



Programming and Software Development

COMP90041

Lecture 5

Classes and Methods II

NOTE: Some of the Material in these slides are adopted from

- * Lectures Notes prepared by Dr. Peter Schachte and
- * the Textbook resources

- **Class definitions**
 - **Class structure**
 - **Variables**
 - **Methods**
- **Encapsulation**
 - **Access modifiers (e.g., public vs private)**
 - **Accessor and mutator methods**
- **Overloading**
- **Constructors**

Review: Week 4

- **Static methods and static variables**
 - The Math class and wrapper classes
 - Automatic boxing and unboxing mechanism
- **References and class parameters**
 - Variables and Memory
 - Using and misusing references

Outline

- **Static methods and static variables**
 - **The Math class and wrapper classes**
 - **Automatic boxing and unboxing mechanism**
- **References and class parameters**
 - **Variables and Memory**
 - **Using and misusing references**

Outline

- A *static method* is one that can be used without a calling object
- A static method still belongs to a class, and its definition is given inside the class definition
- When a static method is defined, the keyword **static** is placed in the method header

```
public static returnType myMethod(parameters)  
{ ... }
```

- Static methods are invoked using the **class name** in place of a calling object

```
returnValue = MyClass.myMethod(arguments);
```

Static Methods

Display 5.1 Static Methods (part 1 of 2)

```
1  /**
2  Class with static methods for circles and spheres.
3  */
4  public class RoundStuff
5  {
6      public static final double PI = 3.14159;
7
8      /**
9       Return the area of a circle of the given radius.
10    */
11    public static double area(double radius)
12    {
13        return(PI*radius*radius);
14    }
15
16    /**
17     Return the volume of a sphere of the given radius.
18    */
19    public static double volume(double radius)
20    {
21        return( (4.0/3.0)*PI*radius*radius*radius);
22    }
23 }
```

*This is the file
RoundStuff.java.*



```
1 import java.util.Scanner;  
2 public class RoundStuffDemo  
3 {  
4     public static void main(String[] args)  
5     {  
6         Scanner keyboard = new Scanner(System.in);  
7         System.out.println("Enter radius:");  
8         double radius = keyboard.nextDouble();  
9  
9         System.out.println("A circle of radius"  
10                         + radius + "inches");  
11         System.out.println("has an area of " +  
12                         RoundStuff.area(radius) + " square inches.");  
13         System.out.println("A sphere of radius"  
14                         + radius + "inches");  
15         System.out.println("has an volume of " +  
16                         RoundStuff.volume(radius) + "cubic inches.");  
17     }  
18 }
```

This is the file

RoundStuffDemo.java.

Static methods are invoked using the class name in place of a calling object

Example - RoundStuffDemo

- A static method cannot refer to an instance variable of the class, and it cannot invoke a non-static method of the class
 - A static method has no **this**, so it cannot use an instance variable or method that has an implicit or explicit **this** for a calling object
 - A static method can invoke another static method, however

Pitfall: Invoking a Non-static Method Within a Static Method

- Although the main method is often by itself in a class separate from the other classes of a program, it can also be contained within a regular class definition
 - In this way the class in which it is contained can be used to create objects in other classes, or it can be run as a program
 - A main method so included in a regular class definition is especially useful when it contains diagnostic code for the class

Tip: You Can Put a `main` in any Class

Display 5.3 Another Class with a main Added

```
1 import java.util.Scanner;  
2  
3 /**  
4  * Class for a temperature (expressed in degrees Celsius).  
5  */  
6 public class Temperature  
7 {  
8     private double degrees; //Celsius  
9  
10    public Temperature()  
11    {  
12        degrees = 0;  
13    }  
14  
15    public Temperature(double initialDegrees)  
16    {  
17        degrees = initialDegrees;  
18    }  
19  
20    public void setDegrees(double newDegrees)  
21    {  
22        degrees = newDegrees;  
23    }
```

Note that this class has a main method and both static and nonstatic methods.

(continued)

Another Class with a main Added (Part 1 of 4)

Display 5.3 Another Class with a main Added

```
20     public double getDegrees()
21     {
22         return degrees;
23     }
24
25     public String toString()
26     {
27         return (degrees + " C");
28     }
29
30     public boolean equals(Temperature otherTemperature)
31     {
32         return (degrees == otherTemperature.degrees);
33     }
```

(continued)

**Another Class with a main Added
(Part 2 of 4)**

Display 5.3 Another Class with a main Added

```
33     /**
34      Returns number of Celsius degrees equal to
35      degreesF Fahrenheit degrees.
36  */
37  public static double toCelsius(double degreesF)
38  {
39
40      return 5*(degreesF - 32)/9;
41  }
42
43  public static void main(String[] args)
44  {
45      double degreesF, degreesC;
46
47      Scanner keyboard = new Scanner(System.in);
48      System.out.println("Enter degrees Fahrenheit:");
49      degreesF = keyboard.nextDouble();
50
51      degreesC = toCelsius(degreesF);
```

Because this is in the definition of the class Temperature, this is equivalent to Temperature.toCelsius(degreesF).

(continued)

Another Class with a main Added (Part 3 of 4)

Display 5.3 Another Class with a main Added

```
52     Temperature temperatureObject = new Temperature(degreesC);  
53     System.out.println("Equivalent Celsius temperature is "  
54             + temperatureObject.toString());  
55 }  
56 }
```

Because main is a static method, toString must have a specified calling object like temperatureObject.

SAMPLE DIALOGUE

Enter degrees Fahrenheit:

212

Equivalent Celsius temperature is 100.0 C

Another Class with a main Added (Part 4 of 4)

- 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
- Although a static method cannot access an instance variable, a static method can access a static variable
- A static variable is declared like an instance variable, with the addition of the modifier **static**

```
private static int myStaticVariable;
```

Static Variables

- Static variables can be declared and initialized at the same time

```
private static int myStaticVariable = 0;
```

- If not explicitly initialized, a static variable will be automatically initialized to a default value
 - **boolean** static variables are initialized to **false**
 - Other primitive types static variables are initialized to the zero of their type
 - Class type static variables are initialized to **null**
- It is always preferable to explicitly initialize static variables rather than rely on the default initialization
 - E.g. Human.java > populationCount

Static Variables : Initialization

```
public class Human
{
    private String name;
    private static int populationCount = 0;
    public Human(String aName)
    {
        name = aName;
        populationCount++;
    }
    public static int getPopulation()
    {
        return populationCount;
    }
}
```

Human.java

```
public class HumanDemo
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 10; i++)
        {
            Human newHuman = new Human("Person " + i);
            System.out.println("Current population: "
                + Human.getPopulation());
        }
    }
}
```

HumanDemo.java

Example - Human.java

- A static variable should always be defined private, unless it is also a defined constant
 - The value of a static defined constant cannot be altered, therefore it is safe to make it **public**
 - In addition to **static**, the declaration for a static defined constant must include the modifier **final**, which indicates that its value cannot be changed

```
public static final double PI = 3.14159;
```

- When referring to such a defined constant outside its class, use the name of its class in place of a calling object

```
int year = RoundStuff.PI;
```

Public Static Variables : Constants

- The **Math** class provides a number of standard mathematical methods
 - It is found in the **java.lang** package, so it does not require an **import** statement
 - **All** of its methods and data are **static**, therefore they are invoked with the class name **Math** instead of a calling object
 - The **Math** class has two predefined constants, **E** (e , the base of the natural logarithm system) and **PI** (π , 3.1415 ...)

```
area = Math.PI * radius * radius;
```

The Math Class

Display 5.6 Some Methods in the Class Math

The Math class is in the `java.lang` package, so it requires no `import` statement.

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

Returns base to the power exponent.

EXAMPLE

`Math.pow(2.0, 3.0)` returns 8.0.

(continued)

Some Methods in the Class Math (Part 1 of 5)

Display 5.6 Some Methods in the Class Math

```
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.

(continued)

Part 2 of 3

Display 5.6 Some Methods in the Class Math

```
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(3.6) returns 4.

(continued)

Some Methods in the Class Math (Part 3 of 5)

Display 5.6 Some Methods in the Class Math

```
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.

(continued)

**Some Methods in the Class Math
(Part 4 of 5)**

Display 5.6 Some Methods in the Class Math

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the argument.

EXAMPLE

Math.floor(3.2) and Math.floor(3.9) both return 3.0.

```
public static double sqrt(double argument)
```

Returns the square root of its argument.

EXAMPLE

Math.sqrt(4) returns 2.0.

Some Methods in the Class Math (Part 5 of 5)

- *Wrapper classes* provide a class type corresponding to each of the primitive types
 - This makes it possible to have class types that behave somewhat like primitive types
 - The wrapper classes for the primitive types **boolean**, **byte**, **short**, **long**, **float**, **double**, and **char** are (in order) **Boolean**, **Byte**, **Short**, **Long**, **Float**, **Double**, and **Character**
- Wrapper classes also contain a number of useful predefined constants and static methods

Wrapper Classes

- *Boxing*: the process of going from a value of a primitive type to an object of its wrapper class
 - To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
 - The new object will contain an instance variable that stores a copy of the primitive value
 - Unlike most other classes, a wrapper class does not have a no-argument constructor

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

Wrapper Classes

- *Unboxing*: the process of going from an object of a wrapper class to the corresponding value of a primitive type
 - The methods for converting an object from the wrapper classes **Boolean**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, and **Character** to their corresponding primitive type are (in order) **booleanValue**, **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, and **charValue**
 - None of these methods take an argument

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

Wrapper Classes

- Starting with version 5.0, Java can automatically do boxing and unboxing
- Instead of creating a wrapper class object using the **new** operation (as shown before), it can be done as an automatic type cast:

```
Integer integerObject = 42;
```

- Instead of having to invoke the appropriate method (such as **intValue**, **doubleValue**, **charValue**, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
int i = integerObject;
```

Automatic Boxing and Unboxing

- Wrapper classes include useful constants that provide the largest and smallest values for any of the primitive number types
 - For example, `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE`, etc.
- The `Boolean` class has names for two constants of type `Boolean`
 - `Boolean.TRUE` and `Boolean.FALSE` are the Boolean objects that correspond to the values `true` and `false` of the primitive type `boolean`

Constants and Static Methods in Wrapper Classes

- Wrapper classes have static methods that convert a correctly formed string representation of a number to the number of a given type
 - The methods `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, and `Double.parseDouble` do this for the primitive types (in order) `int`, `long`, `float`, and `double`
- Wrapper classes also have static methods that convert from a numeric value to a string representation of the value
 - For example, the expression
`Double.toString(123.99);`
returns the string value `"123.99"`
- The `Character` class contains a number of static methods that are useful for string processing

Constants and Static Methods in Wrapper Classes

Display 5.8 Some Methods in the Class Character

The class Character is in the `java.lang` package, so it requires no `import` statement.

```
public static char toUpperCase(char argument)
```

Returns the uppercase version of its argument. If the argument is not a letter, it is returned unchanged.

EXAMPLE

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return 'A'.

```
public static char toLowerCase(char argument)
```

Returns the lowercase version of its argument. If the argument is not a letter, it is returned unchanged.

EXAMPLE

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return 'a'.

```
public static boolean isUpperCase(char argument)
```

Returns `true` if its argument is an uppercase letter; otherwise returns `false`.

EXAMPLE

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

acter
(continued)
Part 1 of 3)

Display 5.8 Some Methods in the Class Character

```
public static boolean isLowerCase(char argument)
```

Returns true if its argument is a lowercase letter; otherwise returns false.

EXAMPLE

Character.isLowerCase('a') returns true. Character.isLowerCase('A') and Character.isLowerCase('%') both return false.

```
public static boolean isWhitespace(char argument)
```

Returns true if its argument is a whitespace character; otherwise returns false. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character ('\t'), and the line break character ('\n').

EXAMPLE

Character.isWhitespace(' ') returns true. Character.isWhitespace('A') returns false.

(continued)

Some Methods in the Class Character (Part 2 of 3)

Display 5.8 Some Methods in the Class Character

```
public static boolean isLetter(char argument)
```

Returns true if its argument is a letter; otherwise returns false.

EXAMPLE

Character.isLetter('A') returns true. Character.isLetter('%') and Character.isLetter('5') both return false.

```
public static boolean isDigit(char argument)
```

Returns true if its argument is a digit; otherwise returns false.

EXAMPLE

Character.isDigit('5') returns true. Character.isDigit('A') and Character.isDigit('%') both return false.

```
public static boolean isLetterOrDigit(char argument)
```

Returns true if its argument is a letter or a digit; otherwise returns false.

EXAMPLE

Character.isLetterOrDigit('A') and Character.isLetterOrDigit('5') both return true. Character.isLetterOrDigit('&') returns false.

```
1 import java.util.Scanner;  
2 /**  
3 Illustrate the use of a static method from the class Character.  
4 */  
5  
6 public class StringProcessor  
7 {  
8     public static void main (String[] args)  
9     {  
10         System.out.println("Enter a one line sentence:");  
11         Scanner keyboard = new Scanner(System.in);  
12         String sentence = keyboard.nextLine();  
13  
14         sentence = sentence.toLowerCase();  
15         char firstCharacter = sentence.charAt(0);  
16         sentence = Character.toUpperCase(firstCharacter)  
17             + sentence.substring(1);  
18  
19         System.out.println("The revised sentence is:");  
20         System.out.println(sentence);  
21     }  
22 }
```

```
Enter a one line sentence:  
is you is OR is you ain't my BABY?  
The revised sentence is:  
Is you is or is you ain't my baby?
```

ample

- **Static methods and static variables**
 - The Math class and wrapper classes
 - Automatic boxing and unboxing mechanism
- **References and class parameters**
 - **Variables and Memory**
 - **Using and misusing references**

Outline

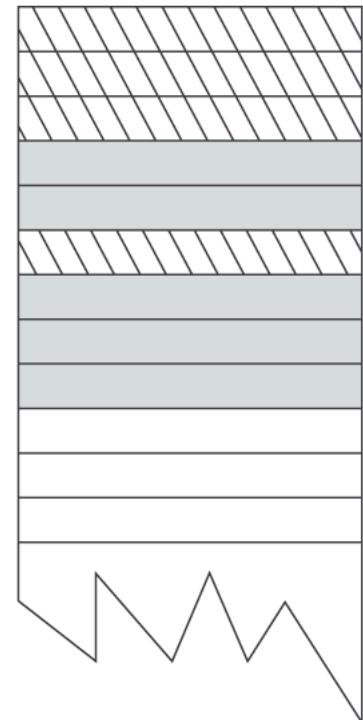
- A computer has two forms of memory
- *Secondary memory* is used to hold files for "permanent" storage
- *Main memory (a.k.a Primary memory)* is used by a computer when it is running a program
 - Values stored in a program's variables are kept in main memory

Variables and Memory

- Main memory consists of a long list of numbered locations called **bytes**
 - Each byte contains eight *bits*: eight 0 or 1 digits
- The number that identifies a byte is called its **address**
 - A data item can be stored in one (or more) of these bytes
 - The address of the byte is used to find the data item when needed

Main Memory

byte 0
byte 1
byte 2
byte 3
byte 4
byte 5
byte 6
byte 7
byte 8



Variables and Memory

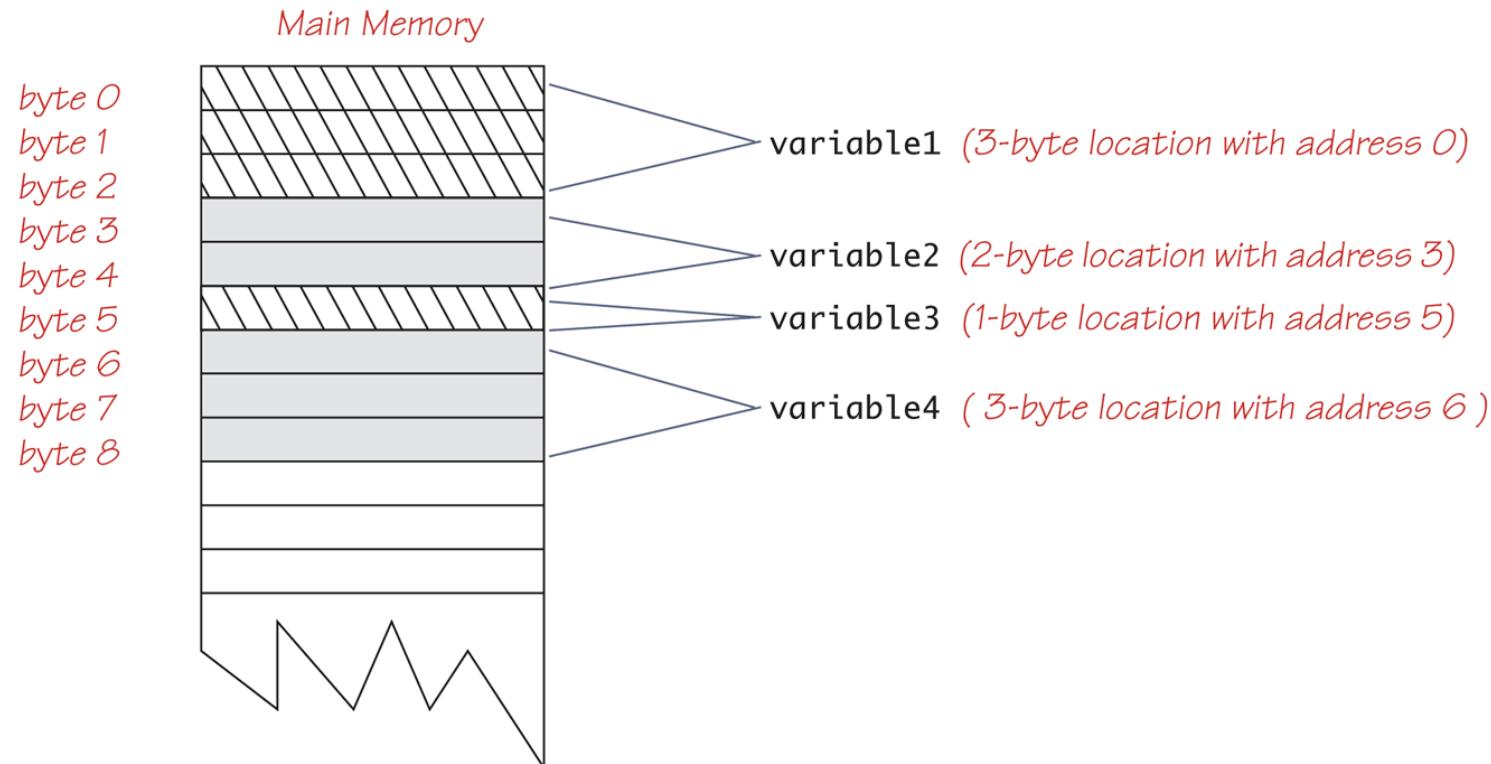
- 1Byte = 8 bits
- 1KB = $2^{10} = 1024 \sim 10^3$
- 1MB = $2^{20} = 1K \times 1K \sim 10^6$
- 1GB = $2^{30} = 1K \times 1M \sim 10^9$
- $2^{32}B = 4 \times 2^{30} = 4GB$

Conversions

- Values of most data types require more than one byte of storage
 - Several adjacent bytes are then used to hold the data item
 - The entire chunk of memory that holds the data is called its ***memory location***
 - The address of the first byte of this memory location is used as the address for the data item
- A computer's main memory can be thought of as a long list of memory locations of *varying sizes*

Variables and Memory

Display 5.10 Variables in Memory



Variables in Memory

- Every variable is implemented as a **location** in computer memory
- When the variable is a primitive type, the value of the variable is stored in the memory location *assigned to the variable*
 - Each primitive type always require the *same amount* of memory to store its values

References

- When the variable is a class type, only the memory address (or *reference*) where its object is located is stored in the memory location assigned to the variable
 - The object named by the variable is stored in some other location in memory
 - Like primitives, the value of a class variable is a fixed size
 - Unlike primitives, the value of a class variable is a memory address or reference
 - The object, whose address is stored in the variable, ***can be of any size***

References

- Two reference variables can contain the same reference, and therefore name the same object
 - The assignment operator sets the reference (memory address) of one class type variable equal to that of another
 - Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object

variable2 = variable1;

References

Display 5.12 Class Type Variables Store a Reference

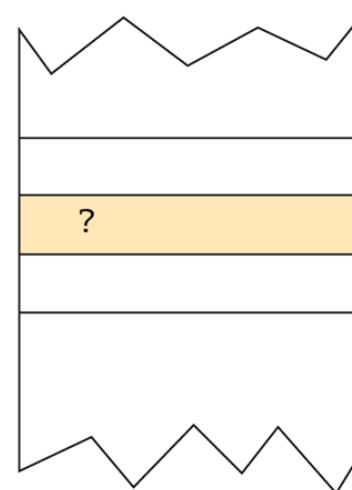
```
public class ToyClass
{
    private String name;
    private int number;
```

The complete definition of the class ToyClass is given in Display 5.11.

```
ToyClass sampleVariable;
```

Creates the variable sampleVariable in memory but assigns it no value.

