface **must have**
  - It contains **method headings** and **constant definitions** only
  - It contains *no instance variables* nor *any complete method definitions*

INTERFACE CONTAINS:

CONSTANTS (understood: public static final)	NO INSTANCE VARIABLES
METHOD HEADINGS (public)	NO COMPLETE DEFINITIONS

- Because an interface is a type, a **method may be written** with a **parameter of an interface type**
  - That parameter will accept as an argument in any class that implements the interface

Display 13.1 The Ordered Interface

```
1  public interface Ordered
2  {
3      public boolean precedes(Object other);

4      /**
5       For objects of the class o1 and o2,
6       o1.follows(o2) == o2.preceded(o1).
7      */
8      public boolean follows(Object other);
9  }
```

Do not forget the semicolons at the end of the method headings.

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

IMPLEMENTING AN INTERFACE

CONCRETE CLASS

implements *Interface_Name*

implements *Interface_Name, Interface_Name2, Interface_Name3*

Display 13.2 Implementation of an Interface

```
1  public class OrderedHourlyEmployee
2           extends HourlyEmployee implements Ordered
3  {
4      public boolean precedes(Object other)
5      {
6          if (other == null)
7              return false;
8          else if (!(other instanceof HourlyEmployee))
9              return false;
10         else
11         {
12             OrderedHourlyEmployee otherOrderedHourlyEmployee =
13                                 (OrderedHourlyEmployee)other;
14              return (getPay() < otherOrderedHourlyEmployee.getPay());
15         }
16     }
```

Although `getClass` works better than `instanceof` for defining `equals`, `instanceof` works better here. However, either will do for the points being made here.

```
17      public boolean follows(Object other)
18      {
19          if (other == null)
20              return false;
21          else if (!(other instanceof OrderedHourlyEmployee))
22              return false;
23          else
24          {
25              OrderedHourlyEmployee otherOrderedHourlyEmployee =
26                                  (OrderedHourlyEmployee)other;
27              return (otherOrderedHourlyEmployee.precedes(this));
28          }
29      }
30  }
```

Abstract classes can implement interfaces, but a concrete class must give a complete definition of all interfaces methods.

Display 13.3 An Abstract Class Implementing an Interface

```
1   public abstract class MyAbstractClass implements Ordered
2   {
3       int number;
4       char grade;
5
6       public boolean precedes(Object other)
7       {
8           if (other == null)
9               return false;
10          else if (!(other instanceof HourlyEmployee))
11              return false;
12          else
13          {
14              MyAbstractClass otherOfMyAbstractClass =
15                                          (MyAbstractClass)other;
16              return (this.number < otherOfMyAbstractClass.number);
17          }
18      }

19      public abstract boolean follows(Object other);

20  }
```

DERIVED INTERFACES

- This is called *extending* the interface
- The derived interface must include the phrase

extends *BaseInterfaceName*

Display 13.4 **Extending an Interface**

```
1  public interface ShowablyOrdered extends Ordered
2  {
3      /**
4        Outputs an object of the class that precedes the calling object.
5      */
6      public void showOneWhoPrecedes();
7  }
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the ShowablyOrdered interface must have a definition for the method showOneWhoPrecedes and also have definitions for the methods precedes and follows given in the Ordered interface.

- In Java, a **class** can have **only one base class**
- In addition, a **class** may implement **any number of interfaces**

INCONSISTENT INTERFACES (CLASSES)

- When a class implements two interfaces:
  - One type of inconsistency will occur if the interfaces have constants with the same name, but with different values
  - Another type of inconsistency will occur if the interfaces contain methods with the same name but different return types

The Comparable interface

An algorithm for sorting Doubles, can sort any type, int, Strings, etc.

- The **Comparable** interface is in the **java.lang** package, and so is automatically available to any program
- It has only the following method heading that must be implemented:

public int compareTo(Object other);

- The method **compareTo** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter **other** is **not of the same type** as the class being defined, then a **ClassCastException** should be thrown

Double and **String** classes implement the **Comparable** interface

(No **double** (primitive type))

EXCEPTION HANDLING

- A Java method can signal when something went wrong
  - This is called ***throwing an exception***
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called ***handling the exception***

try-throw-catch Mechanism