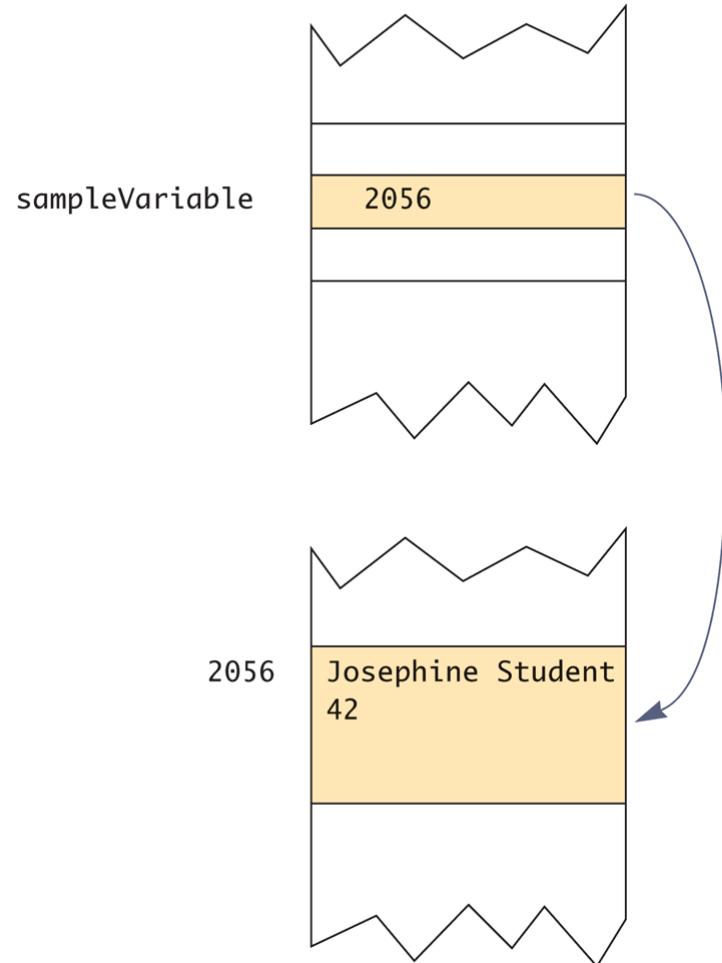
sampleVariable



```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

Creates an object, places the object someplace in memory, and then places the address of the object in the variable sampleVariable. We do not know what the address of the object is, but let's assume it is 2056. The exact number does not matter.

(continued)

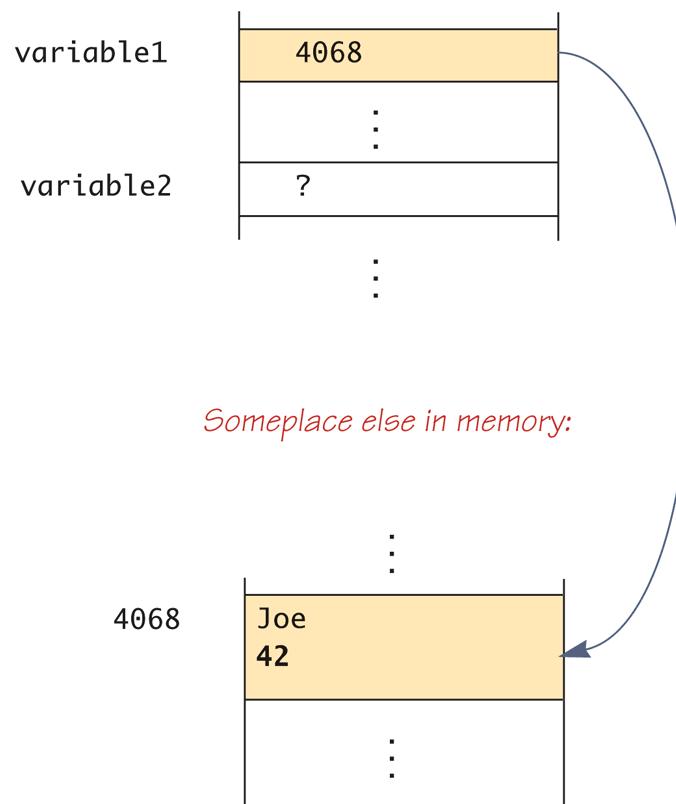
Display 5.12 Class Type Variables Store a Reference

For emphasis, we made the arrow point to the memory location referenced.

2 of 2)

Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);  
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. Let's say it is 4068. The exact number does not matter.

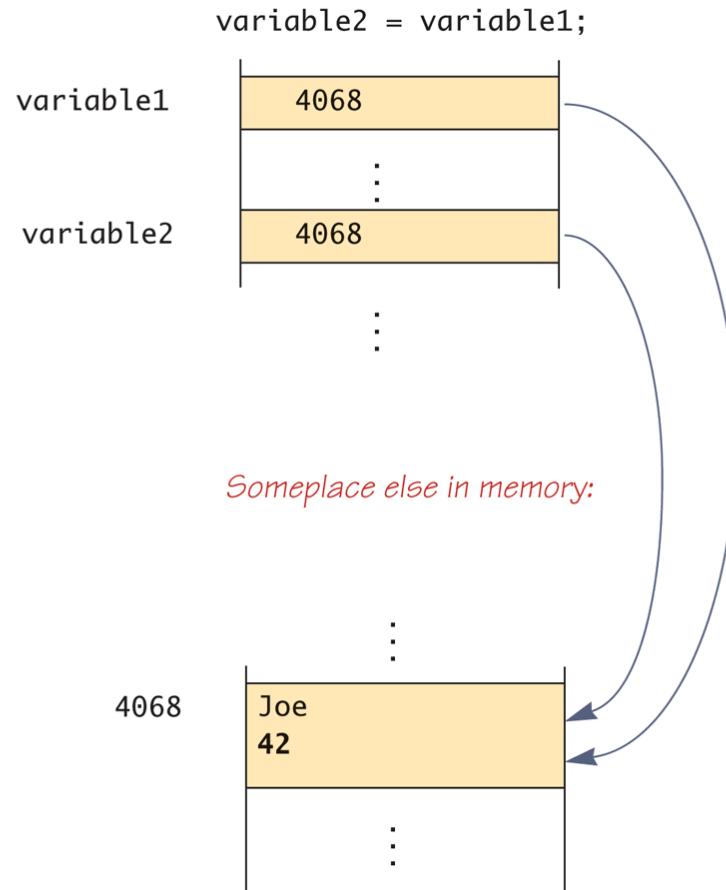
Note that you can think of

`new ToyClass("Joe", 42)`

as returning a reference.

(continued)

Part 1 of
3)

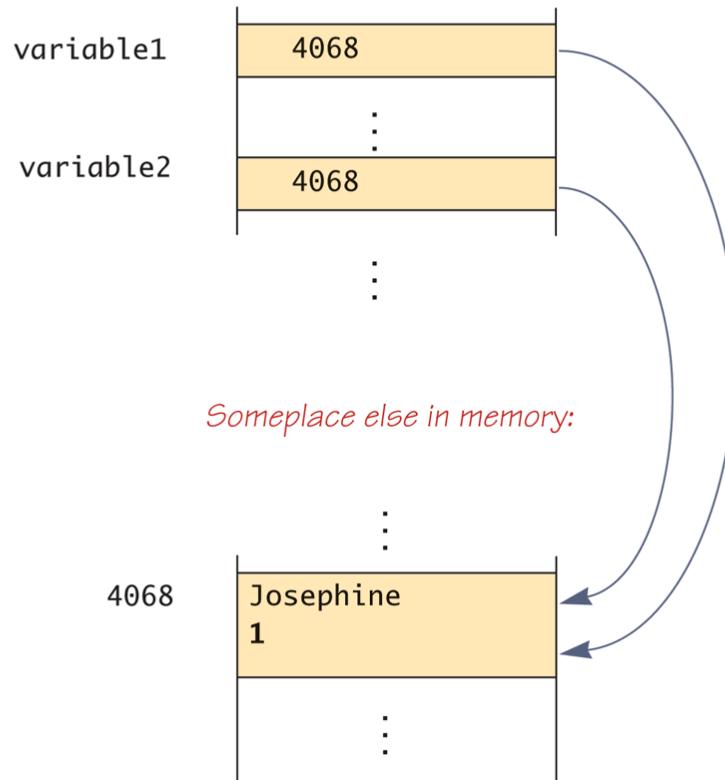
Display 5.13 Assignment Operator with Class Type Variables

(continued)

**(Part 2 of
3)**

Display 5.13 Assignment Operator with Class Type Variables

```
variable2.set("Josephine", 1);
```



Part 3 of
3)

- All parameters in Java are *call-by-value* parameters
 - A parameter is a **local variable** that is set equal to the value of its argument
 - Therefore, any change to the value of the parameter cannot change the value of its argument
- 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

Class Parameters

- The value plugged into a class type parameter is a reference (memory address)
 - Therefore, the parameter becomes another name for the argument
 - 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
 - Note that, because it still is a call-by-value parameter, any change made to the class type parameter itself (i.e., its address) will not change its argument (the reference or memory address)

The value of the object named by the argument can be updated but the argument itself will not be changed

Class Parameters

Display 5.14 Parameters of a Class Type

```
1 public class ClassParameterDemo
2 {
3     public static void main(String[] args)
4     {
5         ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6         System.out.println(anObject);
7         System.out.println(
8             "Now we call changer with anObject as argument.");
9         ToyClass.changer(anObject);
10        System.out.println(anObject);
11    }
12 }
```

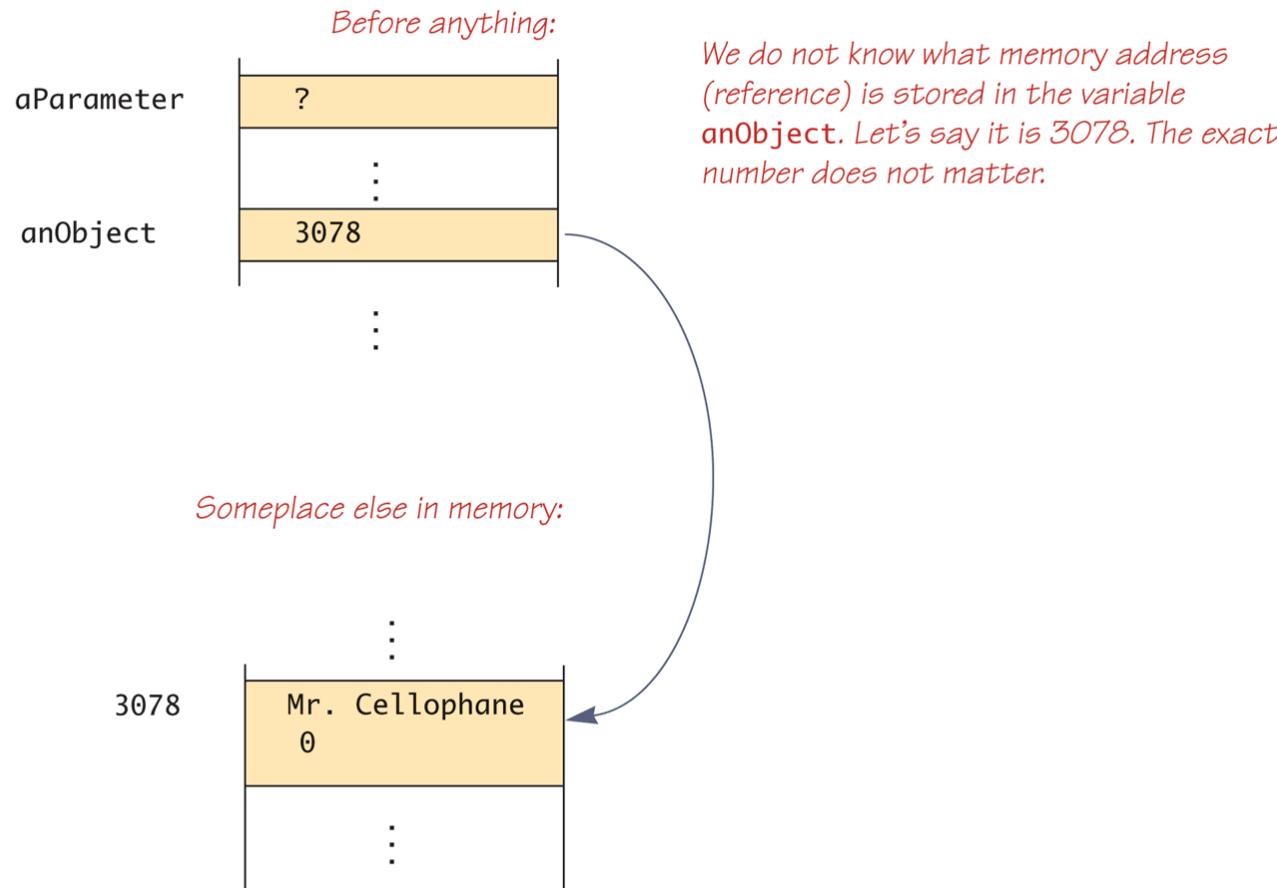
ToyClass is defined in Display 5.11.

Notice that the method changer changed the instance variables in the object anObject.

SAMPLE DIALOGUE

```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

E.g. ClassParameterDemo
Parameters of a Class Type

Display 5.15 Memory Picture for Display 5.14

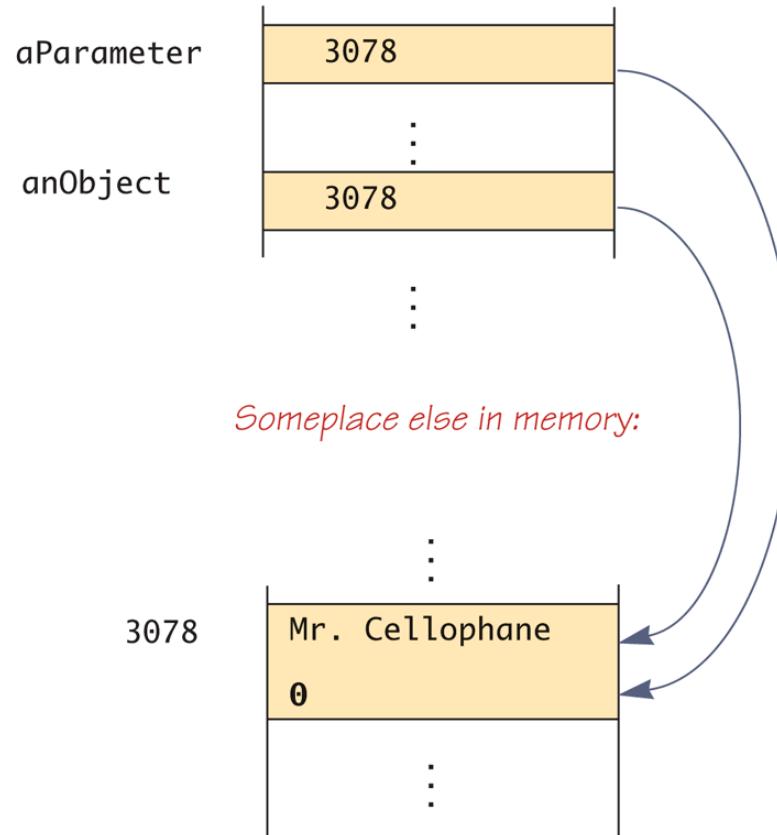
(continued)

**ay 5.14
1 of 3)**

Display 5.15 Memory Picture for Display 5.14

anObject is plugged in for aParameter.

anObject and aParameter become two names for the same object.



(continued)

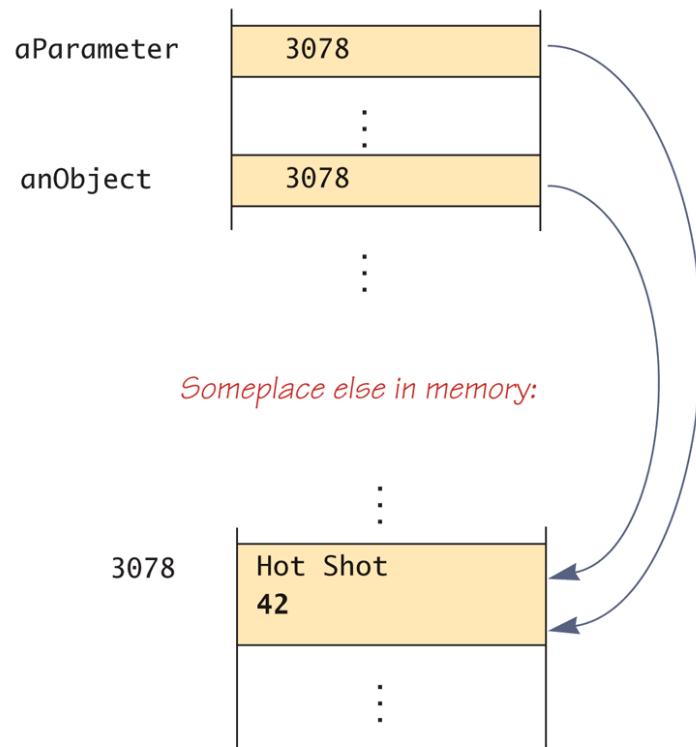
Display 5.14
Part 2 of 3)

Display 5.15 Memory Picture for Display 5.14

*ToyClass.changer(anObject); is executed
and so the following are executed:*

*aParameter.name = "Hot Shot";
aParameter.number = 42;*

As a result, anObject is changed.



isplay 5.14
Part 3 of 3)

- A method cannot change the value of a variable of a primitive type that is an argument to the method
- In contrast, a method can change the values of the instance variables of a class type that is an argument to the method

Differences Between Primitive and Class-Type Parameters

Display 5.16 Comparing Parameters of a Class Type and a Primitive Type

```
1  public class ParametersDemo
2  {
3      public static void main(String[] args)
4      {
5          ToyClass2 object1 = new ToyClass2(),
6              object2 = new ToyClass2();
7          object1.set("Scorpius", 1);
8          object2.set("John Crichton", 2);
9          System.out.println("Value of object2 before call to method:");
10         System.out.println(object2);
11         object1.makeEqual(object2);
12         System.out.println("Value of object2 after call to method:");
13         System.out.println(object2);
14
15         int aNumber = 42;
16         System.out.println("Value of aNumber before call to method: "
17                         + aNumber);
18         object1.tryToMakeEqual(aNumber);
19         System.out.println("Value of aNumber after call to method: "
20                         + aNumber);
21     }
22 }
```

ToyClass2 is defined in Display 5.17.

(continued)
**Type
of 2**

Display 5.16 Comparing Parameters of a Class Type and a Primitive Type

SAMPLE DIALOGUE

Value of object2 before call to method:

John Crichton 2

Value of object2 after call to method:

Scorpius 1

Value of aNumber before call to method: 42

Value of aNumber after call to method: 42

An argument of a class type can change.

An argument of a primitive type cannot change.

**Comparing Parameters of a Class Type and a Primitive Type
(Part 2 of 2)**

Display 5.17 A Toy Class to Use in Display 5.16

```
1  public class ToyClass2
2  {
3      private String name;
4      private int number;
5
6      public void set(String newName, int newNumber)
7      {
8          name = newName;
9          number = newNumber;
10
11     public String toString()
12     {
13         return (name + " " + number);
14     }
15 }
```

(continued)

A Toy Class to Use in Display 5.16
(Part 1 of 2)

Display 5.17 A Toy Class to Use in Display 5.16

```
14     public void makeEqual(ToyClass2 anObject)
15     {
16         anObject.name = this.name;
17         anObject.number = this.number;
18     }
19
20     public void tryToMakeEqual(int aNumber)
21     {
22         aNumber = this.number;
23     }
24
25     public boolean equals(ToyClass2 otherObject)
26     {
27         return ( name.equals(otherObject.name))
28             && (number == otherObject.number) );
29     }
```

Read the text for a discussion of the problem with this method.

<Other methods can be the same as in Display 5.11, although no other methods are needed or used in the current discussion.>

(Part 2 of 2)

- Used with variables of a class type, the assignment operator (`=`) produces two variables that name the same object
 - This is very different from how it behaves with primitive type variables
- The test for equality (`==`) also behaves differently for class type variables
 - The `==` operator only checks that two class type variables have the same memory address
 - Unlike the `equals` method, it does not check that their instance variables have the same values
 - Two objects in two different locations whose instance variables have exactly the same values would still test as being "not equal"

Pitfall: Use of `=` and `==` with Variables of a Class Type

- **null** is a special constant that may be assigned to a variable of any class type
YourClass yourObject = null;
- It is used to indicate that the variable has no "real value"
 - It is often used in constructors to initialize class type instance variables when there is no obvious object to use
- **null** is not an object: It is, rather, a kind of "placeholder" for a reference that does not name any memory location
 - Because it is like a memory address, use **==** or **!=** (instead of **equals**) to test if a class variable contains null
if (yourObject == null) ...

The Constant **null**

- Even though a class variable can be initialized to **null**, this does not mean that **null** is an object
 - **null** is only a placeholder for an object
- A method cannot be invoked using a variable that is initialized to **null**
 - The calling object that must invoke a method does not exist
- Any attempt to do this will result in a "Null Pointer Exception" error message
 - For example, if the class variable has not been initialized at all (and is not assigned to **null**), the results will be the same

Pitfall: Null Pointer Exception

- The **new** operator invokes a constructor which initializes an object, and returns a reference to the location in memory of the object created
 - This reference can be assigned to a variable of the object's class type
- Sometimes the object created is used as an argument to a method, and never used again
 - In this case, the object need not be assigned to a variable, i.e., given a name
- An object whose reference is not assigned to a variable is called an **anonymous object**
 - **ToyClass variable1= new ToyClass("Joe", 42);**
 - **if (variable1.equals (new ToyClass("JOE", 42)))**

The new Operator and Anonymous Objects

Display 5.18 Use of the method Double.parseDouble

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

3 public class InputExample
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);

8         System.out.println("Enter two numbers on a line.");
9         System.out.println("Place a comma between the numbers.");
10        System.out.println("Extra blank space is OK.");
11        String inputLine = keyboard.nextLine();

12        String delimiters = ", ";
13        StringTokenizer numberFactory =
14            new StringTokenizer(inputLine, delimiters);
```

(continued)

**Another Approach to Keyboard Input Using
Double.parseDouble (Part 1 of 3)**

Display 5.18 Use of the method Double.parseDouble

```
15     String string1 = null;
16     String string2 = null;
17     if (numberFactory.countTokens() >= 2)
18     {
19         string1 = numberFactory.nextToken();
20         string2 = numberFactory.nextToken();
21     }
22     else
23     {
24         System.out.println("Fatal Error.");
25         System.exit(0);
26     }

27     double number1 = Double.parseDouble(string1);
28     double number2 = Double.parseDouble(string2);

29     System.out.print("You input ");
30     System.out.println(number1 + " and " + number2);
31 }
32 }
```

(continued)

Display 5.18 Use of the method Double.parseDouble

SAMPLE DIALOGUE

Enter two numbers on a line.

Place a comma between the numbers.

Extra blank space is OK.

41.98, 42

You input is 41.98 and 42.0

**Another Approach to Keyboard Input Using
Double.parseDouble (Part 3 of 3)**

- When writing a program, it is very important to ensure that private instance variables remain truly private
- For a primitive type instance variable, just adding the **private** modifier to its declaration should insure that there will be no **privacy leaks**
- For a class type instance variable, however, adding the **private** modifier alone is not sufficient

Using and Misusing References

- A simple **Person** class could contain instance variables representing a person's name, the date on which they were born, and the date on which they died
- E.g. **Person.java**
- These instance variables would all be class types: name of type **String**, and two dates of type **Date**
- As a first line of defense for privacy, each of the instance variables would be declared **private**

```
public class Person
{
    private String name;
    private Date born;
    private Date died; //null is still alive
    ...
}
```

Designing A Person Class: Instance Variables

- In order to exist, a person must have (at least) a name and a birth date
 - Therefore, it would make no sense to have a no-argument **Person** class constructor
- A person who is still alive does not yet have a date of death
 - Therefore, the **Person** class constructor will need to be able to deal with a **null** value for date of death
- A person who has died must have had a birth date that preceded his or her date of death
 - Therefore, when both dates are provided, they will need to be checked for consistency

Designing a Person Class: Constructor

```
public Person(String initialName, Date birthDate,  
             Date deathDate)  
{  
    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);  
    }  
}
```

A Person Class Constructor

- A statement that is always true for every object of the class is called a *class invariant*
 - A class invariant can help to define a class in a consistent and organized way
- For the **Person** class, the following should always be true:
 - An object of the class **Person** has a date of birth (which is not **null**), and if the object has a date of death, then the date of death is equal to or later than the date of birth
- Checking the **Person** class confirms that this is true of every object created by a constructor, and all the other methods (e.g., the private method **consistent**) preserve the truth of this statement

Designing a Person Class: the Class Invariant

```
/** 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));  
}
```

Designing a Person Class: the Class Invariant

- The definition of **equals** for the class **Person** includes an invocation of **equals** for the class **String**, and an invocation of the method **equals** for the class **Date**
- Java determines which **equals** method is being invoked from the type of its calling object
- Also note that the **died** instance variables are compared using the **datesMatch** method instead of the **equals** method, since their values may be **null**

Designing a Person Class: the equals and datesMatch Methods

```
public boolean equals(Person otherPerson)
{
    if (otherPerson == null)
        return false;
    else
        return (name.equals(otherPerson.name) &&
                born.equals(otherPerson.born) &&
                datesMatch(died, otherPerson.died));
}
```

Designing a Person Class: the equals Method

```
/** To match date1 and date2 must either be the
   same date or both be null.
*/
private static boolean datesMatch(Date date1,
                                  Date date2)
{
    if (date1 == null)
        return (date2 == null);
    else if (date2 == null) //&& date1 != null
        return false;
    else // both dates are not null.
        return(date1.equals(date2));
}
```

Designing a Person Class: the datesMatch Method

- Like the **equals** method, note that the **Person** class **toString** method includes invocations of the **Date** class **toString** method

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

Designing a Person Class: the **toString** Method

- A *copy constructor* is a constructor with a single argument of the same type as the class
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object
- Note how, in the **Date** copy constructor, the values of all of the primitive type private instance variables are merely copied

Copy Constructors

```
public Date(Date aDate) //constructor - chapter 4
{
    if (aDate == null) //Not a real date.
    {
        System.out.println("Fatal Error.");
        System.exit(0);
    }

    month = aDate.month;
    day = aDate.day;
    year = aDate.year;
}
```

Copy Constructor for a Class with Primitive Type Instance Variables

- Unlike the **Date** class, the **Person** class contains three class type instance variables
- If the **born** and **died** class type instance variables for the new **Person** object were merely copied, then they would simply rename the **born** and **died** variables from the original **Person** object
 - born = original.born //dangerous**
 - died = original.died //dangerous**
 - This would not create an independent copy of the original object

Copy Constructor for a Class with Class Type Instance Variables

- The actual copy constructor for the **Person** class is a "safe" version that creates completely new and independent copies of **born** and **died**, and therefore, a completely new and independent copy of the original **Person** object
 - For example:
born = new Date(original.born);
- Note that in order to define a correct copy constructor for a class that has class type instance variables, *copy constructors must already be defined for the instance variables' classes*

Copy Constructor for a Class with Class Type Instance Variables

```
public Person(Person original)
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);
    if (original.died == null)
        died = null;
    else
        died = new Date(original.died);
}
```

Copy Constructor for a Class with Class Type Instance Variables

- The previously illustrated examples from the **Person** class show how an incorrect definition of a constructor can result in a *privacy leak*
- A similar problem can occur with incorrectly defined mutator or accessor methods
 - For example:

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

- Instead of:

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

Pitfall: Privacy Leaks

- The accessor method **getName** from the **Person** class appears to contradict the rules for avoiding privacy leaks:
public String getName()
{
return name; //Isn't this dangerous?
}
- Although it appears the same as some of the previous examples, it is not: The class **String** contains no mutator methods that can change any of the data in a **String** object

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
 - Objects of such a class are called ***immutable objects***
 - It is perfectly safe to return a reference to an immutable object because the object cannot be changed in any way
 - The **String** class is an immutable class

Mutable and Immutable Classes

- A class that contains public mutator methods or other public methods that can change the data in its objects is called a ***mutable class***, and its objects are called ***mutable objects***
 - Never write a method that returns a mutable object
 - Instead, use a copy constructor to return a reference to a completely independent copy of the mutable object

Mutable and Immutable Classes

- 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*
 - This type of copy can cause dangerous privacy leaks in a program

Deep Copy Versus Shallow Copy

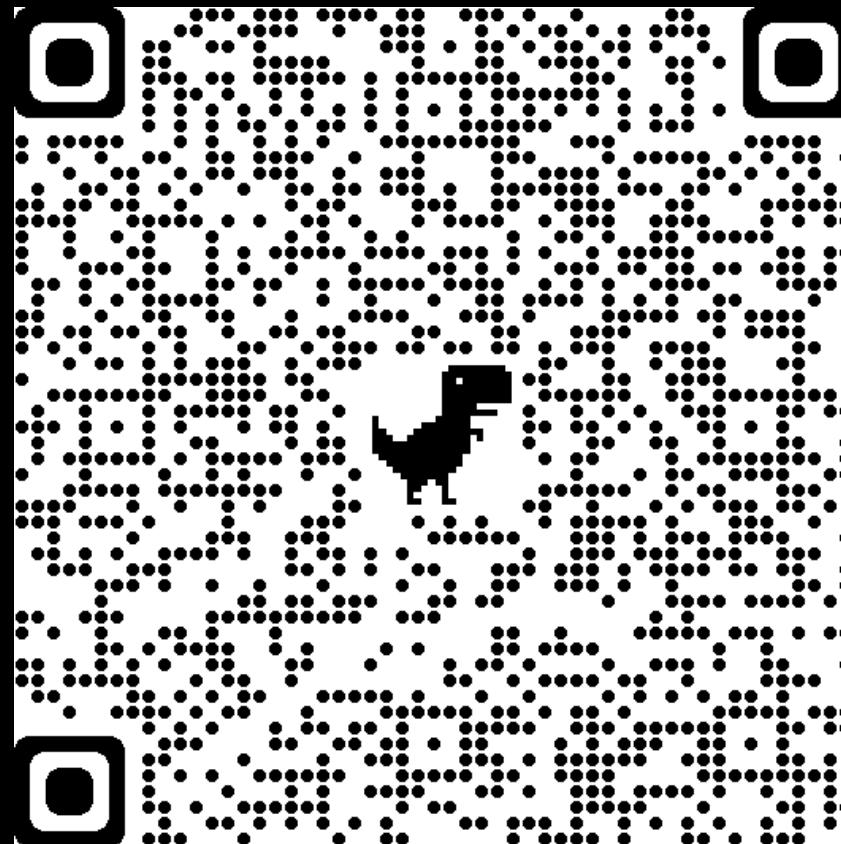
- **Static methods and static variables**
 - The Math class and wrapper classes
 - Automatic boxing and unboxing mechanism
- **References and class parameters**
 - Variables and Memory
 - Using and misusing references

Learning Outcomes

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
 - Examine your experience. Why do you care?)
- What insights have you had?
 - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections

Please fill in this microblog.



<http://go.unimelb.edu.au/5o8>

i.

- Class structure
- Instance variables and methods
- Different types of methods and their invocation
- Information hiding & Encapsulation
- Overloading methods
- Class constructors

Learning Outcomes



Programming and Software Development

COMP90041

Lecture 6

Arrays

NOTE: Some of the Material in these slides are adopted from

- * Lectures Notes prepared by Dr. Peter Schachte and
- * the Textbook resources

- **Static methods and static variables**
 - **The Math class and wrapper classes**
 - **Automatic boxing and unboxing mechanism**
- **References and class parameters**
 - **Variables and Memory**
 - **Using and misusing references**

Review: Week 5

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Outline

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Outline

```
1 import java.util.Scanner;
2
3 public class Scores {
4     public static void main(String[] args) {
5         Scanner keyboard = new Scanner(System.in);
6
7         double var1, var2, var3, var4, var5, max;
8
9         System.out.println("Enter 5 scores:");
10        var1 = keyboard.nextDouble();
11        max = var1;
12        var2 = keyboard.nextDouble();
13        max = (var2 > max)?var2:max;
14        var3 = keyboard.nextDouble();
15        max = (var3 > max)?var3:max;
16        var4 = keyboard.nextDouble();
17        max = (var4 > max)?var4:max;
18        var5 = keyboard.nextDouble();
19        max = (var5 > max)?var5:max;
20
21        System.out.println("The highest score is " + max);
22        System.out.println("The scores are:");
23
24        System.out.println("Score 1 differs from max by " + (max - var1));
25        System.out.println("Score 2 differs from max by " + (max - var2));
26        System.out.println("Score 3 differs from max by " + (max - var3));
27        System.out.println("Score 4 differs from max by " + (max - var4));
28        System.out.println("Score 5 differs from max by " + (max - var5));
29    }
30 }
```

Why do we need arrays?

- An **array** is a data structure used to process a collection of data that is all of the **same type**
 - An array behaves like a numbered list of variables with a uniform naming mechanism
 - It has a part that does not change: the name of the array
 - It has a part that can change: an integer in square brackets
 - For example, given five scores:

score[0], score[1], score[2], score[3], score[4]

Introduction to Arrays

- An array that behaves like this collection of variables, all of type **double**, can be created using one statement as follows:
double[] score = new double[5];
- Or using two statements:
double[] score;
score = new double[5];
 - The first statement declares the variable **score** to be of the array type **double []**
 - The second statement creates an array with five numbered variables of type **double** and makes the variable **score** a name for the array

Creating and Accessing Arrays

- The individual variables that together make up the array are called *indexed variables*
 - They can also be called **subscripted variables** or **elements** of the array
 - The number in square brackets is called an *index* or **subscript**
 - In Java, *indices must be numbered starting with 0, and nothing else*

score[0], score[1], score[2], score[3], score[4]

Creating and Accessing Arrays

- The number of indexed variables in an array is called the **length** or **size** of the array
- When an array is created, the length of the array is given in square brackets after the array type
- The indexed variables are then numbered starting with **0**, and ending with the integer that is *one less than the length of the array*

score[0], score[1], score[2], score[3], score[4]

Creating and Accessing Arrays

double[] score = new double[5];

- A variable may be used in place of the integer index (i.e., in place of the integer **5** above)
 - The value of this variable can then be read from the keyboard
 - This enables the size of the array to be determined when the program is run

double[] score = new double[count];

- An array can have indexed variables of any type, including any class type
- All of the indexed variables in a single array must be of the same type, called the **base type** of the array

Creating and Accessing Arrays

- An array is declared and created in almost the same way that objects are declared and created:

BaseType [] ArrayName = new BaseType [size];

- The **size** may be given as an expression that evaluates to a nonnegative integer, for example, an **int** variable

```
char[] line = new char[80];  
double[] reading = new double[count];  
Person[] specimen = new Person[100];
```

Example: ArrayOfScores.java

Declaring and Creating an Array

```
1 import java.util.Scanner;
2
3 public class ArrayOfScores
4 {
5     /**
6      * Reads in 5 scores and shows how much each
7      * score differs from the highest score.
8     */
9     public static void main(String[] args)
10    {
11        Scanner keyboard = new Scanner(System.in);
12        double[] score = new double[5];
13        int index;
14        double max;
15
16        System.out.println("Enter 5 scores:");
17        score[0] = keyboard.nextDouble();
18        max = score[0];
19        for (index = 1; index < 5; index++)
20        {
21            score[index] = keyboard.nextDouble();
22            if (score[index] > max)
23                max = score[index];
24            //max is the largest of the values score[0], ..., score[index]
25        }
26
27        System.out.println("The highest score is " + max);
28        System.out.println("The scores are:");
29        for (index = 0; index < 5; index++)
30            System.out.println(score[index] +
31                                " differs from max by " + (max - score[index]));
32    }
33 }
```

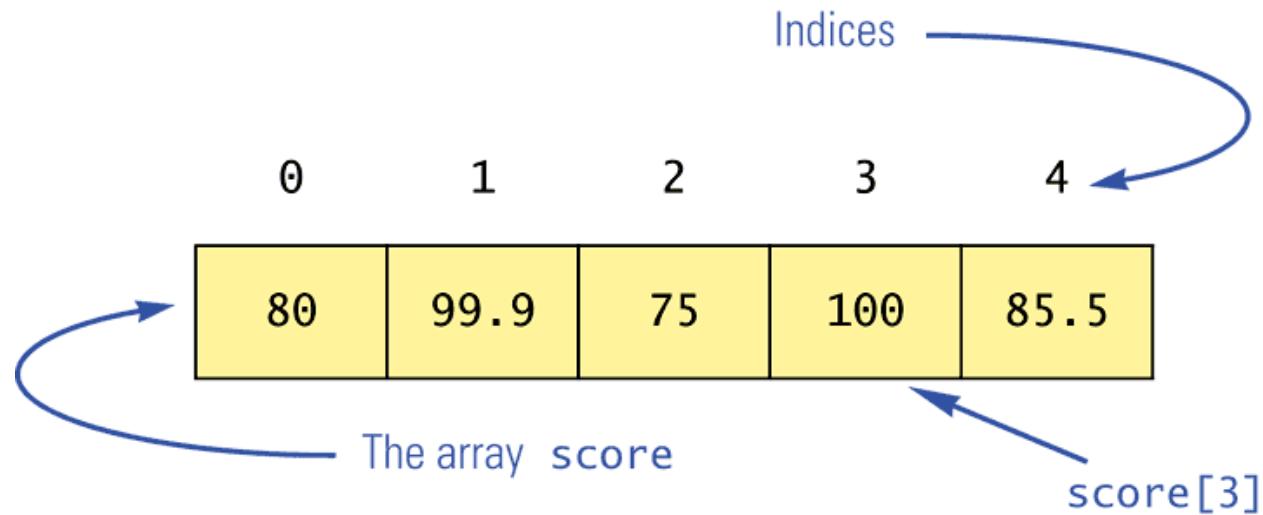
Example: ArrayOfScores.java

- Each array element can be used ***just like any other single variable*** by referring to it using an indexed expression: **score[0]**
- The array itself (i.e., the entire collection of indexed variables) can be referred to using the array name (without any square brackets): **score**
- An array index can be computed when a program is run
 - It may be represented by a variable: **score[index]**
 - It may be represented by an expression that evaluates to a suitable integer: **score[next + 1]**

Referring to Arrays and Array Elements

- The **for** loop is ideally suited for performing array manipulations:

```
for (index = 0; index < 5; index++)  
    System.out.println(score[index] +  
        " differs from max by " +  
        (max-score[index]));
```



Using the score Array in a Program

- Square brackets can be used to create a type name:
double[] score;
- Square brackets can be used with an integer value as part of the special syntax Java uses to create a new array:
score = new double[5];
- Square brackets can be used to name an indexed variable of an array:
max = score[0];

3 ways to use square brackets []

- An array is considered to be an object
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named **length**
 - When an array is created, the instance variable **length** is automatically set equal to its size
 - The value of **length** cannot be changed (other than by creating an entirely new array with **new**)
double[] score = new double[5];
 - Given **score** above, **score.length** has a value of 5

The length Instance Variable

- Array indices always start with **0**, and always end with the integer that is one less than the size of the array
 - The most common programming error made when using arrays is attempting to use a **nonexistent array index**
- When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be *out of bounds*
 - An out of bounds index will cause a program to terminate with a **run-time error message**
 - Array indices get out of bounds most commonly at the *first* or *last* iteration of a loop that processes the array: Be sure to test for this! (**off-by-one error**)

Pitfall: Array Index Out of Bounds

- An array can be initialized when it is declared
 - Values for the indexed variables are enclosed in braces, and separated by commas
 - The array size is automatically set to the number of values in the braces
- Given `age` above, `age.length` has a value of 3

Initializing Arrays

- Another way of initializing an array is by using a **for** loop

```
double[] reading = new double[100];
int index;
for (index = 0; index < reading.length; index++)
    reading[index] = 42.0;
```
- If the elements of an array are not initialized explicitly, they will **automatically be initialized** to the default value for their base type

Initializing Arrays

- **Introduction to arrays**
 - Creating and accessing arrays
 - Use for loops with arrays
- **Arrays and references**
- Programming with arrays
 - Sorting
- **ArrayList**
- Multidimensional arrays

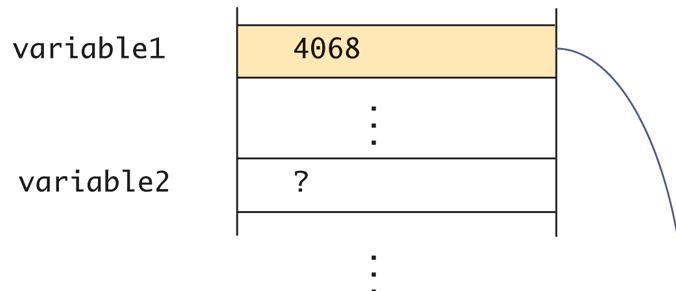
Outline

- Like class types, a variable of an array type holds a **reference**
 - Arrays are objects
 - A variable of an array type holds the address of where the array object is stored in memory
 - Array types are (usually) considered to be class types

Arrays and References

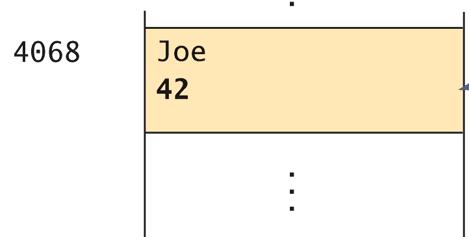
Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);  
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. Let's say it is `4068`. The exact number does not matter.

Someplace else in memory:



Note that you can think of

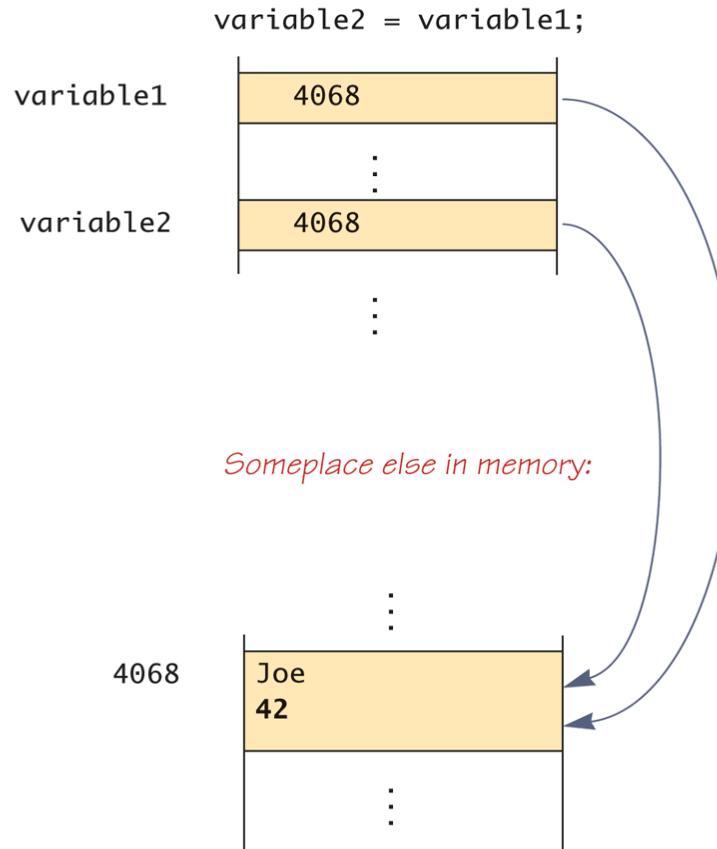
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

Assignment Operator with Class Type Variables (Part 1 of 3)

Display 5.13 Assignment Operator with Class Type Variables

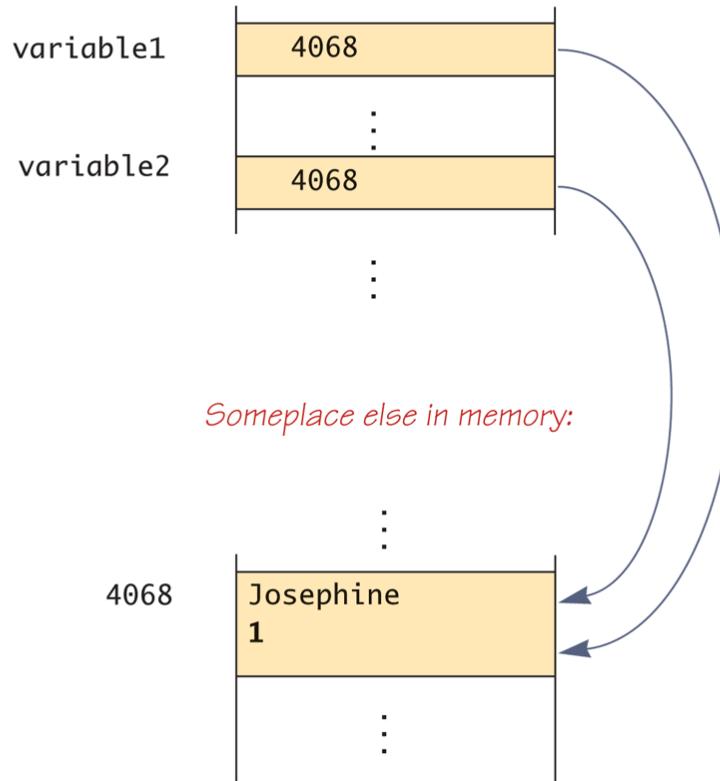


(continued)

Assignment Operator with Class Type Variables (Part 2 of 3)

Display 5.13 Assignment Operator with Class Type Variables

```
variable2.set("Josephine", 1);
```

**Assignment Operator with Class Type Variables (Part 3 of 3)**

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
 - An array variable names the array as a single item
double[] a;
 - A **new** expression creates an array object and stores the object in memory
new double[10]
 - An assignment statement places a reference to the memory address of an array object in the array variable
a = new double[10];

Arrays are Objects

- The previous steps can be combined into one statement
double[] a = new double[10];
- Note that the **new** expression that creates an array invokes a constructor that uses a nonstandard syntax
- Note also that as a result of the assignment statement above, **a** contains a single value: a memory address or *reference*
- Since an array is a reference type, the behavior of arrays with respect to assignment (**=**), equality testing (**==**), and parameter passing are the same as that described for classes

Arrays are Objects

- The base type of an array can be a class type
Date[] holidayList = new Date[20];
- The above example creates 20 indexed variables of type **Date**
 - It does **NOT** create 20 objects of the class **Date**
 - Each of these indexed variables are automatically initialized to **null**
 - Any attempt to reference any them at this point would result in a "null pointer exception" error message

Pitfall: Arrays with a Class Base Type

- Like any other object, each of the indexed variables requires a separate invocation of a constructor using **new** (singly, or perhaps using a **for** loop) to create an object to reference

```
holidayList[0] = new Date();
```

...