```
. . . // method code

try
{
  . . .
   If (invalid input or data)
  throw new Exception(StringArgument);   //StringArgument ”bla bla bla”
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
}  . . .
```

try {}

A block with the code for the algorithm and **throw** for exceptions

throw

throw new Exception(*StringArgument*);

throw new *ExceptionClassName*(*PossiblySomeArguments*);

- In the above example, the object of class ***ExceptionClassName*** is created using a string (PossiblySomeArgs) as its argument
- This object, which is an argument to the **throw** operator, is the exception object thrown

- Instead of calling a method, a **throw** statement jumps to a **catch** block

catch {}

catch(ExceptionClassName e) {...}

- The catch block has only one parameter
- The exception object thrown is plugged in for the catch block parameter
- ***e*** is called the **catch block parameter**
- Catch the class of Exception

- Different types of exceptions can be caught by placing more than one **catch** block after a **try** block
- Any number of **catch** blocks can be included, but they must be placed in the correct order:
  - Catch the More Specific Exception First:

```
catch (NegativeNumberException e)      //derived of Exception Class
          { . . . }
catch (Exception e)
          { . . . }
```

getMessage() Method

- Every exception has a **String** instance variable that contains some message
  - This string typically identifies the reason for the exception

- In the previous example, **StringArgument** is an argument to the **Exception** constructor
- This is the string used for the value of the **string instance variable** of exception **e**
  - Therefore, the method call **e.getMessage()** returns this **string**

EXCEPTION CLASSES

- All predefined exception classes have the following properties:
  - There is a constructor that takes a **single argument** of type **String**
  - The class has an accessor method **getMessage** that can **recover** the **string given as an argument** to the constructor when the exception object was created
- All programmer-defined classes must be **derived** from the class **Exception**
  - Every exception class is a descendent class of the class **Exception**
  - Although the **Exception** class can be used directly in a class or program, it is most often used to **define a derived class**
- Numerous predefined exception classes are included in the standard packages that come with Java
  - For example:

    IOException

    NoSuchMethodException

    FileNotFoundException
  - Many exception classes must be imported in order to use them

```
import java.io.IOException;
```

Defining Exception Classes

- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

Display 9.3 A Programmer-Defined Exception Class

```
1  public class DivisionByZeroException extends Exception
2  {
3      public DivisionByZeroException()
4      {
5          super("Division by Zero!");
6      }

7      public DivisionByZeroException(String message)
8      {
9          super(message);
10     }
11 }
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

- **The two most important things about an exception object are its type (i.e., exception class) and the message it carries**

GUIDELINES

- Must be a **derived class** of an already **existing exception class**
- **At least two constructors** should be defined, sometimes more
  - A constructor that takes a string argument and begins with a call to **super**, which takes the string argument
  - A no-argument constructor that includes a call to **super** with a default string as the argument
- The exception class **should allow** for the fact that the method **getMessage** is inherited

Throwing an Exception in a Method

- Sometimes it makes sense to **throw an exception** in a method, but **not catch it in the same method**
- In this case, the method itself would not include **try** and **catch** blocks
    - However, it would have to include a ***throws*** *clause. Providing a warning in the heading.* ***Declaring the exception***, if doesn't the method ends immediately.

```
public void aMethod() throws AnException
public void aMethod() throws AnException, AnotherException
```

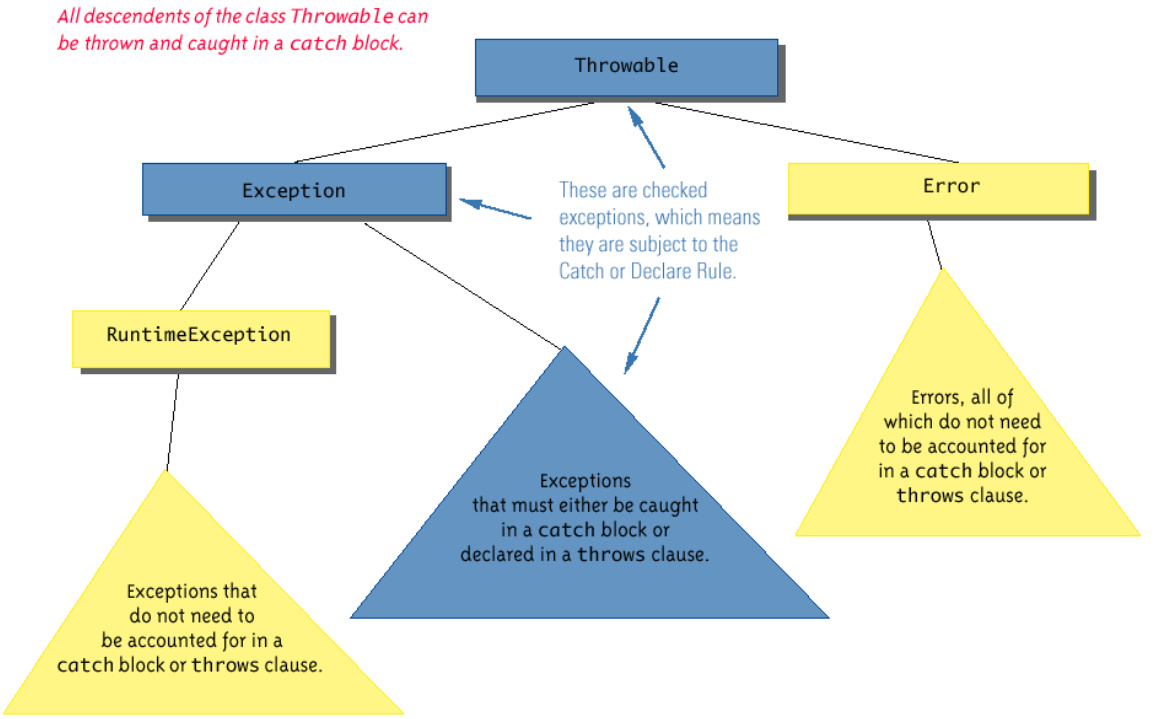
EXAMPLE:

```
public void someMethod() throws SomeException
{
  . . .
  throw new SomeException(SomeArgument);
  . . .
}
```

HANDLING THE **EXCEPTION** IN **OTHER METHOD** THAT **INVOKES THE SOME METHOD**

```
public void otherMethod()
{
  try
  {
    someMethod();
    . . .
  }
  catch (SomeException e)
  {
    CodeToHandleException
  }
  . . .
}
```

Display 9.10 **Hierarchy of Throwable Objects**



- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes **Throwable**, **Exception**, and all descendants of the class **Exception** are checked exceptions
- All other exceptions are *unchecked* exceptions -- **must be corrected**.
- The class **Error** and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

The throws Clause in Derived Classes

- When a method in a derived class is overridden, it should have **the same exception classes listed** in its **throws** clause that it had in the **base class** (Or it should have a subset of them)
- A derived class **may not add any exceptions** to the **throws** clause
  - But it **can delete some**

The finally block

```
public some myMethod(args){

try
{  . . .  }
catch(ExceptionClass1 e)
{  . . .  }
  . . .
catch(ExceptionClassN e)
{  . . .  }
finally
{
  CodeToBeExecutedInAllCases
}
```

1. The **try** block runs to the end, no exception is thrown, and the finally block is executed
2. An exception is thrown in the **try** block, caught in one of the **catch** blocks, and the **finally** block is executed
3. An exception is thrown in the **try** block, there is no matching **catch** block in the method, the **finally** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