```
holidayList[19] = new Date();
```

OR

```
for (int i = 0; i < holidayList.length; i++)
```

```
    holidayList[i] = new Date();
```

- Each of the indexed variables can now be referenced since each holds the memory address of a **Date** object

Pitfall: Arrays with a Class Base Type



Intentionally blank

- Both ***array indexed variables*** and ***entire arrays*** can be used as arguments to methods
 - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument

Array Parameters

```
double n = 0.0;  
double[] a = new double[10];//all elements  
//are initialized to 0.0  
int i = 3;
```

- Given **myMethod** which takes one argument of type **double**, then all of the following are legal:

```
myMethod(n);//n evaluates to 0.0  
myMethod(a[3]);//a[3] evaluates to 0.0  
myMethod(a[i]);//i evaluates to 3,  
//a[3] evaluates to 0.0
```

```
void myMethod(double a){  
    a= a+1;  
}
```

In which case can we change the value of the argument?

Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
 - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only
 - **BaseType[]**
 - It does not specify the length of the array

Array Parameters

- The following method, **doubleElements**, specifies an array of **double** as its single argument:

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

Array Parameters

- Arrays of double may be defined as follows:
double[] a = new double[10];
double[] b = new double[30];
- Given the arrays above, the method **doubleElements** from class **SampleClass** can be invoked as follows:
SampleClass.doubleElements(a);
SampleClass.doubleElements(b);
 - Note that no square brackets are used when an entire array is given as an argument
 - Note also that a method that specifies an array for a parameter can take an **array of any length** as an argument

Array Parameters

- Because an array variable contains the memory address of the array it names, the assignment operator (**=**) only copies this memory address
 - It does not copy the values of each indexed variable
 - Using the assignment operator will make two array variables be different names for the same array
b = a;
 - The memory address in **a** is now the same as the memory address in **b**: **They reference the same array**

Pitfall: Use of **=** and **==** with Arrays

- For the same reason, the equality operator (**`==`**) only tests two arrays to see if they are stored in the same location in the computer's memory
 - It does not test two arrays to see if they contain the same values
`(a == b)`
 - The result of the above **boolean** expression will be **true** if **a** and **b** share the same memory address (and, therefore, reference the same array), and **false** otherwise

Pitfall: Use of `=` and `==` with Arrays

- In the same way that an **equals** method can be defined for a class, an **equalsArray** method can be defined for a type of array
 - This is how two arrays must be tested to see if they contain the same elements
 - The following method tests two integer arrays to see if they contain the same integer values

Pitfall: Use of = and == with Arrays

```
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;
}
```

Pitfall: Use of = and == with Arrays

- The heading for the **main** method of a program has a parameter for an array of **String**
 - It is usually called **args** by convention
public static void main(String[] args)
 - Note that since **args** is a parameter, it could be replaced by any other non-keyword identifier
- If a Java program is run without giving an argument to **main**, then a default empty array of strings is automatically provided

Arguments for the Method **main**

- Here is a program that expects three string arguments:

```
public class SomeProgram
{
    public static void main(String[] args)
    {
        System.out.println(args[0] + " " +
            args[2] + args[1]);
    }
}
```

- Note that if it needed numbers, it would have to convert them from strings first

Arguments for the Method main

- If a program requires that the **main** method be provided an array of strings argument, each element must be provided from the command line when the program is run

java SomeProgram Hi ! there

- This will set **args[0]** to "Hi", **args[1]** to "!", and **args[2]** to "there"
- It will also set **args.length** to 3
- When **SomeProgram** is run as shown, its output will be:

Hi there!

Arguments for the Method main

- In Java, a method may also return an array
 - The return type is specified in the same way that an array parameter is specified

```
public static int[] incrementArray(int[] a, int increment)
{
    int[] temp = new int[a.length];
    int i;
    for (i = 0; i < a.length; i++) {
        temp[i] = a[i] + increment;
    }
    return temp;
}
```

Methods that return an array

- **Introduction to arrays**
 - Creating and accessing arrays
 - Use for loops with arrays
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Outline

- The exact size needed for an array is not always known when a program is written, or it may vary from one run of the program to another
- A common way to handle this is to declare the array to be of the largest size that the program could possibly need
- Care must then be taken to keep track of how much of the array is actually used
 - An indexed variable that has not been given a meaningful value ***must never be referenced***

Partially Filled Arrays

- A variable can be used to keep track of how many elements are currently stored in an array
 - For example, given the variable **count**, the elements of the array **someArray** will range from positions **someArray[0]** through **someArray[count - 1]**
 - Note that the variable **count** will be used to process the partially filled array instead of **someArray.length**
 - Note also that this variable (**count**) must be an argument to any method that manipulates the partially filled array

Partially Filled Arrays

- The standard Java libraries include a number of collection classes
 - Classes whose objects store a collection of values
- Ordinary **for** loops cannot cycle through the elements in a collection object
 - Unlike array elements, collection object elements are not normally associated with indices
- However, there is a new kind of **for** loop, first available in Java 5.0, called a **for-each loop** or **enhanced for loop**
- This kind of loop can cycle through each element in a collection even though the elements are not indexed

The “for each” Loop

- Although an ordinary **for** loop cannot cycle through the elements of a collection class, an enhanced **for** loop can cycle through the elements of an array
- The general syntax for a **for**-each loop statement used with an array is

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

- The above **for**-each line should be read as "for each **VariableName** in **ArrayName** do the following:"
 - Note that **VariableName** must be declared within the **for**-each loop, not before
 - Note also that a colon (not a semicolon) is used after **VariableName**

The “for each” Loop

- The **for-each** loop can make code cleaner and less error prone
- If the indexed variable in a **for** loop is used only as a way to cycle through the elements, then it would be preferable to change it to a **for-each** loop
 - For example:

```
double sum = 0.0;
for (int i = 0; i < a.length; i++)
    sum += a[i];
```
 - Can be changed to:

```
double sum = 0.0;
for (double element : a)
    sum += element;
```
- Note that the **for-each** syntax is simpler and quite easy to understand
- Don't use for-each loop for arrays of primitive types if you want to update elements' values

The “for each” Loop

- Starting with Java 5.0, methods can be defined that take any number of arguments
- Essentially, it is implemented by taking in an array as argument, but the job of placing values in the array is done automatically
 - The values for the array are given as arguments
 - Java automatically creates an array and places the arguments in the array
 - Note that arguments corresponding to regular parameters are handled in the usual way

Methods with a variable number of parameters

- Such a method has as the last item on its parameter list a *vararg specification* of the form:

Type... **ArrayName**

- Note the three dots called an *ellipsis* that must be included as part of the vararg specification syntax
- Following the arguments for regular parameters are any number of arguments of the type given in the vararg specification
 - These arguments are automatically placed in an array
 - This array can be used in the method definition
 - Note that a vararg specification allows any number of arguments, including zero

Methods with a variable number of parameters

Display 6.7 Method with a Variable Number of Parameters

```
1 public class UtilityClass
2 {
3     /**
4      * Returns the largest of any number of int values.
5      */
6     public static int max(int... arg)
7     {
8         if (arg.length == 0)
9         {
10             System.out.println("Fatal Error: maximum of zero values.");
11             System.exit(0);
12         }
13
14         int largest = arg[0];
15         for (int i = 1; i < arg.length; i++)
16             if (arg[i] > largest)
17                 largest = arg[i];           This is the file UtilityClass.java.
18         return largest;
19     }
20 }
```

(continued)

Methods with a variable number of parameters

- If an accessor method does return the contents of an array, special care must be taken
 - Just as when an accessor returns a reference to any private object

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

- The example above will result in a *privacy leak*

Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array **anArray** itself
- Instead, an accessor method should return a reference to a *deep copy* of the private array object
 - Below, both **a** and **count** are instance variables of the class containing the **getArray** method

```
public double[] getArray()
{
    double[] temp = new double[count];
    for (int i = 0; i < count; i++)
        temp[i] = a[i];
    return temp
}
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

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

Privacy Leaks with Array Instance Variables

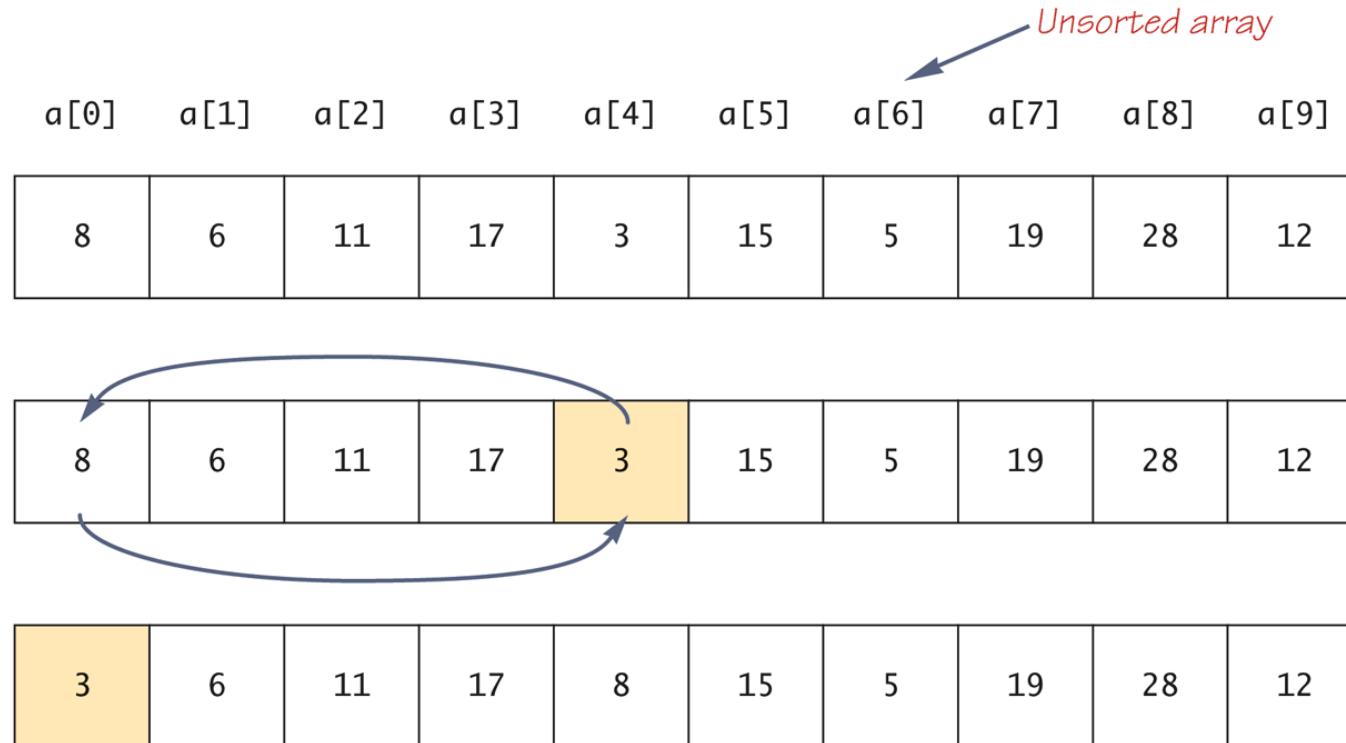
- A sort method takes in an array parameter **a**, and rearranges the elements in **a**, so that after the method call is finished, the elements of **a** are sorted in ascending order
- A *selection sort* accomplishes this by using the following algorithm:

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

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

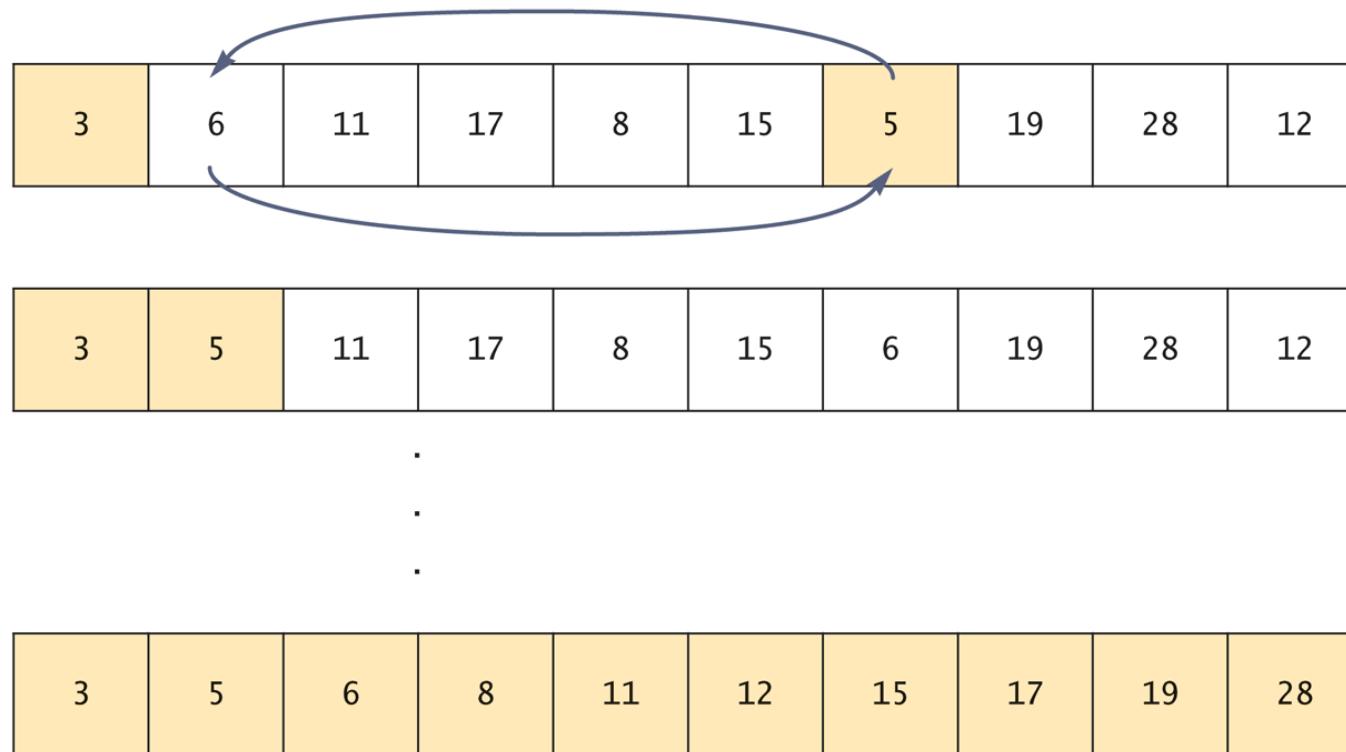
Sorting an Array

Display 6.10 Selection Sort



(continued)

Selection Sort (Part 1 of 2)

Display 6.10 Selection Sort

Selection Sort (Part 2 of 2)

```
public class SelectionSort
{
    /**
     * Precondition: count <= a.length;
     * The first count indexed variables have
     * values.
     * Action: Sorts a so that a[0] <= a[1] <=
     * ... <= a[count - 1].
     */
}
```

Example: SelectionSort.java

SelectionSort Class (Part 1 of 5)

```
public static void sort(double[] a, int count)
{
    int index, indexOfNextSmallest;
    for (index = 0; index < count - 1; index++)
    { //Place the correct value in a[index]:
        indexOfNextSmallest =
            indexOfSmallest(index, a, count);
        interchange(index, indexOfNextSmallest, a);
        //a[0] <= a[1] <= ... <= a[index] and these are
        //the smallest of the original array
        //elements. The remaining positions contain
        //the rest of the original array elements.
    }
}
```

SelectionSort Class (Part 2 of 5)

```
/**
```

```
Returns the index of the smallest value among  
a[startIndex], a[startIndex+1], ...  
a[numberUsed - 1]  
*/
```

```
private static int indexOfSmallest(int  
    startIndex, double[] a, int count)  
{  
    double min = a[startIndex];  
    int indexOfMin = startIndex;  
    int index;
```

SelectionSort Class (Part 3 of 5)

```
for (index = startIndex + 1; index < count; index++)
if (a[index] < min)
{
    min = a[index];
    indexOfMin = index;
    //min is smallest of a[startIndex] through
    //a[index]
}
return indexOfMin;
}
```

SelectionSort Class (Part 4 of 5)

```
/**
```

```
Precondition: i and j are legal indices for  
the array a.
```

```
Postcondition: Values of a[i] and a[j] have  
been interchanged.
```

```
*/
```

```
private static void interchange(int i, int j,  
double[] a)
```

```
{
```

```
    double temp;
```

```
    temp = a[i];
```

```
    a[i] = a[j];
```

```
    a[j] = temp; //original value of a[i]
```

```
}
```

```
}
```

SelectionSort Class (Part 5 of 5)

- Starting with version 5.0, Java permits enumerated types
 - An enumerated type is a type in which all the values are given in a (typically) short list
- 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};
 - Note that a value of an enumerated type is a kind of **named constant** and so, by convention, is spelled with all **uppercase** letters
 - As with any other type, variables can be declared of an enumerated type

Enumerated Types

- Given the following definition of an enumerated type:
enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
- A variable of this type can be declared as follows:
WorkDay meetingDay, availableDay;
- The value of a variable of this type can be set to one of the values listed in the definition of the type, or else to the special value **null**:

**meetingDay = WorkDay.THURSDAY;
availableDay = null;**

Enumerated Types Example

- Just like other types, variable of this type can be declared and initialized at the same time:

WorkDay meetingDay = WorkDay.THURSDAY;

- Note that the value of an enumerated type must be prefaced with the name of the type
- The value of a variable or constant of an enumerated type can be output using **println**
 - The code:
System.out.println(meetingDay);
 - Will produce the following output:
THURSDAY
 - As will the code:
System.out.println(WorkDay.THURSDAY);
 - Note that the type name **WorkDay** is not output

Enumerated Types Usage

- Although they may look like **String** values, values of an enumerated type are not **String** values
- However, they can be used for tasks which could be done by **String** values and, in some cases, work better
 - Using a **String** variable allows the possibility of setting the variable to a **nonsense value**
 - Using an enumerated type variable **constrains** the possible values for that variable
 - An error message will result if an attempt is made to give an enumerated type variable a value that is not defined for its type

Enumerated Types Usage

- Two variables or constants of an enumerated type can be compared using the **equals** method or the **==** operator
- However, the **==** operator has a nicer syntax
if (meetingDay == availableDay)
System.out.println("Meeting will be on
schedule.");
if (meetingDay == WorkDay.THURSDAY)
System.out.println("Long weekend!");

Enumerated Types Usage

Display 6.13 An Enumerated Type

```
1 public class EnumDemo
2 {
3     enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
4
5     public static void main(String[] args)
6     {
7         WorkDay startDay = WorkDay.MONDAY;
8         WorkDay endDay = WorkDay.FRIDAY;
9
10    }
11 }
```

SAMPLE DIALOGUE

Work starts on MONDAY
Work ends on FRIDAY

An Enumerated Type

- Enumerated types can be used to control a **switch** statement
 - The **switch** control expression uses a variable of an enumerated type
 - Case labels are the ***unqualified*** values of the same enumerated type
- The enumerated type control variable is set by using the static method **valueOf** to convert an input string to a value of the enumerated type
 - The input string must contain all upper case letters, or be converted to all upper case letters using the **toUpperCase** method

Enumerated Types in switch statements

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

(continued)

Example (Part 1 of 3)

Display 6.16 Enumerated Type in a switch Statement

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

14    switch (favorite)          The case labels must have just the name of
15    {                          the value without the type name and dot.
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 }
```

(continued)

Example (Part 2 of 3)

Display 6.16 Enumerated Type in a switch Statement

SAMPLE DIALOGUE

What is your favorite flavor?

Vanilla

Classic

SAMPLE DIALOGUE

What is your favorite flavor?

STRAWBERRY

I bet you said STRAWBERRY.

SAMPLE DIALOGUE

What is your favorite flavor?

PISTACHIO

This input causes the program to end and issue an error message.

Example (Part 3 of 3)

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Outline

- **ArrayList is a class in the standard Java libraries**
 - Unlike arrays, which have a fixed length once they have been created, an **ArrayList** is an object that can **grow** and **shrink** while your program is running
- In general, an **ArrayList** serves the same purpose as an array, except that an **ArrayList** can **change length** while the program is running
- The class **ArrayList** is implemented using an array as a private instance variable
 - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

The ArrayList Class

Why not always use an **ArrayList** instead of an array?

1. An **ArrayList** is **less efficient** than an array
 2. It does not have the convenient **square bracket** notation
 3. The base type of an **ArrayList** must be a class type or interface type (or other reference type): it cannot be a primitive type
- This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives

The ArrayList Class

- In order to make use of the **ArrayList** class, it must first be imported from the package **java.util**
- An **ArrayList** is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =  
    new ArrayList<BaseType>();
```

Compare with array:

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

Using the ArrayList Class

- An **initial capacity** can be specified when creating an **ArrayList** as well
 - The following code creates an **ArrayList** that stores objects of the base type **String** with an initial capacity of 20 items

```
ArrayList<String> list =  
    new ArrayList<String>(20);
```
 - Specifying an initial capacity does not limit the size to which an **ArrayList** can eventually grow
- Note that the base type of an **ArrayList** is specified as a **type parameter**

Using the **ArrayList** Class

- The **add** method is used to set an element for the first time in an **ArrayList**
list.add("something");
 - The method name **add** is overloaded
 - There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position

Using the ArrayList Class

- The **size** method is used to find out how many indices already have elements in the **ArrayList**
int howMany = list.size();
- The **set** method is used to replace any existing element, and the **get** method is used to access the value of any existing element
list.set(index, "something else");
String thing = list.get(index);

Using the ArrayList Class

- The simplest **add** method has a single parameter for the element to be added to the end of the ArrayList
 - **list.add("one");**
 - **list.add("two");**
 - **list.add("three");**

Tip: Summary of Adding to an ArrayList

- An element can be added at an already occupied list position by using the two-parameter version of **add**
- This causes the new element to be placed at the index specified, and every other member of the **ArrayList** to be **moved up** by one position
- **list.add(0,"Zero");**

Tip: Summary of Adding to an ArrayList

- The tools for manipulating arrays consist only of the square brackets and the instance variable **length**
- **ArrayLists, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays**

Methods in the Class ArrayList

- The **ArrayList** class is an example of a *collection* class
- Starting with version 5.0, Java has added a new kind of for loop called a **for-each** or **enhanced for loop**
 - This kind of loop has been designed to cycle through all the elements in a collection (like an **ArrayList**)

The “For Each” Loop

Display 14.2 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)

Example (Part 1 of 3)

Display 14.2 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)

Example (Part 2 of 3)

Display 14.2 A for-each Loop Used with an ArrayList

SAMPLE DIALOGUE

Enter list entries, when prompted.

Input an entry:

Practice Dancing.

More items for the list? **yes**

Input an entry:

Buy tickets.

More items for the list? **yes**

Input an entry:

Pack clothes.

More items for the list? **no**

The list contains:

Practice Dancing.

Buy tickets.

Pack clothes.