Exception Handling with the `Scanner` Class

- The `nextInt` method of the `Scanner` class can be used to read `int` values from the keyboard
- However, if a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input

The `InputMismatchException`

- The `InputMismatchException` is in the standard Java package `java.util`
  - A program that refers to it must use an `import` statement, such as the following:

```
import java.util.InputMismatchException;
```

- It is a descendent class of `RuntimeException`
  - Therefore, it is an unchecked exception and does not have to be caught in a `catch` block or declared in a `throws` clause
  - However, catching it in a `catch` block is allowed, and can sometimes be useful

- An **ArrayIndexOutOfBoundsException** is thrown whenever a program attempts to use an array index that is out of bounds
- A **NullPointerException** is thrown when a program attempts to send a message to **null**
- These are unchecked exceptions, like all other descendents of the class **RuntimeException** (There is no requirement to handle it) Fix the program

GENERICS

- Java allows class and method definitions that include **parameters for types**
- Such definitions are called ***generics***
  - Generic programming with a type parameter **enables code** to be written that **applies to any class**

ArrayList Class

same purpose as an ARRAY

- an **ArrayList** is an object that can **grow** and **shrink** while your program is running
- Has an **array** as a **private instance variable**
- It does not have the convenient **square bracket** notation
- The base type must be a **class type** or **interface type. (no primitive)**, but no problem because of Boxing and Unboxing.

```
import java.util.ArrayList
ArrayList<BaseType> aList = new ArrayList<BaseType>();
```

Compare with array:

```
double[] score= new double[5];
```

initial capacity

```
ArrayList<String> list = new ArrayList<String>(20);
```

METHODS

```
list.add("something");              //overloaded, adds in next empty index
                                    //another two-arguments version
                                    //for (index, element)
                                    // move-up the rest of elements
int howMany = list.size();          //how many elements already have

list.set(index, "something else"); //set an element (if already exist)
```

```
String thing = list.get(index);    //return index of the element
```

CONSTRUCTORS

Display 14.1 Some Methods in the Class ArrayList

CONSTRUCTORS

public ArrayList(class_type first, initialCapacity)

Creates an empty ArrayList with the specified base_type and initial capacity.

public ArrayList(class_type())

Creates an empty ArrayList with the specified base_type and an initial capacity of 10.

(continued)

METHODS

Display 14.1 **Some Methods in the Class** `ArrayList`

ARRAYLIKE METHODS

`public` *Base_Type* `set(` `int` `index,` *Base_Type* `newElement)`

Sets the element at the specified `index` to `newElement`. Returns the element previously at that position, but the method is often used as if it were a `void` method. If you draw an analogy between the `ArrayList` and an array `a`, this statement is analogous to setting `a[index]` to the value `newElement`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws an `IndexOutOfBoundsException` if the `index` is not in this range.

`public` *Base_Type* `get(int index)`

Returns the element at the specified index. This statement is analogous to returning `a[index]` for an array `a`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range.

(continued)

Display 14.1 Some Methods in the Class ArrayList

METHODS TO ADD ELEMENTS
`public boolean add(Base_Type newElement)` Adds the specified element to the end of the calling ArrayList and increases the ArrayList's size by one. The capacity of the ArrayList is increased if that is required. Returns true if the add was successful. (The return type is boolean, but the method is typically used as if it were a void method.)
`public void add(int index, Base_Type newElement)` Inserts newElement as an element in the calling ArrayList at the specified index. Each element in the ArrayList with an index greater or equal to index is shifted upward to have an index that is one greater than the value it had previously. The index must be a value greater than or equal to 0 and less than or equal to the current size of the ArrayList. Throws IndexOutOfBoundsException if the index is not in this range. Note that you can use this method to add an element after the last element. The capacity of the ArrayList is increased if that is required.

(continued)

Display 14.1 Some Methods in the Class ArrayList

METHODS TO REMOVE ELEMENTS
`public Base_Type remove(int index)` Deletes and returns the element at the specified index. Each element in the ArrayList with an index greater than index is decreased to have an index that is one less than the value it had previously. The index must be a value greater than or equal to 0 and less than the current size of the ArrayList. Throws IndexOutOfBoundsException if the index is not in this range. Often used as if it were a void method.

(continued)

move down the rest o elements and decrease the current ArrayList size

Display 14.1 Some Methods in the Class ArrayList

`protected void removeRange(int fromIndex, int toIndex)` Deletes all the element with indices i such that fromIndex ≤ i < toIndex. Element with indices greater than or equal to toIndex are decreased appropriately.
`public boolean remove(Object theElement)` Removes one occurrence of theElement from the calling ArrayList. If theElement is found in the ArrayList, then each element in the ArrayList with an index greater than the removed element's index is decreased to have an index that is one less than the value it had previously. Returns true if theElement was found (and removed). Returns false if theElement was not found in the calling ArrayList.
`public void clear()` Removes all elements from the calling ArrayList and sets the ArrayList's size to zero.

(continued)

Display 14.1 Some Methods in the Class ArrayList

SOME METHODS

```
public boolean contains(Object target)
```

Returns true if the calling ArrayList contains target; otherwise, returns false. Uses the method equals of the object target to test for equality with any element in the calling ArrayList.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to target. Uses the method equals of the object target to test for equality. Returns −1 if target is not found.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to target. Uses the method equals of the object target to test for equality. Returns −1 if target is not found.

(continued)

Display 14.1 Some Methods in the Class ArrayList

MEMORY MANAGEMENT (SIZE AND CAPACITY)

```
public boolean isEmpty()
```

Returns true if the calling ArrayList is empty (that is, has size 0); otherwise, returns false.

(continued)

Display 14.1 Some Methods in the Class ArrayList

```
public int size()
```

Returns the number of elements in the calling ArrayList.

```
public void ensureCapacity(int newCapacity)
```

Increases the capacity of the calling ArrayList, if necessary, in order to ensure that the ArrayList can hold at least newCapacity elements. Using ensureCapacity can sometimes increase efficiency, but it is not needed for any other reason.

```
public void trimToSize()
```

Trims the capacity of the calling ArrayList to the ArrayList's current size. This method is used to save storage space.

(continued)

SAVE SPACE

Display 14.1 Some Methods in the Class ArrayList

OVERRIDES

```
public Object[] toArray()
```

Returns an array containing all the elements on the list. Preserves the order of the elements.

```
public Type[] toArray(Type[] a)
```

Returns an array containing all the elements on the list. Preserves the order of the elements. Type can be any class type. If the list will fit in a, the elements are copied to a and a is returned. Any elements of a not needed for list elements are set to null. If the list will not fit in a, a new array is created.

(As we will discuss in Section 14.2, the correct Java syntax for this method heading is

```
public <Type> Type[] toArray(Type[] a)
```

However, at this point we have not yet explained this kind of type parameter syntax.)

(continued)

Display 14.1 Some Methods in the Class ArrayList

```
public Object clone()
```

Returns a shallow copy of the calling ArrayList. Warning: The clone is not an independent copy. Subsequent changes to the clone may affect the calling object and vice versa. (See Chapter 5 for a discussion of shallow copy.)

(continued) **MUST BE ANOTHER FORM TO MAKE A DEEP COPY**

Display 14.1 Some Methods in the Class ArrayList

OVERRIDES

```
public boolean equals(Object other)
```

If other is another ArrayList of any base type, then equals returns true if and only if both ArrayLists are of the same size and contain the same list of elements in the same order. (In fact, if other is any kind of list then equals returns true if and only if both the calling ArrayList and other are of the same size and contain the same list of elements in the same order. Lists are discussed in Chapter 16.)