Example (Part 3 of 3)

Display 14.3 Golf Score Program

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

3 public class GolfScores
4 {
5     /**
6      Shows differences between each of a list of golf scores and their average.
7     */
8     public static void main(String[] args)
9     {
10         ArrayList<Double> score = new ArrayList<Double>();

11         System.out.println("This program reads golf scores and shows");
12         System.out.println("how much each differs from the average.");

13         System.out.println("Enter golf scores:");
14         fillArrayList(score);
15         showDifference(score);    Parameters of type ArrayList<Double> () are
16     }                                handled just like any other class parameter.
```

(continued)

Golf Score Program (Part 1 of 6)

Display 14.3 Golf Score Program

```
17     /**
18      Reads values into the array a.
19  */
20  public static void fillArrayList(ArrayList<Double> a)
21  {
22      System.out.println("Enter a list of nonnegative numbers.");
23      System.out.println("Mark the end of the list with a negative number.");
24      Scanner keyboard = new Scanner(System.in);
```

(continued)

Golf Score Program (Part 2 of 6)

Display 14.3 Golf Score Program

```
25     double next;
26     int index = 0;
27     next = keyboard.nextDouble();
28     while (next >= 0)
29     {
30         a.add(next);
31         next = keyboard.nextDouble();
32     }
33 }
34 /**
35     Returns the average of numbers in a.
36 */
37 public static double computeAverage(ArrayList<Double> a)
38 {
39     double total = 0;
40     for (Double element : a)
41         total = total + element;
```

Because of automatic boxing, we can treat values of type `double` as if their type were `Double`.

A for-each loop is the nicest way to cycle through all the elements in an `ArrayList`.

(continued)

Golf Score Program (Part 3 of 6)

Display 14.3 Golf Score Program

```
42     int number0fScores = a.size();
43     if (number0fScores > 0)
44     {
45         return (total/number0fScores);
46     }
47     else
48     {
49         System.out.println("ERROR: Trying to average 0 numbers.");
50         System.out.println("computeAverage returns 0.");
51         return 0;
52     }
53 }
```

(continued)

Golf Score Program (Part 4 of 6)

Display 14.3 Golf Score Program

```
54     /**
55      Gives screen output showing how much each of the elements
56      in a differ from their average.
57  */
58  public static void showDifference(ArrayList<Double> a)
59  {
60      double average = computeAverage(a);
61      System.out.println("Average of the " + a.size()
62                           + " scores = " + average);
63      System.out.println("The scores are:");
64      for (Double element : a)
65          System.out.println(element + " differs from average by "
66                               + (element - average));
67  }
68 }
```

(continued)

Golf Score Program (Part 5 of 6)

Display 14.3 Golf Score Program

SAMPLE DIALOGUE

This program reads golf scores and shows how much each differs from the average.

Enter golf scores:

Enter a list of nonnegative numbers.

Mark the end of the list with a negative number.

69 74 68 -1

Average of the 3 scores = 70.3333

The scores are:

69.0 differs from average by -1.33333

74.0 differs from average by 3.66667

68.0 differs from average by -2.33333

Golf Score Program (Part 6 of 6)

- An **ArrayList** automatically increases its capacity when needed
 - However, the capacity may increase beyond what a program requires
 - In addition, although an **ArrayList** grows automatically when needed, it does not shrink automatically
- If an **ArrayList** has a large amount of excess capacity, an invocation of the method **trimToSize** will shrink the capacity of the **ArrayList** down to the size needed

Tip: Use trimToSize to save memory

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Outline

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
 - You simply use as many square brackets as there are indices
 - Each index must be enclosed in its own brackets

```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] = new Person[10][100];
```

Multidimensional Arrays

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
 - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column

char[][] a = new char[5][12];

- Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, **char**)

Multidimensional Arrays

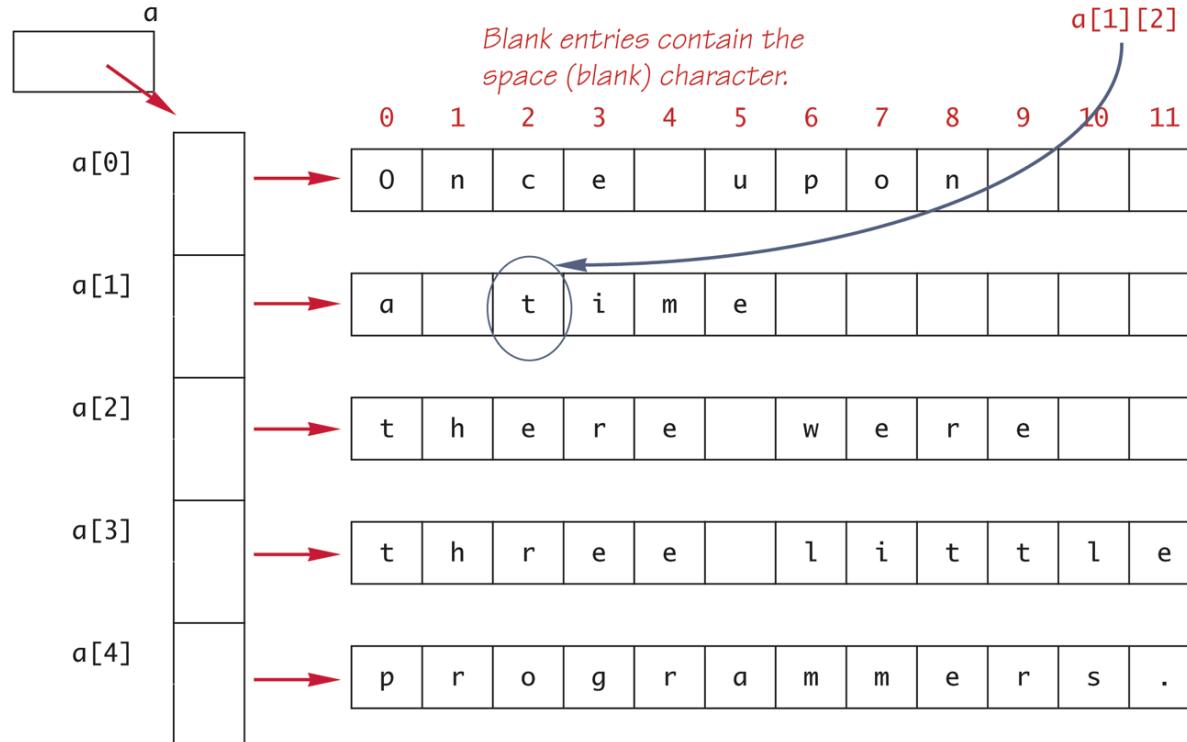
- In Java, a two-dimensional array, such as **a**, is actually an array of arrays
 - The array **a** contains a reference to a one-dimensional array of size 5 with a base type of **char[]**
 - Each indexed variable (**a[0]**, **a[1]**, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of **char[]**
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions

Multidimensional Arrays

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
char[][] a = new char[5][12];
```

Code that fills the array is not shown.



(continued)

2-D Array as an Array of Arrays (Part 1 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
        System.out.print(a[row][column]);
    System.out.println();
}
```

We will see that these can and should be replaced with expressions involving the length instance variable.

Produces the following output:

Once upon
a time
there were
three little
programmers.

2-D Array as an Array of Arrays (Part 2 of 2)

char[][] page = new char[30][100];

- The instance variable **length** does not give the total number of indexed variables in a two-dimensional array
 - Because a two-dimensional array is actually an array of arrays, the instance variable **length** gives the number of first indices (or "rows") in the array
 - **page.length** is equal to 30
 - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **length** for that "row" variable
 - **page[0].length** is equal to 100

Using the length instance variable

- The following program demonstrates how a nested **for** loop can be used to process a two-dimensional array
 - Note how each **length** instance variable is used

```
int row, column;  
for (row = 0; row < page.length; row++)  
    for (column = 0; column < page[row].length; column++)  
        page[row][column] = 'Z';
```

Using the length instance variable

- Each row in a two-dimensional array need not have the same number of elements
 - Different rows can have different numbers of columns
- An array that has a different number of elements per row it is called a *ragged array*

Ragged Arrays

double[][] a = new double[3][5];

- The above line is equivalent to the following:

double [][] a;

a = new double[3][]; //Note below

a[0] = new double[5];

a[1] = new double[5];

a[2] = new double[5];

- Note that the second line makes **a** the name of an array with room for 3 entries, each of which can be an array of **doubles** *that can be of any length*
- The next 3 lines each create an array of doubles of size 5

Ragged Arrays

```
double [][] a;
a = new double[3][];
```

- Since the above line does not specify the size of **a[0]**, **a[1]**, or **a[2]**, each could be made a different size instead:

```
a[0] = new double[5];
a[1] = new double[10];
a[2] = new double[4];
```

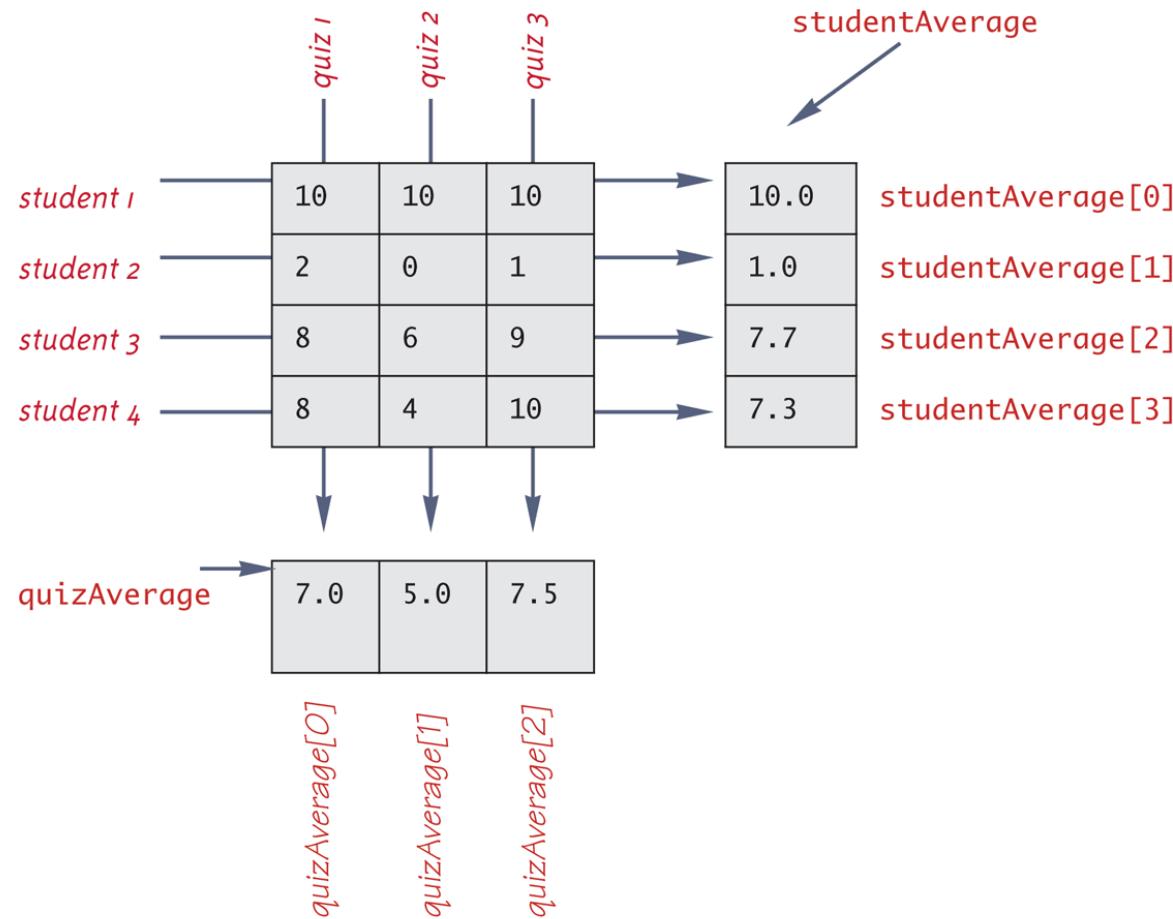
Ragged Arrays

- As an example of using arrays in a program, a class **GradeBook** is used to process quiz scores
- Objects of this class have three instance variables
 - **grade**: a two-dimensional array that records the grade of each student on each quiz
 - **studentAverage**: an array used to record the average quiz score for each student
 - **quizAverage**: an array used to record the average score for each quiz

A Grade Book Class

- The score that student 1 received on quiz number 3 is recorded in **grade[0][2]**
- The average quiz grade for student 2 is recorded in **studentAverage[1]**
- The average score for quiz 3 is recorded in **quizAverage[2]**
- Note the relationship between the three arrays

A Grade Book Class

Display 6.19 The Two-Dimensional Array grade

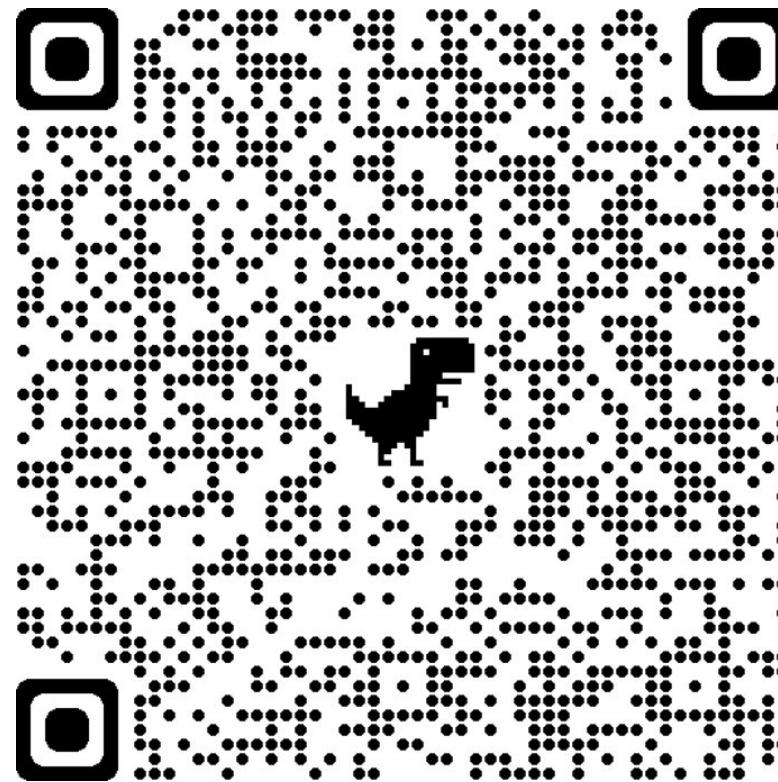
A Grade Book Class

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Learning Outcomes

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
 - Examine your experience. Why do you care?)
- What insights have you had?
 - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections



[http://go.unimelb.edu.au/5o8i.](http://go.unimelb.edu.au/5o8i)

Class Reflections



Programming and Software Development

COMP90041

Lecture 7

Inheritance & Polymorphism

NOTE: Some of the Material in these slides are adopted from

- * Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- * the Textbook resources

- **Introduction to arrays**
 - **Creating and accessing arrays**
 - **Use for loops with arrays**
- **Arrays and references**
- **Programming with arrays**
 - **Sorting**
- **ArrayList**
- **Multidimensional arrays**

Review: Week 6

- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

Outline

- **Introduction to UML & Packages & javadoc**
- Inheritance
- Access
- Polymorphism
- Abstract Classes

Outline

- Graphical representation systems for program design have been used
 - Flowcharts and *structure diagrams for example*
- *Unified Modeling Language (UML) is yet another graphical representation formalism*
 - UML is designed to reflect and be used with the OOP philosophy

Unified Modeling Language (UML)

- In 1996, Brady Booch, Ivar Jacobson, and James Rumbaugh released an early version of UML
 - Its purpose was to produce **a standardized graphical representation** language for object-oriented design and documentation
- Since then, UML has been developed and revised in response to feedback from the OOP community
 - Today, the UML standard is maintained and certified by the Object Management Group (OMG)
 - UML 2.5 is the newest version

History of UML

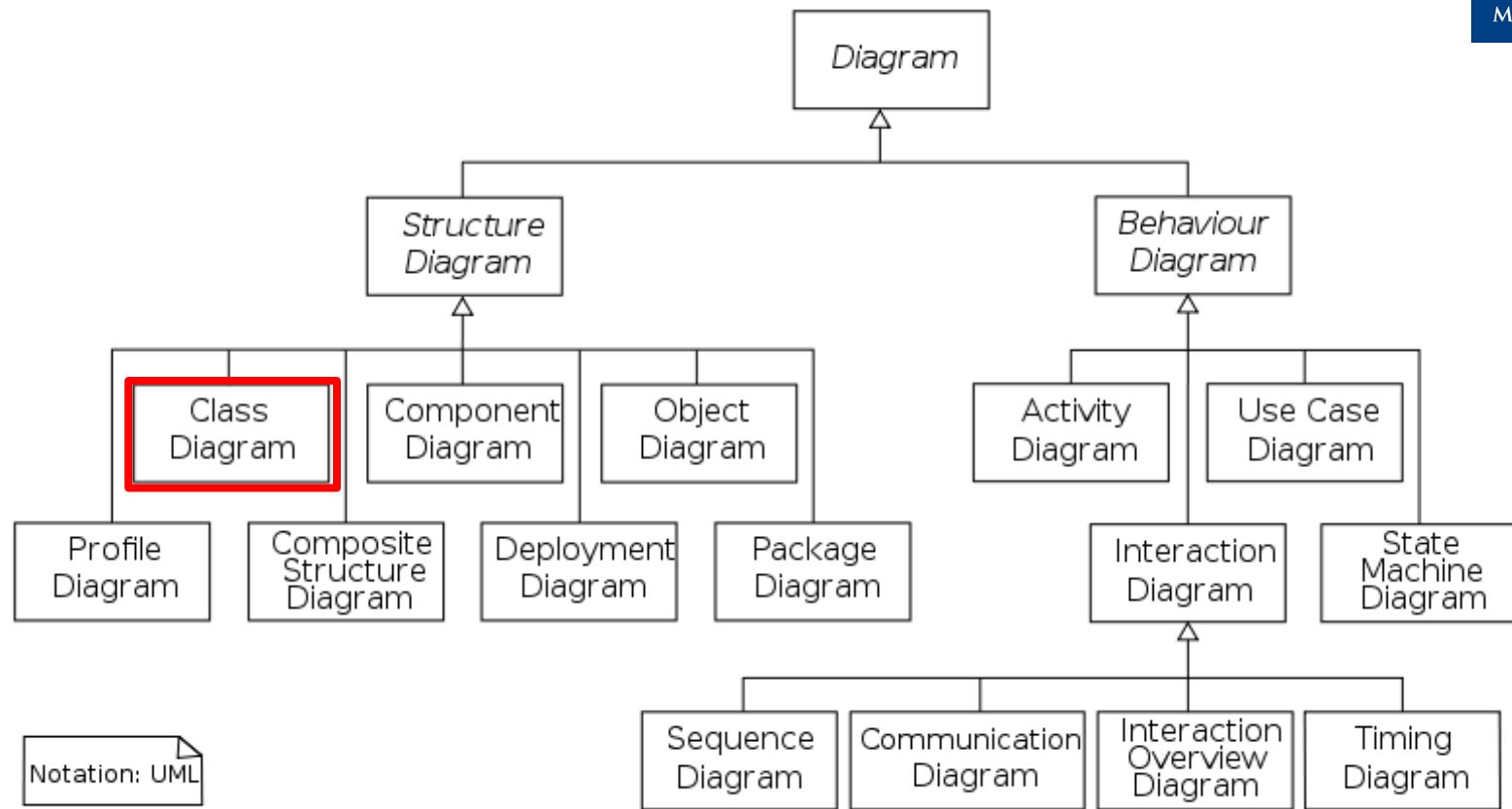


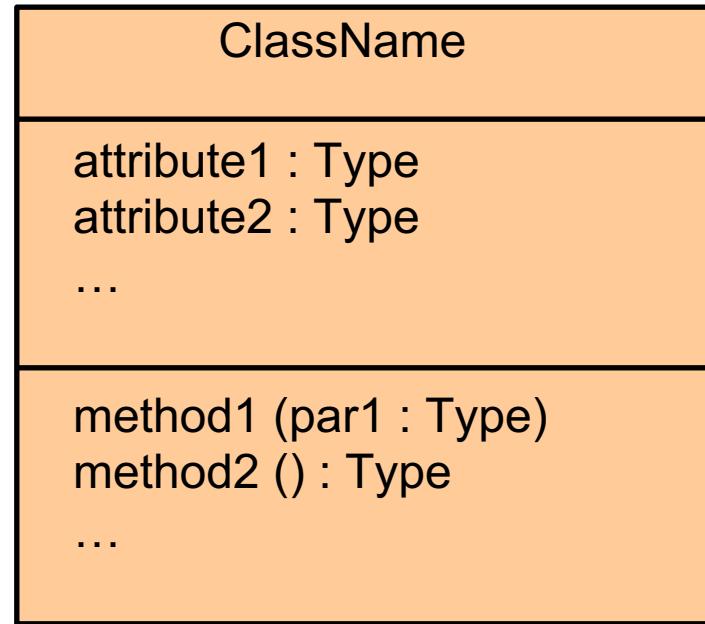
Image from Wikipedia

UML Diagrams

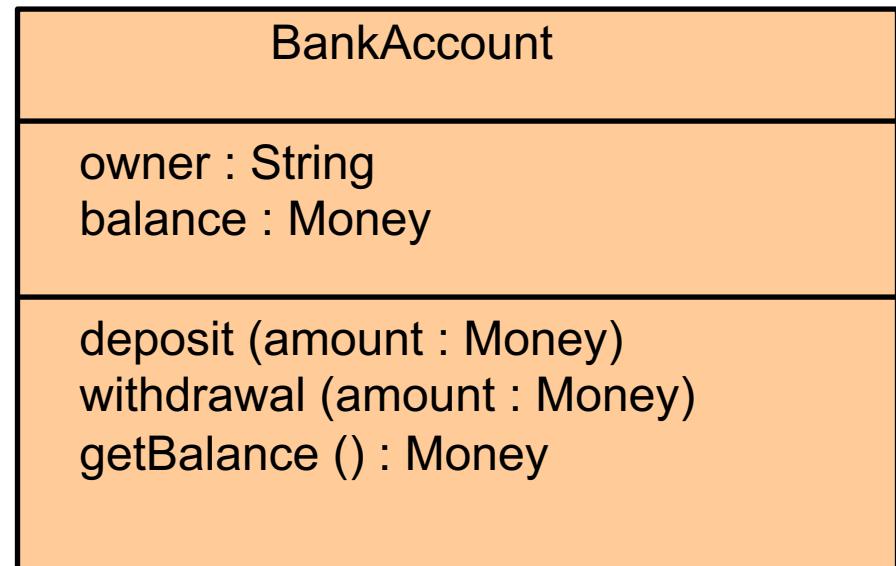
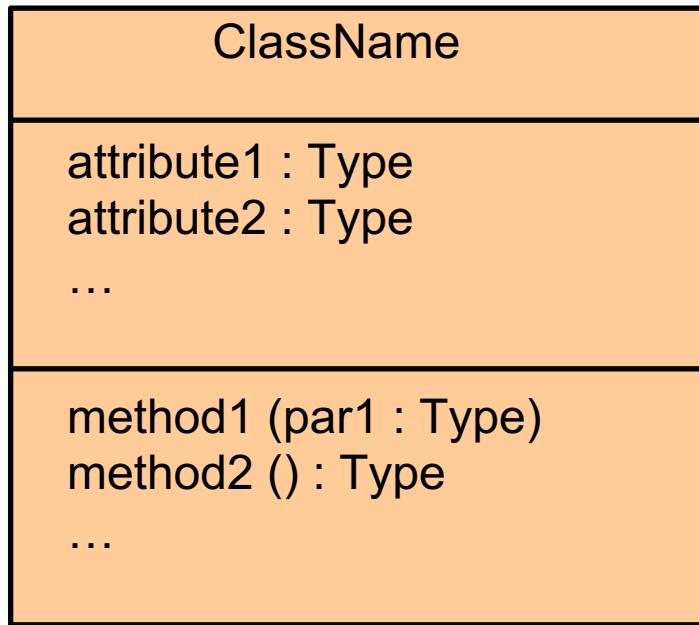
- A type of *structure diagram*
- Describes the structure of a system by showing the system's:
 - Classes
 - Their attributes (and the accessibility)
 - Methods
 - The relationships among the classes

UML Class Diagrams

- Classes are central to OOP, and the ***class diagram*** is the easiest of the UML graphical representations to understand and use
- A class diagram is divided up into three sections
 - The top section contains the class name
 - The middle section contains the data specification for the class
 - The bottom section contains the actions or methods of the class



UML Class Diagrams

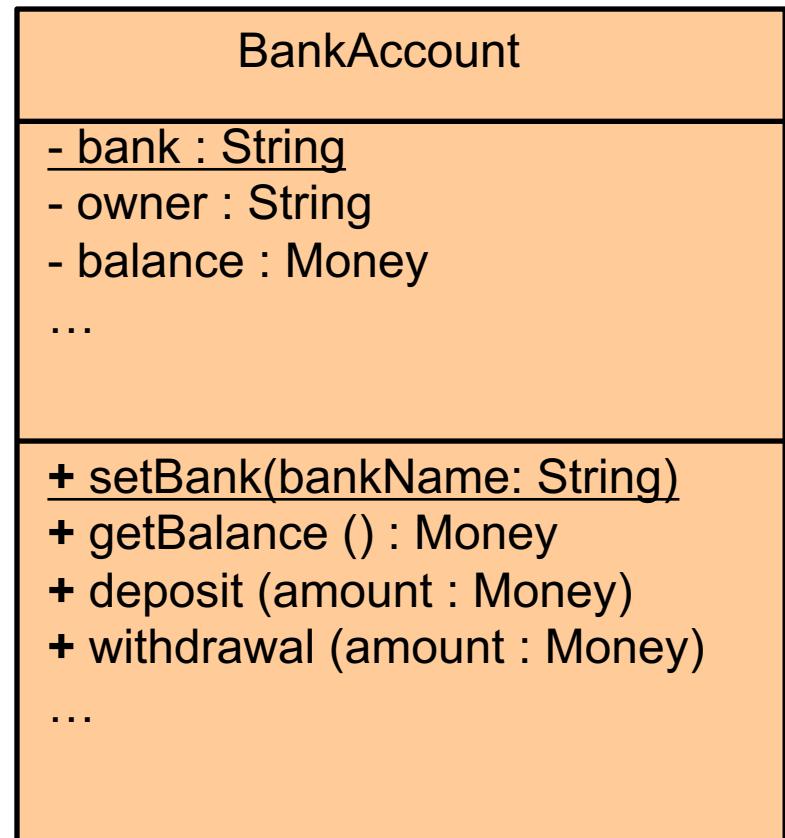


A UML Class

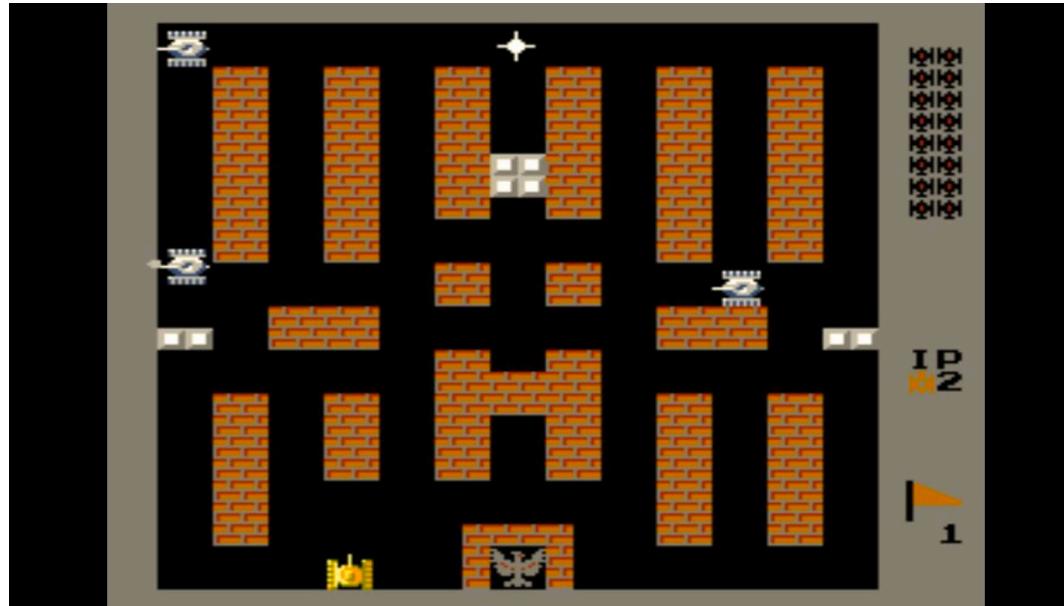
- The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type
- Each name is preceded by a character that specifies its **access type**:
 - A minus sign (-) indicates private access
 - A plus sign (+) indicates public access
 - A sharp (#) indicates protected access
 - A tilde (~) indicates package access

UML Class Diagrams

- A class diagram need not give a complete description of the class
 - If a given analysis does not require that all the class members be represented, then those members are not listed in the class diagram
 - Missing members are indicated with an ellipsis (three dots)



UML Class Diagrams



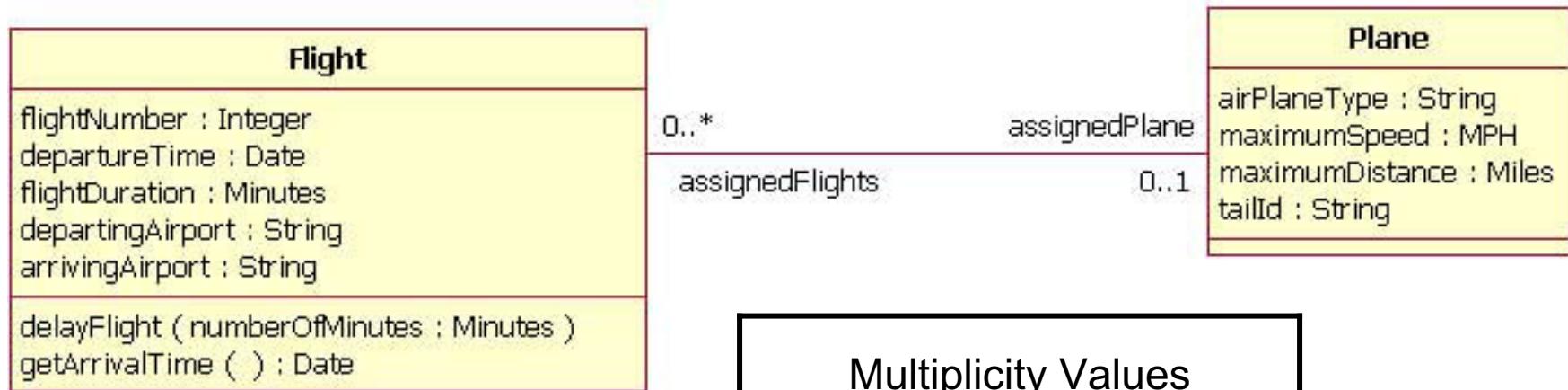
Battle City Tank - Gameplay:

- The player controls a [tank](#) and shoot projectiles to destroy enemy tanks around the playfield.
- The enemy tanks enter from the top of the screen and attempts to destroy the player's base (represented on the screen as a [phoenix](#) symbol), as well as the player's tank itself.
- A level is completed when the player destroys 20 enemy tanks, but the game ends if the player's base is destroyed or the player loses all available lives.
- Note that the player can destroy the base as well, so the player can still lose even after all enemy tanks are destroyed.

Practice: Identifying Classes and Modelling Classes using UML

- Rather than show just the interface of a class, class diagrams are primarily designed to show the relationships among classes
- UML has various ways to indicate the information flow from one class object to another using different sorts of ***annotated arrows***
- UML has annotations for class groupings into packages, for inheritance, and for other interactions
- In addition to these established annotations, **UML is extensible**

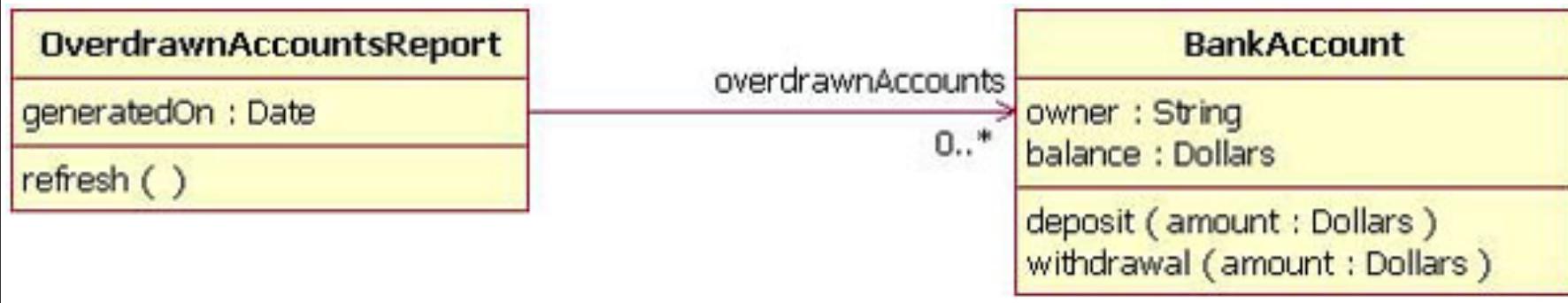
Class Interactions



Multiplicity Values	
Indicator	Meaning
n	exactly n
*	zero or many
0..n	zero to n
m..n	m to n

Associations: Bidirectional

Account



UML Packages

- Java uses **packages** to form libraries of classes
- A package is a group of classes that have been placed in a directory or folder, and that can be used in any program that includes an **import statement** that names the package
 - The import statement must be located at the beginning of the program file: Only blank lines, comments, and package statements may precede it
 - The program can be in a different directory from the package

Packages and Import Statements

- We have already used import statements to include some predefined packages in Java, such as **Scanner** from the **java.util** package

import java.util.Scanner;

- It is possible to make all the classes in a package available instead of just one class:

import java.util.*;

- Note that there is no additional overhead for importing the entire package
- Drawbacks of using *****
 - Worse readability of code due to lack of info of which package is used.
 - Possibly longer compilation time
 - Larger possibility of conflict of package names with other packages

Import Statements

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each java file for those classes:

package package_name;

- Only the **.class** files must be in the directory or folder, the **.java** files are optional
- Only blank lines and comments may precede the package statement
- *If there are both import and package statements, the package statement must precede any import statements*

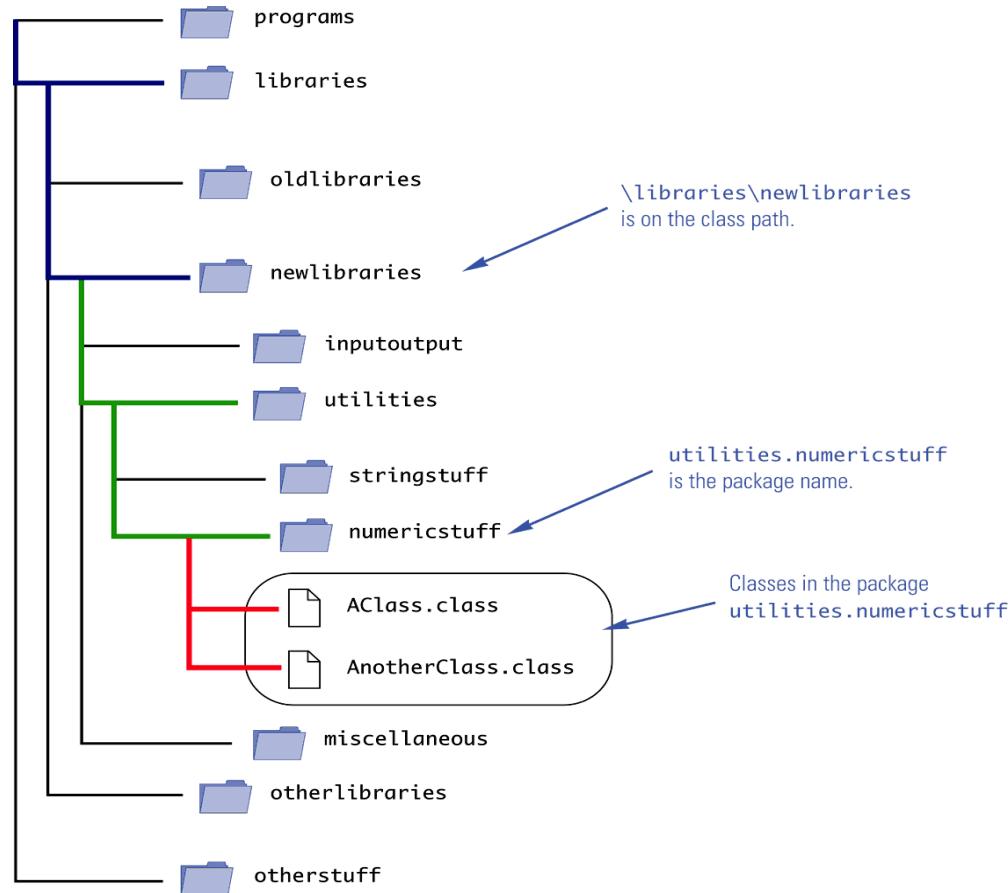
The package Statement

- The package **java.lang** contains the classes that are fundamental to Java programming
 - It is imported automatically, so no import statement is needed
 - Classes made available by **java.lang** include **Math**, **String**, and the wrapper classes

The package **java.lang**

- A package name is the path name for the directory or subdirectories that contain the package classes
- Java needs two things to find the directory for a package: the name of the package and the value of the **CLASSPATH** variable
 - The **CLASSPATH** environment variable is similar to the **PATH** variable, and is set in the same way for a given operating system
 - The **CLASSPATH** variable is set equal to the list of directories (including the current directory, ".") in which Java will look for packages on a particular computer
 - Java searches this list of directories in order, and uses the first directory on the list in which the package is found

Package Names and Directories

Display 5.14 A Package Name

A Package Name

- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- The import statement:

```
import utilities.numericstuff.*;
```

imports the **utilities.numericstuff** package only

- The import statements:

```
import utilities.numericstuff.*;
```

```
import utilities.numericstuff.statistical.*;
```

import both the **utilities.numericstuff** and
utilities.numericstuff.statistical packages

Pitfall: Subdirectories Are Not Automatically Imported

- All the classes in the current directory belong to an unnamed package called the *default package*
- As long as the current directory (.) is part of the **CLASSPATH** variable, all the classes in the default package are automatically available to a program

The Default Package

- If the **CLASSPATH** variable is set, the current directory must be included as one of the alternatives
 - Otherwise, Java may not even be able to find the **.class** files for the program itself
- If the **CLASSPATH** variable is not set, then all the class files for a program must be put in the current directory

Pitfall: Not Including the Current Directory in Your Class Path

- The class path can be manually specified when a class is compiled
 - Just add **-classpath** followed by the desired class path
 - This will compile the class, overriding any previous **CLASSPATH** setting
- You should use the **-classpath** option again when the class is run

Specifying a Class Path When You Compile

- In addition to keeping class libraries organized, packages provide a way to deal with *name clashes*: a situation in which two classes have the same name
 - Different programmers writing different packages may use the same name for one or more of their classes
 - This ambiguity can be resolved by using the *fully qualified name* (i.e., precede the class name by its package name) to distinguish between each class
package_name.ClassName
 - If the fully qualified name is used, it is no longer necessary to import the class (because it includes the package name already)

Name Clashes

- `CityTankGame.java`
 - `objects` folder
 - `Tank.java`
 - `Brick.java`
 - `Stone.java`
 - `Base.java`
 - ...
 - `graphics` folder
 - `Color.java`
 - `Renderer.java`
 - ...
 - `utilities` folder
 - `Commands.java`
 - ...
- package objects
- package graphics
- package utilities

Example – Battle City Tank

- Unlike a language such as C++, Java places both the interface and the implementation of a class in the same file
- However, Java has a program called **javadoc** that automatically extracts the interface from a class definition and produces documentation
 - This information is presented in HTML format, and can be viewed with a Web browser
 - If a class is correctly commented, a programmer need only refer to this *API (Application Programming Interface)* documentation in order to use the class
 - **javadoc can obtain documentation for anything from a single class to an entire package**

Introduction to javadoc

- The **javadoc** program extracts class headings, the headings for some comments, and headings for all public methods, instance variables, and static variables
 - In the normal default mode, no method bodies or private items are extracted
- To extract a comment, the following must be true:
 1. The comment must *immediately precede* a public class or method definition, or some other public item
 2. The comment must be a block comment, and the opening **/*** must contain an extra ***** (**/** . . . */**)
 - Note: Extra options would have to be set in order to extract line comments (**//**) and private items

Commenting Classes for javadoc

- In addition to any general information, the comment preceding a public method definition should include descriptions of parameters, any value returned, and any exceptions that might be thrown
 - This type of information is preceded by the @ symbol and is called an *@ tag*
 - @ tags come after any general comment, and each one is on a line by itself

```
/**
```

General Comments about the method . . .

@param aParameter Description of aParameter

@return What is returned

```
...  
*/
```

Commenting Classes for javadoc

- @ tags should be placed in the order found below
- If there are multiple parameters, each should have its own **@param** on a separate line, and each should be listed according to its left-to-right order on the parameter list
- If there are multiple authors, each should have its own **@author** on a separate line

@param Parameter_Name Parameter_Description
@return Description_Of_Value_Returned
@throws Exception_Type Explanation
@deprecated
@see Package_Name.Class_Name
@author Author
@version Version_Information

@ Tags

- To run **javadoc** on a package, give the following command:

javadoc -d Documentation_Directory Package_Name

- The HTML documents produced will be placed in the **Documentation_Directory**
 - If the **-d** and **Documentation_Directory** are omitted, **javadoc** will create suitable directories for the documentation
- To run **javadoc** on a single class, give the following command from the directory containing the class file:
javadoc ClassName.java
- To run javadoc on all the classes in a directory, give the following command instead: **javadoc *.java**

Running javadoc

Display 5.23 Options for `java.doc`

<code>-link Link_To_Other_Docs</code>	Provides a link to another set of documentation. Normally, this is used with either a path name to a local version of the Java documentation or the URL of the Sun Web site with standard Java documentation.
<code>-d Documentation_Directory</code>	Specifies a directory to hold the documentation generated. <i>Documentation_Directory</i> may be a relative or absolute path name.
<code>-author</code>	Includes author information (from @author tags). This information is omitted unless this option is set.
<code>-version</code>	Includes version information (from @version tags). This information is omitted unless this option is set.
<code>-classpath List_of_Directories</code>	Overrides the CLASSPATH environment variable and makes <i>List_of_Directories</i> the CLASSPATH for the execution of this invocation of javadoc. Does not permanently change the CLASSPATH variable.
<code>-private</code>	Includes private members as well as public members in the documentation.

Options for javadoc

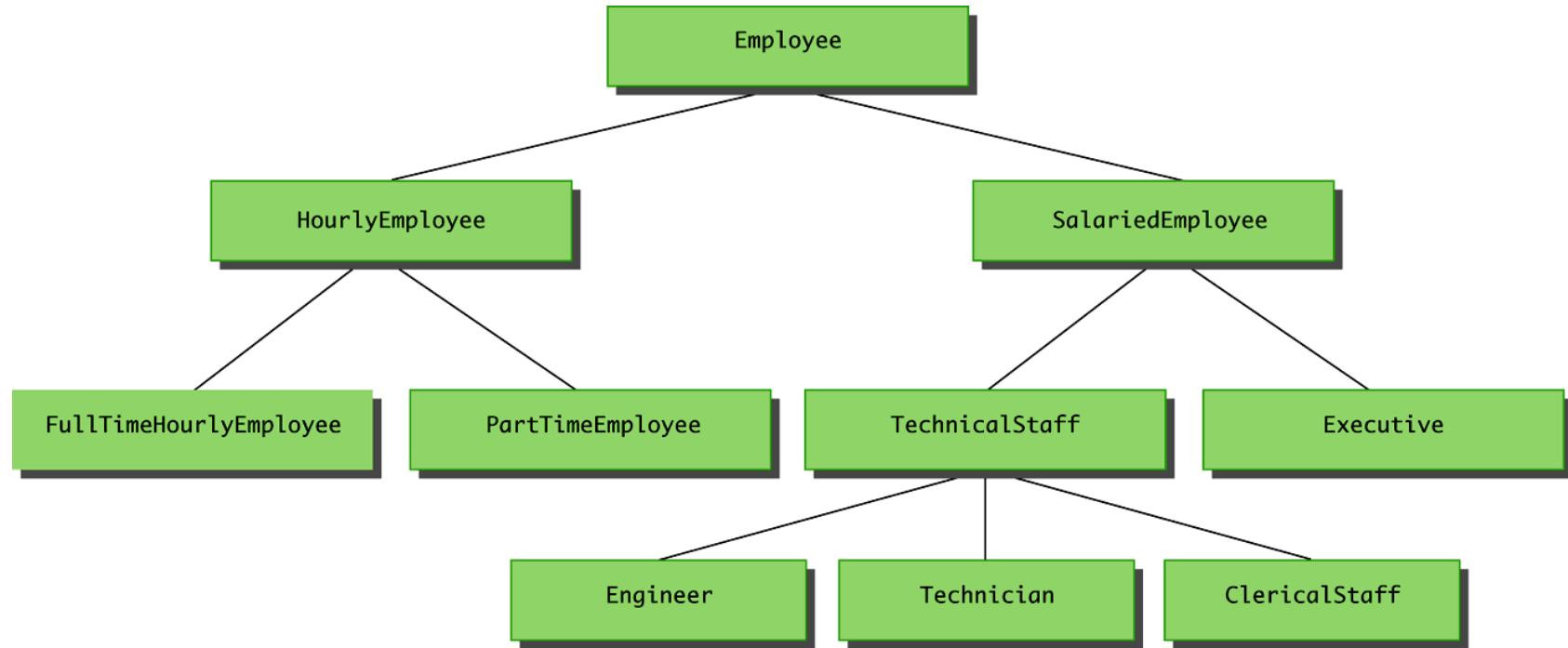
- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

Outline

- *Inheritance is one of the main techniques of object-oriented programming (OOP)*
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
 - The specialized classes are said to *inherit* the methods and instance variables of the general class

Introduction to Inheritance

Display 7.1 A Class Hierarchy



A Class Hierarchy

- Inheritance is the process by which a new class is created from another class
 - The new class is called a ***derived/child/sub*** class
 - The original class is called the ***base/parent/super*** class
- 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

Introduction to Inheritance

- When designing certain classes, there is often a natural hierarchy for grouping them
 - In a record-keeping program for the employees of a company, there are hourly employees and salaried employees
 - Hourly employees can be divided into full time and part time workers
 - Salaried employees can be divided into those on technical staff, and those on the executive staff

Derived Classes

- All employees share certain characteristics in common
 - All employees have a name and a hire date
 - The methods for setting and changing names and hire dates would be the same for all employees
- Some employees have specialized characteristics
 - Hourly employees are paid an hourly wage, while salaried employees are paid a fixed wage
 - The methods for calculating wages for these two different groups would be different

Derived Classes

- Eg.
 - Employee.java,
 - HourlyEmployee.java,
 - SalariedEmployee.java
- Within Java, a class called **Employee** can be defined that includes all employees
- This class can then be used to define classes for hourly employees and salaried employees
 - In turn, the **HourlyEmployee** class can be used to define a **PartTimeHourlyEmployee** class, and so forth

Derived Classes

- Since an hourly employee is an employee, it is defined as a *derived class* of the class **Employee**
 - Employee is the *base class*
 - The phrase **extends BaseClass** must be added to the derived class definition:

```
public class HourlyEmployee extends Employee
```

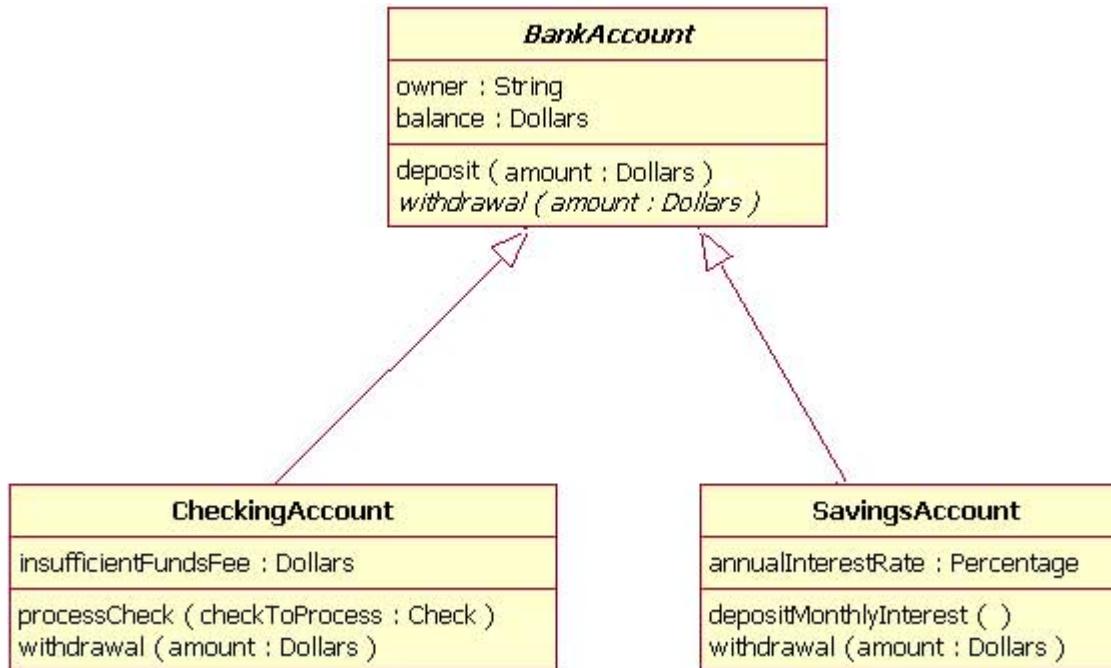
Derived Classes: Syntax

- Class **Employee** defines the instance variables **name** and **hireDate** in its class definition
- Class **HourlyEmployee** also has these instance variables, but they are not specified in its class definition
- Class **HourlyEmployee** has additional instance variables **wageRate** and **hours** that are specified in its class definition
- **Inherited Members:** The derived class inherits all the **public** methods, all the instance variables, and all the static variables from the base class
- The derived class can add more instance variables, static variables, and/or methods

Derived Classes: Inheritance

- A base class is often called the *parent class*
 - A derived class is then called a *child class*
- These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an *ancestor class*
 - If class **A** is an ancestor of class **B**, then class **B** can be called a *descendent* of class **A**

Ancestor and descendent Classes



Derived Classes (UML)

- Although a derived class inherits methods from the base class, it can change or *override* an inherited method if necessary
 - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class

Overriding a Method Definition

- Ordinarily, the type returned may not be changed when overriding a method
- However, if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
- This is known as a *covariant return type*
 - *Covariant return types are new in Java 5.0; they are not allowed in earlier versions of Java*

Changing the Return Type of an Overridden Method

- Given the following base class:

```
public class BaseClass
```

```
{ ...
```

```
public Employee getSomeone(int someKey)
```

```
...
```

- The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
```

```
{ ...
```

```
public HourlyEmployee getSomeone(int someKey)
```

```
...
```

Covariant Return Type

- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class
- However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class

Changing the Access Permission of an Overridden Method

- Given the following method header in a base case:
private void doSomething()
- The following method header is valid in a derived class:
public void doSomething()
- However, the opposite is not valid
- Given the following method header in a base case:
public void doSomething()
- The following method header is not valid in a derived class:
private void doSomething()

Changing the Access Permission of an Overridden Method

- Do not confuse *overriding* a method in a derived class with *overloading* a method name
 - When a method is **overridden**, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
 - When a method in a derived class has a different signature from the method in the base class, that is **overloading**
 - Note that when the derived class overloads the original method, **it still inherits the original method from the base class as well**

Pitfall: Overriding vs Overloading

- If the modifier **final** is placed before the definition of a *method*, then that method may not be redefined in a derived class
- If the modifier **final** is placed before the definition of a *class*, then that class may not be used as a base class to derive other classes

The final Modifier

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
 - In order to invoke a constructor from the base class, it uses a special syntax:

```
public derivedClass(int p1, int p2, double p3)
{
    super(p1, p2);
    instanceVariable = p3;
}
```

- In the above example, `super(p1, p2);` is a call to the base class constructor

The super Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword **super** instead
- A call to **super** must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to **super**

The super Constructor: Rules

- If a derived class constructor does not include an invocation of **super**, then the no-argument constructor of the base class will automatically be invoked
 - This can result in an error if the base class has not defined a no-argument constructor
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to **super** should always be used

The super Constructor: default

- Within the definition of a constructor for a class, **this** can be used as a name for invoking another constructor in the same class
 - The same restrictions on how to use a call to **super** apply to the **this** constructor
- If it is necessary to include a call to both **super** and **this**, the call using **this** must be made first, and then the constructor that is called must call **super** as its first action

The **this** Constructor

- Often, a no-argument constructor uses **this** to invoke an explicit-value 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)  
{  
    ...  
}
```

The **this** Constructor: typical usage

```
public HourlyEmployee()
{
    this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName, Date
theDate, double theWageRate, double theHours)
```

The **this** Constructor: example

- An object of a derived class has the type of the derived class, and it also has the type of the base class
- More generally, 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
- In fact, a derived class object can be used anywhere that an object of any of its ancestor types can be used
- Note, however, that this relationship does not go the other way
 - An ancestor type can never be used in place of one of its derived types

Tip: An Object of a Derived Class has more than One Type

- An instance variable that is private 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
 - For example, an object of the **HourlyEmployee** class cannot access the private instance variable **hireDate** by name, even though it is inherited from the **Employee** base class
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class
 - An object of the **HourlyEmployee** class can use the **getHireDate** or **setHireDate** methods to access **hireDate**

Pitfall: Use of Private Instance Variables from the Base Class

- If private instance variables of a class were accessible in method definitions of a derived class, then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access it in a method of that class
 - This would allow private instance variables to be changed by mistake or in inappropriate ways (for example, by not using the base type's accessor and mutator methods only)

Pitfall: Use of Private Instance Variables from the Base Class

- The private methods of the base class are like private variables in terms of not being directly available
- However, a private method is completely unavailable, unless invoked indirectly
 - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method
- This should not be a problem because private methods should just be used as helping methods
 - If a method is not just a helping method, then it should be public, not private

Pitfall: Private Methods Are Effectively Not Inherited

- Thanks to inheritance, most of the standard Java library classes can be enhanced by defining a derived class with additional methods
- For example, the **StringTokenizer** class enables all the tokens in a string to be generated one time
 - However, sometimes it would be nice to be able to cycle through the tokens a second or third time

An Enhanced StringTokenizer Class

- This can be made possible by creating a derived class:
 - For example, **Enhanced StringTokenizer** can inherit the useful behavior of **StringTokenizer**
 - It inherits the **countTokens** method unchanged
- The new behavior can be modeled by adding new methods, and/or overriding existing methods
 - A new method, **tokensSoFar**, is added
 - While an existing method, **nextToken**, is overridden

An Enhanced StringTokenizer Class

Display 7.7 Enhanced StringTokenizer

```
1 import java.util.StringTokenizer;
2
3 public class EnhancedStringTokenizer extends StringTokenizer
4 {
5     private String[] a;
6     private int count;
7
8     public EnhancedStringTokenizer(String theString)
9     {
10         super(theString);
11         a = new String[countTokens()];
12         count = 0;
13
14     public EnhancedStringTokenizer(String theString, String delimiters)
15     {
16         super(theString, delimiters);
17         a = new String[countTokens()];
18         count = 0;
```

The method `countTokens` is inherited and is not overridden.

(continued)

An Enhanced StringTokenizer Class (Part 1 of 4)

Display 7.7 Enhanced StringTokenizer

```
19     /**
20      Returns the same value as the same method in the StringTokenizer class,
21      but it also stores data for the method tokensSoFar to use.
22  */
23 public String nextToken()
24 {
25     String token = super.nextToken();
26     a[count] = token;
27     count++;
28     return token;
29 }
```

This method `nextToken` has its definition overridden.

`super.nextToken` is the version of `nextToken` defined in the base class `StringTokenizer`. This is explained more fully in Section 7.3.

(continued)

An Enhanced StringTokenizer Class (Part 2 of 4)

Display 7.7 Enhanced StringTokenizer

```
30     /**
31      Returns the same value as the same method in the StringTokenizer class,
32      changes the delimiter set in the same way as does the same method in the
33      StringTokenizer class, but it also stores data for the method tokensSoFar to use.
34   */
35   public String nextToken(String delimiters) {
36     String token = super.nextToken(delimiters);
37     a[count] = token;
38     count++;
39     return token;
40   }
41 }
```

This method `nextToken` also has its definition overridden.

`super.nextToken` is the version of `nextToken` defined in the base class `StringTokenizer`.

(continued)

An Enhanced StringTokenizer Class (Part 3 of 4)

Display 7.7 Enhanced StringTokenizer

```
42     /**
43      Returns an array of all tokens produced so far.
44      Array returned has length equal to the number of tokens produced so far.
45  */
46 public String[] tokensSoFar()
47 {
48     String[] arrayToReturn = new String[count];
49     for (int i = 0; i < count; i++)
50         arrayToReturn[i] = a[i];
51     return arrayToReturn;
52 }
53 }
```

tokensSoFar is a new method.

An Enhanced StringTokenizer Class (Part 4 of 4)

- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

Outline

- If a method or instance variable is modified by **protected** (rather than **public** or **private**), then it 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
- The **protected** modifier provides very weak protection compared to the **private** modifier
 - It allows direct access to any programmer who defines a suitable derived class
 - Therefore, instance variables should normally not be marked **protected**

Protected and Package Access

- 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
- Instance variables or methods having package access can be accessed *by name* inside the definition of any class in the same package
 - However, neither can be accessed outside the package
- Note that package access is more restricted than **protected**
 - Package access gives more control to the programmer defining the classes
 - Whoever controls the package directory (or folder) controls the package access

Protected and Package Access

Display 7.9 Access Modifiers

```
package somePackage;
```

```
public class A
{
    public int v1;
    protected int v2;
    int v3.//package
           //access
    private int v4;
```

```
public class B
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

```
public class C
    extends A
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

```
public class D
    extends A
{
    can access v1.
    can access v2.
    cannot access v3.
    cannot access v4.
```

```
public class E
{
    can access v1.
    cannot access v2.
    cannot access v3.
    cannot access v4.
```

In this diagram, “access” means access directly, that is, access by name.

A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.

Access Modifiers

- When considering package access, do not forget the default package
 - All classes in the **current directory** (not belonging to some other package) belong to an unnamed package called the **default package**
- If a class in the current directory is not in any other package, then it is in the default package
 - If an instance variable or method has package access, it can be accessed by name in the definition of any other class in the default package

Pitfall: Forgetting About the Default Package

- If a class **B** is derived from class **A**, and class **A** has a protected instance variable **n**, but the classes **A** and **B** are in *different packages*, then the following is true:
 - A method in class **B** can access **n** by name (**n** is inherited from class **A**)
 - A method in class **B** can create a local object of itself, which can access **n** by name (again, **n** is inherited from class **A**)

Pitfall: A Restriction on Protected Access

- However, if a method in class **B** creates an object of class **A**, it can not access **n** by name
 - A class knows about its own inherited variables and methods
 - However, it cannot directly access any instance variable or method of an ancestor class *unless they are public*
 - Therefore, **B** can access **n** whenever it is used as an instance variable of **B**, but **B** cannot access **n** when it is used as an instance variable of **A**
- This is true if **A** and **B** are *not* in the same package
 - If they were in the same package there would be no problem, because **protected** access implies package access

Pitfall: A Restriction on Protected Access

- A derived class demonstrates an "*is a*" relationship between it and its base class
 - Forming an "*is a*" relationship is one way to make a more complex class out of a simpler class
 - For example, an **HourlyEmployee** "*is an*" **Employee**
 - **HourlyEmployee** **is a more complex class compared to the more general Employee class**

Tip: “Is a” vs “Has a”

- Another way to make a more complex class out of a simpler class is through a "has a" relationship
 - This type of relationship, called *composition*, occurs when a class contains an instance variable of a class type
 - The **Employee** class contains an instance variable, **hireDate**, of the class **Date**, so therefore, an **Employee "has a" Date**

Tip: “Is a” vs “Has a”

- Both kinds of relationships are commonly used to create complex classes, often within the same class
 - Since **HourlyEmployee** is a derived class of **Employee**, and contains an instance variable of class **Date**, then **HourlyEmployee** "*is an*" **Employee** and "*has a*" **Date**

Tip: “Is a” vs “Has a”

- Static variables 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 variables as they do for instance variables

Tip: Static Variables Are Inherited

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
 - Simply preface the method name with super and a dot
- ```
public String toString()
{
 return (super.toString() + "$" + wageRate);
}
```
- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

## Access to a Redefined Base Method

- It is only valid to use **super** to invoke a method from a direct parent
  - Repeating **super** will not invoke a method from some other ancestor class
- For example, if the **Employee** class were derived from the class **Person**, and the **HourlyEmployee** class were derived from the class **Employee**, it would not be possible to invoke the **toString** method of the **Person** class within a method of the **HourlyEmployee** class  
**super.super.toString() // ILLEGAL!**

# You Cannot Use Multiple supers

- In Java, every class is a descendent of the class **Object**
  - Every class has **Object** as its ancestor
  - Every object of every class is of type **Object**, as well as being of the type of its own class
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class **Object**

# The Class Object

- The class **Object** is in the package **java.lang** which is always imported automatically
- Having an **Object** class enables methods to be written with a parameter of type **Object**
  - A parameter of type **Object** can be replaced by an object of any class whatsoever
  - For example, some library methods accept an argument of type **Object** so they can be used with an argument that is an object of any class

# The Class Object

- The class **Object** has some methods that every Java class inherits
  - For example, the **equals** and **toString** methods
- Every object inherits these methods from some ancestor class
  - Either the class **Object** itself, or a class that itself inherited these methods (ultimately) from the class **Object**
- However, these inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

# The Class Object

- Since the **equals** method is always inherited from the class **Object**, methods like the following simply overload it:  
**public boolean equals(Employee otherEmployee)**  
**{ ... }**
- However, this method should be overridden, not just overloaded:

```
public boolean equals(Object otherObject)
{ ... }
```

## The Right Way to Define equals

- The overridden version of **equals** must meet the following conditions
  - The parameter **otherObject** of type **Object** must be type cast to the given class (e.g., **Employee**)
  - However, the new method should only do this if **otherObject** really is an object of that class, and if **otherObject** is not equal to **null**
  - Finally, it should compare each of the instance variables of both objects

## The Right Way to Define **equals**

- Every object inherits the same **getClass()** method from the **Object** class
  - This method is marked **final**, so it cannot be overridden
- An invocation of **getClass()** on an object returns a representation *only* of the class that was used with **new** to create the object
  - The results of any two such invocations can be compared with **==** or **!=** to determine whether or not they represent the exact same class  
**(object1.getClass() == object2.getClass())**

# The **getClass()** Method

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

## A Better equals Method

- The **instanceof** operator checks if an object is of the type given as its second argument

### **Object instanceof ClassName**

- This will return **true** if **Object** is of type **ClassName**, and otherwise return **false**
- Note that this means it will return **true** if **Object** is the type of *any descendent class* of **ClassName**

# The instanceof Operator

- Many authors suggest using the **instanceof** operator in the definition of **equals**
  - Instead of the **getClass()** method
- The **instanceof** operator will return **true** if the object being tested is a member of the class for which it is being tested
  - However, it will return **true** if it is a descendent of that class as well
- It is possible (and especially disturbing), for the **equals** method to behave inconsistently given this scenario

## getClass vs instanceof

- Here is an example using the class **Employee**

```
... //excerpt from bad equals method
else if(!(OtherObject instanceof Employee))
 return false; ...
```

and the class **HourlyEmployee**

```
... //excerpt from bad equals method
else if(!(OtherObject instanceof HourlyEmployee))
 return false; ...
```

- Now consider the following:

```
Employee e = new Employee("Joe", new Date());
HourlyEmployee h = new
 HourlyEmployee("Joe", new Date(), 8.5, 40);
boolean testH = e.equals(h);
boolean testE = h.equals(e);
```

## getClass vs instanceof

- **testH** will be **true**, because **h** is an **Employee** with the **same name and hire date as e**
- However, **testE** will be **false**, because **e** is not an **HourlyEmployee**, and cannot be compared to **h**
- Note that this problem would not occur if the **getClass ()** method were used instead, as in the previous **equals** method example

## getClass vs instanceof

- Both the **instanceof** operator and the **getClass()** method can be used to check the class of an object
- However, the **getClass()** method is more exact
  - The **instanceof** operator simply tests the class of an object
  - The **getClass()** method used in a test with **==** or **!=** tests if two objects *were created with* the same class

# Instanceof and getClass

- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

# Outline

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

# Introduction to Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism allows changes to be made to method definitions in the derived classes, *and have those changes apply to the software written for the base class*

# Introduction to Polymorphism

- If an appropriate **toString** method is defined for a class, then an object of that class can be output using **System.out.println**

```
Sale aSale = new Sale("tire gauge", 9.95);
System.out.println(aSale);
```

- Output produced:  
*tire gauge Price and total cost = \$9.95*
- This works because of late binding

## Late Binding with **toString**

- One definition of the method **println** takes a single argument of type **Object**:

```
public void println(Object theObject)
{
 System.out.println(theObject.toString());
}
```

- In turn, It invokes the version of **println** that takes a **String** argument
- Note that the **println** method was defined **before** the **Sale** class existed
- Yet, because of late binding, the **toString** method from the **Sale** class is used, not the **toString** from the **Object** class

## Late Binding with **toString**

- `toString();`

```
HourlyEmployee joe = new HourlyEmployee("Joe Worker",
 new Date("January", 1, 2004), 50.50, 160);
```

```
Employee mike = new Employee("Mike Jordan", new Date("March", 1,
1984));
```

```
System.out.println();
System.out.println("joe's record is as follows:");
System.out.println(joe);
```

```
Sstem.out.println();
System.out.println("mike's record is as follows:");
System.out.println(mike);
```

## Example

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called *early binding* or *static binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*

## Late Binding

- Java uses late binding for all methods (except private, **final**, and static methods)
- Because of late binding, a method can be written in a base class to perform a task, **even if portions of that task aren't yet defined**
- For an example, the relationship between a base class called **Sale** and its derived class **DiscountSale** will be examined

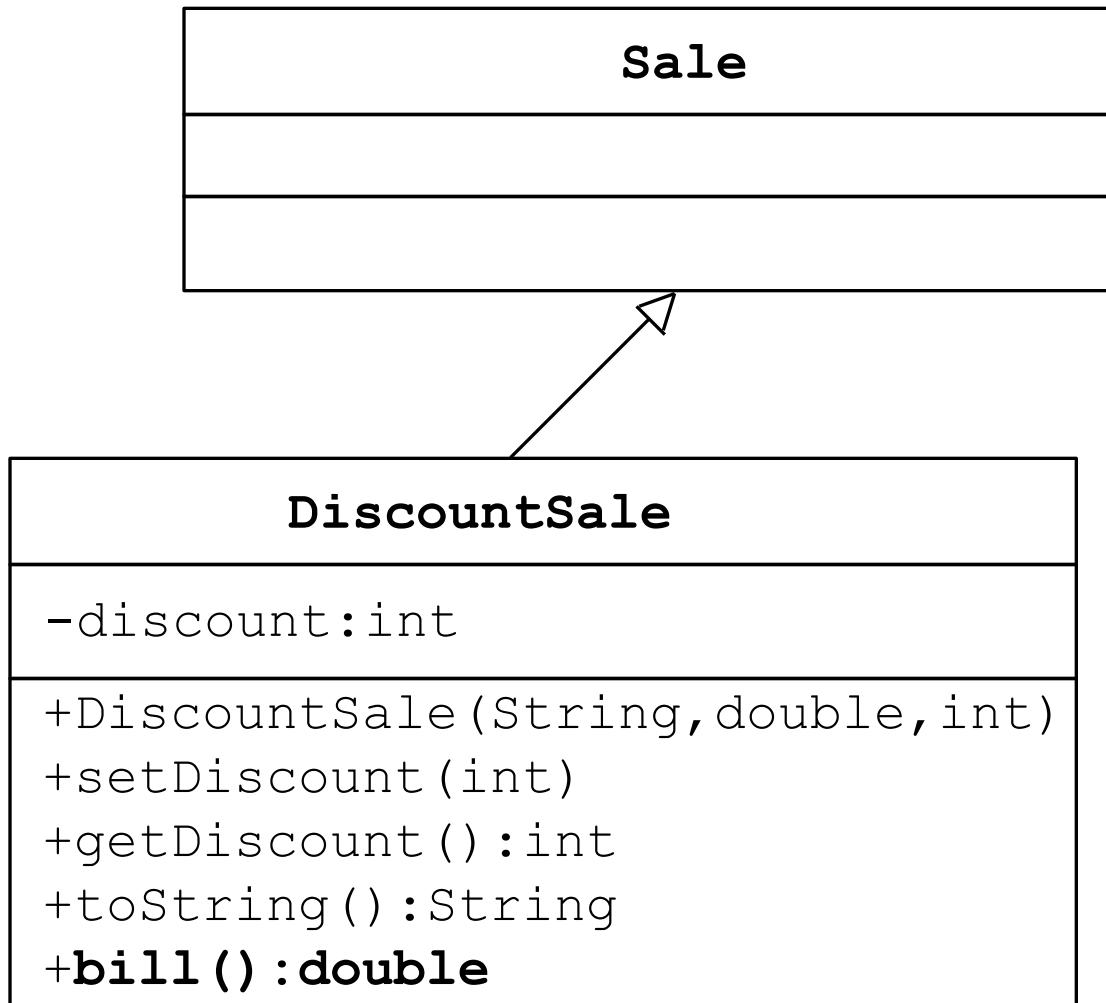
## Late Binding

## Sale

-name :String  
-price:double

+Sale()  
+Sale(String, double)  
+Sale(Sale)  
+setName(String)  
+getName():String  
+setPrice(double)  
+getPrice():double  
+toString():String  
+bill():double  
+equalDeals(Sale):boolean  
+lessThan(Sale):boolean

# Sale class



# DiscountSale class

- 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 and DiscountSale Classes

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

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

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

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

# The Sale and DiscountSale Classes

- Given the following in a program:

...

**Sale simple = new sale("floor mat", 10.00);**

**DiscountSale discount = new**

**DiscountSale("floor mat", 11.00, 10);**

...

**if (discount.lessThan(simple))**

**System.out.println("\$" + discount.bill() +**

**" < \$" + "\$" + simple.bill() +**

**" because late-binding works!");**

...

- Output would be:

**\$9.90 < \$10 because late-binding works!**

# The Sale and DiscountSale Classes

- In the previous example, the **boolean** expression in the **if** statement returns **true**
- As the output indicates, when the **lessThan** method in the **Sale** class is executed, it knows which **bill()** method to invoke
  - The **DiscountSale** class **bill()** method for **discount**, and the **Sale** class **bill()** method for **simple**
- Note that when the **Sale** class was created and compiled, the **DiscountSale** class and its **bill()** method did not yet exist
  - These results are made possible by late-binding

# The Sale and DiscountSale Classes

- *Upcasting is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)*