- ArrayList class **implements a number of interfaces**, and **inherits methods from various ancestor classes**

- These **interfaces and ancestor** classes specify that certain parameters have **type Object**

A for-each Loop can be Used with an ArrayList

Display 4.2 A for-each Loop Used with an ArrayList

```
1  import java.util.ArrayList;
2  import java.util.Scanner;

3  public class ArrayListDemo
4  {
5     public static void main(String[] args)
6     {
7        ArrayList<String> toDoList = new ArrayList<String>(20);
8        System.out.println(
9                        "Enter list entries, when prompted.");
10       boolean done = false;
11       String next = null;
12       String answer;
13       Scanner keyboard = new Scanner(System.in);
```

(continued)

Display 4.2 A for-each Loop Used with an ArrayList

```
14          while (! done)
15          {
16              System.out.println("Input an entry:");
17              next = keyboard.nextLine();
18              toDoList.add(next);

19              System.out.print("More items for the list? ");
20              answer = keyboard.nextLine();
21              if (!(answer.equalsIgnoreCase("yes")))
22                    done = true;
23          }

24          System.out.println("The list contains:");
25          for (String entry : toDoList)
26              System.out.println(entry);
27      }
28  }
29
```

(continued)

The Vector Class

- **Vector Class behaves almost exactly the same as the class ArrayList**

Parameterized Classes and Generics

- Starting with version 5.0, Java allows **class definitions** with **parameters for types**
  - These classes that have type parameters are called ***parameterized class*** **or** ***generic definitions***, or, simply, ***generics***
- Classes and methods can have a **type parameter**
  - A type parameter can have **any reference type** (i.e., any class type) plugged in for the type parameter
  - When a specific type is plugged in, **this produces a specific class type or method**

GENERIC CLASS

Display 13.4 A Class Definition with a Type Parameter

```
1  public class Sample<T>
2  {
3      private T data;

4      public void setData(T newData)
5      {
6          data = newData;                    T is a parameter for a type.
7      }

8      public T getData()
9      {
10         return data;
11     }
12 }
```

INSTANTIATION in other Class

```
Sample<String> object = new Sample<String>();
```

BUT

```
Pair<String>[] a = new Pair<String>[10];          //ILLEGAL
Pair<String[10]> a = new Pair<String[10]>();      //CORRECT!!?   ME
```

CONSTRUCTORS NO Pair<T>

Display 14.5 A Generic Ordered Pair Class

```
1  public class Pair<T>
2  {
3      private T first;
4      private T second;

5      public Pair()
6      {
7          first = null;
8          second = null;
9      }

10     public Pair(T firstItem, T secondItem)
11     {
12         first = firstItem;
13         second = secondItem;
14     }
```

Constructor headings do not include the type parameter in angular brackets

(continued)

METHODS

Display 14.5 A Generic Ordered Pair Class

```
15     public void setFirst(T newFirst)
16     {
17         first = newFirst;
18     }

19     public void setSecond(T newSecond)
20     {
21         second = newSecond;
22     }

23     public T getFirst()
24     {
25         return first;
26     }
```

(continued)

Display 14.5 A Generic Ordered Pair Class

```
27      public T getSecond()
28      {
29          return second;
30      }

31      public String toString()
32      {
33          return ( "first: " + first.toString() + "\n"
34                   + "second: " + second.toString() );
35      }
36
```

(continued)

Display 14.5 A Generic Ordered Pair Class

```
37      public boolean equals(Object otherObject)
38      {
39          if (otherObject == null)
40              return false;
41          else if (getClass() != otherObject.getClass())
42              return false;
43          else
44          {
45              PairNew otherPair = (PairNew)otherObject;
46              return (first.equals(otherPair.first)
47                  && second.equals(otherPair.second));
48          }
49      }
50  }
```

USING THE GENERIC CLASS instantiation

Display 14.6 **Using Our Ordered Pair Class**

```
1   import java.util.Scanner;

2   public class GenericPairDemo
3   {
4      public static void main(String[] args)
5      {
6           Pair<String> secretPair =
7                new Pair<String>("Happy", "Day");
8
9           Scanner keyboard = new Scanner(System.in);
10          System.out.println("Enter two words:");
11          String word1 = keyboard.next();
12          String word2 = keyboard.next();
13          Pair<String> inputPair =
14              new Pair<String>(word1, word2);
```

(continued)

T object = new T(); //ILEGAL

```
T[] a = new T[10];                    //ILEGAL
```

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```
1  import java.util.Scanner;

2  public class GenericPairDemo2
3  {
4      public static void main(String[] args)
5      {
6          Pair<Integer> secretPair =
7              new Pair<Integer>(42, 24);
8
9          Scanner keyboard = new Scanner(System.in);
10         System.out.println("Enter two numbers:");
11         int n1 = keyboard.nextInt();
12         int n2 = keyboard.nextInt();
13         Pair<Integer> inputPair =
14             new Pair<Integer>(n1, n2);
```

Automatic boxing allows you to use an int argument for an Integer parameter.

(continued)

MULTIPLE TYPE PARAMETERS Name<T1, T2, T3>

Display 14.8 **Multiple Type Parameters**

```
1  public class TwoTypePair<T1, T2>
2  {
3      private T1 first;
4      private T2 second;

5      public TwoTypePair()
6      {
7          first = null;
8          second = null;
9      }

10     public TwoTypePair(T1 firstItem, T2 secondItem)
11     {
12         first = firstItem;
13         second = secondItem;
14     }
```

(continued)

Display 14.8 Multiple Type Parameters

```
15      public void setFirst(T1 newFirst)
16      {
17          first = newFirst;
18      }

19      public void setSecond(T2 newSecond)
20      {
21          second = newSecond;
22      }

23      public T1 getFirst()
24      {
25          return first;
26      }
```

(continued)

Display 14.8 Multiple Type Parameters

```
27      public T2 getSecond()
28      {
29          return second;
30      }

31      public String toString()
32      {
33          return ( "first: " + first.toString() + "\n"
34                   + "second: " + second.toString() );
35      }
36
```

(continued)

. . . And the equals method . . .}

USING

Display 14.9 **Using a Generic Class with Two Type Parameters**

```
1  import java.util.Scanner;

2  public class TwoTypePairDemo
3  {
4     public static void main(String[] args)
5     {
6          TwoTypePair<String, Integer> rating =
7               new TwoTypePair<String, Integer>("The Car Guys", 8);

8          Scanner keyboard = new Scanner(System.in);
9          System.out.println(
10                    "Our current rating for " + rating.getFirst());
11         System.out.println(" is " + rating.getSecond());

12         System.out.println("How would you rate them?");
13         int score = keyboard.nextInt();
14         rating.setSecond(score);
```

(continued)

- **It is not permitted to create a generic class with Exception, Error, Throwable, or any descendent class of Throwable**

public class GEx<T> extends Exception **COMPILER ERROR**

RESTRICT THE TYPE PARAMETER <T>

For example:

An interface

public class RClass<T extends Comparable>

- Any attempt to **plug in a type for T** which does **not implement the Comparable interface** will result in a compiler error message

A Class

public class ExClass<T extends Class1>

- **Only descendants of Class1 are allowed**

Multipe ways

public class Two<T1 extends Class1, T2 extends Class2 & Comparable>

Display 14.10 A Bounded Type Parameter

```
1  public class Pair<T extends Comparable>
2  {
3      private T first;
4      private T second;

5      public T max()
6      {
7          if (first.compareTo(second) < 0)
8              return first;
9          else
10             return second;
11     }

    <All the constructors and methods given in Display 14.3
          are also included as part of this generic class definition>

12 }
```

GENERIC INTERFACES (THE SAME AS WITH GENERIC CLASSES)