```
Sale saleVariable; //Base class
```

```
DiscountSale discountVariable = new
```

```
DiscountSale("paint", 15,10); //Derived class
```

```
saleVariable = discountVariable; //Upcasting
```

```
System.out.println(saleVariable.toString());
```

- Because of late binding, **toString** above uses the definition given in the **DiscountSale** class

# Upcasting and Downcasting

- *Downcasting is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)*
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

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

**discountVariable = saleVariable //will produce  
//compiler error**

- There are times, however, when downcasting is necessary, e.g., inside the **equals** method for a class:

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

# Upcasting and Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense
  - The compiler does not check to see if downcasting is a reasonable thing to do
- Using downcasting in a situation that does not make sense usually results in a run-time error

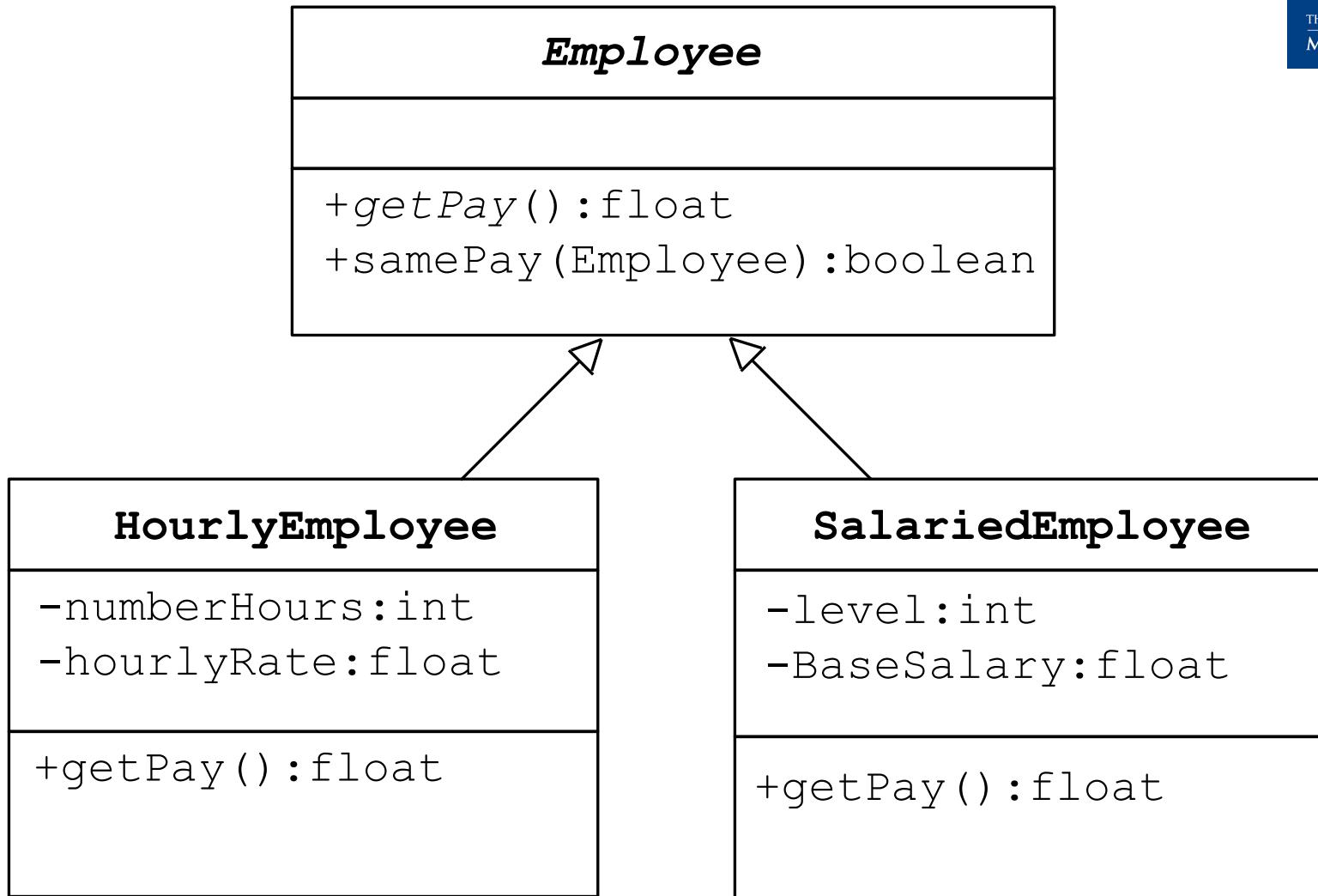
## Pitfall: Downcasting

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **instanceof** operator tests for:  
***object instanceof ClassName***
  - It will return true if ***object*** is of type ***ClassName***
  - In particular, it will return true if ***object*** is an instance of any descendent class of ***ClassName***

**Tip: Checking if Downcasting is Legitimate**

- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

# Outline



# Introduction to Abstract Classes

- In Chapter 7, the **Employee** base class and two of its derived classes, **HourlyEmployee** and **SalariedEmployee** were defined
- The following method is added to the **Employee** class
  - It compares employees to see if they have the same pay:

```
public boolean samePay(Employee other)
{
 return(this.getPay() == other.getPay());
}
```

# Introduction to Abstract Classes

- There are several problems with this method:
  - The **getPay** method is invoked in the **samePay** method
  - There are **getPay** methods in each of the derived classes
  - **There is no getPay** method in the **Employee** class, nor is there any way to define it reasonably without knowing whether the employee is hourly or salaried

## Introduction to Abstract Classes

- The ideal situation would be if there were a way to
  - Postpone the definition of a **getPay** method until the type of the employee were known (i.e., in the derived classes)
  - Leave some kind of note in the **Employee** class to indicate that it was accounted for
- Surprisingly, Java allows this using abstract classes and methods

# Introduction to Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method is called an *abstract class*

# Introduction to Abstract Classes

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **abstract**
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

## Abstract Method

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **abstract** included in its class heading:

```
public abstract class Employee
{
 private instanceVariables;
 .
 .
 .
 public abstract double getPay();
 .
 .
}
```

# Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **abstract** to its modifier
- A class that has no abstract methods is called a **concrete class**

# Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- 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**
  - The constructor in an abstract class is only used by the constructor of its derived classes

## Pitfall: You cannot create Instances of an Abstract Class

- Although an object of an abstract class cannot be created, 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 as it names objects of its concrete descendent classes only

## Tip: An Abstract Class Is a Type

```


 *

TT
TT
TT
TT
```

```

*** *

*** *

```

```


####
```

```
TTTT
TTTT
TT
```

```

 *


```

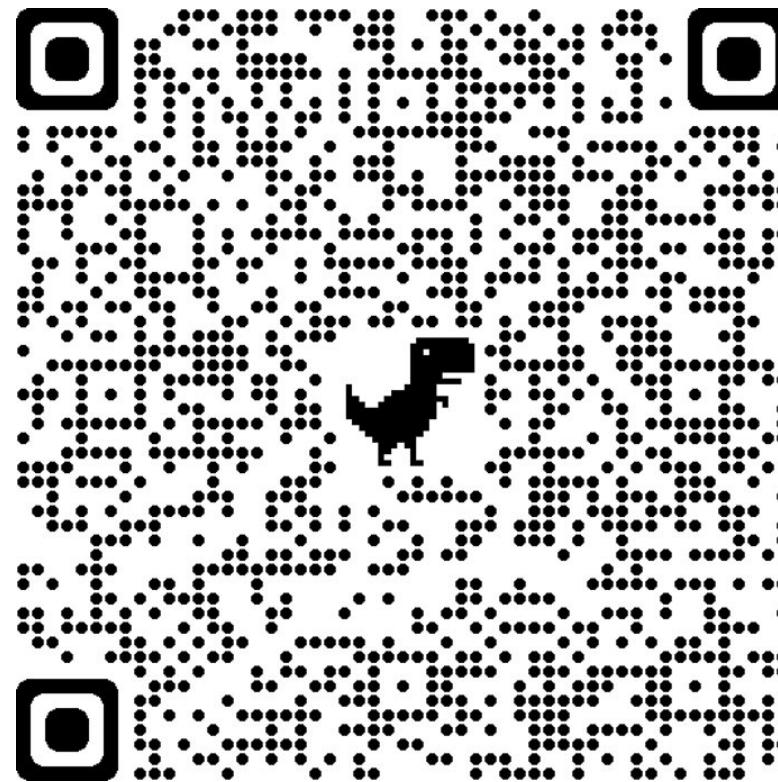


Control your tank by typing a character and press Enter.  
W: up, A: left, S: right, Z: down, Q: quit the game.

# Example – A Simplified Version of The Battle City Tank Game

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

## Class Reflections



<http://go.unimelb.edu.au/5o8i>

## Class Reflections



# Programming and Software Development

## COMP90041

### Lecture 8

# Interfaces & Exception Handling

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- \* the Textbook resources

- **Introduction to UML & Packages & javadoc**
- **Inheritance**
- **Access**
- **Polymorphism**
- **Abstract Classes**

# Review: Week 7

- Interfaces
- Handling Exceptions

# Outline

- **Interfaces**
- Handling Exceptions

# Outline

- 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*
- 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

# Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

# Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected, or package access
- When a class implements an interface, it must make all the methods in the interface public
- 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 any class that implements the interface

# Interfaces

### Display 13.1 The Ordered Interface

```
1 public interface Ordered
2 {
3 public boolean precedes(Object other);
4
5 /**
6 For objects of the class o1 and o2,
7 o1.follows(o2) == o2.preceded(o1).
8
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.

# The Ordered Interface

- To *implement an interface*, a concrete class must do two things:
  1. It must include the phrase **implements Interface Name** at the start of the class definition
    - If more than one interface is implemented, each is listed, separated by commas
  2. The class must implement **all** the method headings listed in the definition(s) of the interface(s)
- Note the use of **Object** as the parameter type in the following examples

# Interfaces

## 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 OrderedHourlyEmployee))
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 in this case. However, either will do for the points being made here.*

# Implementation of an Interface

```
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 }
```

---

# Implementation of an Interface

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

# Abstract Classes Implementing Interfaces

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
20 public abstract boolean follows(Object other);
21 }
```

# Abstract Classes Implementing Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase **extends BaseInterfaceName**
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

# Derived Interfaces

### 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.*

# Extending an Interface

- When a class implements an interface, the compiler and run-time system check the syntax of the interface and its implementation
  - However, neither checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
  - It then becomes the responsibility of each programmer implementing the interface to follow the semantics
- If the method body does not satisfy the specified semantics, then software written for classes that implement the interface may not work correctly

## Pitfall: Interface Semantics Are Not Enforced

- Chapter 6 discussed the Selection Sort algorithm, and examined a method for sorting a partially filled array of type **double** into increasing order
- This code could be modified to sort into decreasing order, or to sort integers or strings instead
  - Each of these methods would be essentially the same, but making each modification would be a nuisance
  - The only difference would be the types of values being sorted, and the definition of the ordering
- Using the **Comparable** interface could provide a single sorting method that covers all these cases

## The Comparable Interface

- 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);**
- It is the programmer's responsibility to follow the semantics of the **Comparable** interface when implementing it

# The Comparable Interface

- 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

## The Comparable Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

## The Comparable Interface Semantics

- The following example reworks the **SelectionSort** class from Chapter 6
- The new version, **GeneralizedSelectionSort**, includes a method that can sort any partially filled array *whose base type implements the Comparable interface*
  - It contains appropriate **indexOfSmallest** and **interchange** methods as well
- Note: Both the **Double** and **String** classes implement the **Comparable** interface
  - Interfaces apply to classes only
  - A primitive type (e.g., **double**) cannot implement an interface

## Using the Comparable Interface

### Display 13.5 Sorting Method for Array of Comparable (Part 1 of 2)

```
1 public class GeneralizedSelectionSort
2 {
3 /**
4 Precondition: numberUsed <= a.length;
5 The first numberUsed indexed variables have values.
6 Action: Sorts a so that a[0], a[1], ... , a[numberUsed - 1] are in
7 increasing order by the compareTo method.
8 */
9 public static void sort(Comparable[] a, int numberUsed)
10 {
11 int index, indexOfNextSmallest;
12 for (index = 0; index < numberUsed - 1; index++)
13 {//Place the correct value in a[index]:
14 indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
15 interchange(index, indexOfNextSmallest, a);
16 //a[0], a[1],..., a[index] are correctly ordered and these are
17 //the smallest of the original array elements. The remaining
18 //positions contain the rest of the original array elements.
19 }
20 }
```

## GeneralizedSelectionSort class: sort Method

**Display 13.5 Sorting Method for Array of Comparable (Part 1 of 2) (continued)**

```
21 /**
22 Returns the index of the smallest value among
23 a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
24 */
25 private static int indexOfSmallest(int startIndex,
26 Comparable[] a, int numberUsed)
27 {
28 Comparable min = a[startIndex];
29 int indexOfMin = startIndex;
30 int index;
31 for (index = startIndex + 1; index < numberUsed; index++)
32 if (a[index].compareTo(min) < 0)//if a[index] is less than min
33 {
34 min = a[index];
35 indexOfMin = index;
36 //min is smallest of a[startIndex] through a[index]
37 }
38 return indexOfMin;
39 }
```

# GeneralizedSelectionSort class: sort Method

### Display 13.5 Sorting Method for Array of Comparable (Part 2 of 2)

```
/**
 * Precondition: i and j are legal indices for the array a.
 * Postcondition: Values of a[i] and a[j] have been interchanged.
 */
private static void interchange(int i, int j, Comparable[] a)
{
 Comparable temp;
 temp = a[i];
 a[i] = a[j];
 a[j] = temp; //original value of a[i]
}
}
```

## GeneralizedSelectionSort class: sort Method

## Display 13.6 Sorting Arrays of Comparable (Part 1 of 2)

```
1 /**
2 Demonstrates sorting arrays for classes that
3 implement the Comparable interface.
4 */
5 public class ComparableDemo The classes Double and String do
6 { implement the Comparable interface.
7 public static void main(String[] args)
8 {
9 Double[] d = new Double[10];
10 for (int i = 0; i < d.length; i++)
11 d[i] = new Double(d.length - i);

12 System.out.println("Before sorting:");
13 int i;
14 for (i = 0; i < d.length; i++)
15 System.out.print(d[i].doubleValue() + ", ");
16 System.out.println();

17 GeneralizedSelectionSort.sort(d, d.length);

18 System.out.println("After sorting:");
19 for (i = 0; i < d.length; i++)
20 System.out.print(d[i].doubleValue() + ", ");
21 System.out.println();
```

# Sorting Arrays of Comparable

### Display 13.6 Sorting Arrays of Comparable (Part 2 of 2)

```
22 String[] a = new String[10];
23 a[0] = "dog";
24 a[1] = "cat";
25 a[2] = "cornish game hen";
26 int numberUsed = 3;

27 System.out.println("Before sorting:");
28 for (i = 0; i < numberUsed; i++)
29 System.out.print(a[i] + ", ");
30 System.out.println();
31
32 GeneralizedSelectionSort.sort(a, numberUsed);
```

# Sorting Arrays of Comparable

## Display 13.6 Sorting Arrays of Comparable (Part 2 of 2) (continued)

```
33 System.out.println("After sorting:");
34 for (i = 0; i < numberUsed; i++)
35 System.out.print(a[i] + ", ");
36 System.out.println();
37 }
38 }
```

**SAMPLE DIALOGUE**

Before Sorting

10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,

After sorting:

1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,

Before sorting;

dog, cat, cornish game hen,

After sorting:

cat, cornish game hen, dog,

# Sorting Arrays of Comparable

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be **public**, **static**, and **final**
  - Because this is understood, Java allows these modifiers to be **omitted**
- Any class that implements the interface has access to these defined constants

## Defined Constants in Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

## Pitfall: Inconsistent Interfaces

- 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
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is **illegal**

## Pitfall: Inconsistent Interfaces

- Interfaces
- Handling Exceptions

# Outline

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

# Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - 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***

# Introduction to Exception Handling

- The basic way of handling exceptions in Java consists of the **try-throw-catch** trio
- The **try** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **try** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
 CodeThatMayThrowAnException
}
```

# try-throw-catch Mechanism

```
... // method code
try
{
 ...
 throw new Exception(StringArgument);
 ...
}
catch(Exception e)
{
 String message = e.getMessage();
 System.out.println(message);
 System.exit(0);
} ...
```

Eg. **ExceptionDemo.java**

## The try-throw-catch Trio

## **throw new**

*ExceptionClassName(PossiblySomeArguments);*

- If exception is thrown, **try** block stops
  - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class

**throw**

- A **throw** statement is similar to a method call:  
**throw new ExceptionClassName(SomeString);**
  - In the above example, the object of class **ExceptionClassName** is created using a string 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 calls a **catch** block

# throw

- When an exception is thrown, the **catch** block begins execution
  - The **catch** block has only **one** parameter
  - The exception object thrown is plugged in for the **catch** block parameter
- The execution of the **catch** block is called *catching the exception*, or *handling the exception*; the catch block is an exception handler
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block

# catch

```
catch(Exception e)
{
 ExceptionHandlingCode
}
```

- A **catch** block looks like a method definition that has a parameter of type ***Exception*** class
  - It is not really a method definition

catch

## `catch(Exception e) { ... }`

- ***e is called the catch block parameter***
- The **catch** block parameter does two things:
  - It specifies the type of thrown exception object that the **catch** block can catch (e.g., an **Exception** class object above)
  - It provides a name (for the thrown object that is caught) on which it can operate in the **catch** block
    - Note: The identifier **e** is often used by convention, but any non-keyword identifier can be used

catch

- When a **try** block is executed, two things can happen:
  1. No exception is thrown in the **try** block
    - The code in the **try** block is executed to the end of the block
    - The **catch** block is skipped
    - The execution continues with the code placed after the **catch** block

## try-throw-catch Mechanism

2. An exception is thrown in the **try** block and caught in the **catch** block

- The rest of the code in the **try** block is skipped
- Control is transferred to a following **catch** block (in simple cases)
- The thrown object is plugged in for the **catch** block parameter
- The code in the **catch** block is executed
- The code that follows that **catch** block is executed (if any)

## try-throw-catch Mechanism

```
... // method code
try
{
 ...
 throw new Exception(StringArgument);
 ...
}
catch(Exception e)
{
 String message = e.getMessage();
 System.out.println(message);
 System.exit(0);
} ...
```

## Using the 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

## Using the getMessage Method

- There are more exception classes than just the single class **Exception**
  - There are more exception classes in the standard Java libraries
  - New exception classes can be defined like any other class
- 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**

# Exception Classes

- The predefined exception class **Exception** is the root class for all exceptions
  - 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
  - The class **Exception** is in the **java.lang** package, and so requires no **import** statement

## Exception Classes from Standard Packages

- 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;**

## Exception Classes from Standard Packages

- A **throw** statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the **catch** block
  - A different type of exception can be defined to identify each different exceptional situation

## Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- 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

## Defining Exception Classes

### Display 9.3 A Programmer-Defined Exception Class

---

```
1 public class DivisionByZeroException extends Exception
2 {
3 public DivisionByZeroException() You can do more in an exception
4 { constructor, but this form is common.
5 super("Division by Zero!");
6 }
7
7 public DivisionByZeroException(String message)
8 {
9 super(message); super is an invocation of the constructor for
10 } the base class Exception.
11 }
```

---

### DivisionByZeroException.java

# A Programmer-Defined Exception Class

- An exception class constructor can be defined that takes an argument of another type
  - It would store its value in an instance variable
  - It would need to define accessor methods for this instance variable

**Tip: An Exception Class Can Carry a Message of Any Type: int Message**

**Display 9.5 An Exception Class with an int Message**

```
1 public class BadNumberException extends Exception
2 {
3 private int badNumber;
4
5 public BadNumberException(int number)
6 {
7 super("BadNumberException");
8 badNumber = number;
9
10 public BadNumberException()
11 {
12 super("BadNumberException");
13
14 public BadNumberException(String message)
15 {
16 super(message);
17
18 public int getBadNumber()
19 {
20 return badNumber;
21 }
```

# An Exception Class with an int Message

- The two most important things about an exception object are its **type** (i.e., exception class) and the **message** it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method **getMessage**, so that the catch block can use the message

# Exception Object Characteristics

- Must be a derived class of an already existing exception class
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method **getMessage** is inherited

## Programmer-Defined Exception Class Guidelines

- For all predefined exception classes, **getMessage** returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class, two constructors must be included:
  - 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

## Preserve getMessage

- A **try** block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a **try** block, at most one exception can be thrown (since a throw statement ends the execution of the **try** block)
  - However, different types of exception values can be thrown on different executions of the **try** block

## Multiple catch Blocks

- Each **catch** block can only catch values of the exception class type given in the **catch** block heading
- 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

## Multiple catch Blocks

- When catching multiple exceptions, the order of the **catch** blocks is important
  - When an exception is thrown in a **try** block, the **catch** blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

## Pitfall: Catch the More Specific Exception First

```
catch (Exception e)
{ ... }
catch (NegativeNumberException e)
{ ... }
```

- Because a **NegativeNumberException** is a type of **Exception**, all **NegativeNumberExceptions** will be caught by the first **catch** block before ever reaching the second block
  - The catch block for **NegativeNumberException** will never be used!
- For the correct ordering, simply reverse the two blocks

### ExceptionDemo.java

## Pitfall: Catch the More Specific Exception First

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a **try** block, and catch the exception in a **catch** block that follows
- In this case, the method itself would not include **try** and **catch** blocks
  - However, it would have to include a **throws** clause

# Throwing an Exception in a Method

- If a method can throw an exception but does not catch it, it must provide a warning in the heading
  - This warning is called a **throws clause**
  - The process of including an exception class in a throws clause is called *declaring the exception*  
**throws AnException //throws clause**
  - The following states that an invocation of **aMethod** could throw **AnException**  
**public void aMethod() throws AnException**
- If a method throws an exception and does not catch it, then the method invocation ends immediately

## Declaring Exceptions in a throw Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

**public void aMethod() throws**

**AnException, AnotherException**

## Declaring Exceptions in a throw Clause

- Here is an example of a method from which the exception originates:

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

## Defining Exceptions in a Method

- When **someMethod** is used by an **otherMethod**, the **otherMethod** must then deal with the exception:

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

## ExceptionDemo2.java

# Handling Exceptions in another Method

- Two ways of handling exceptions thrown in a method:
  1. The code that can throw an exception is placed within a **try** block, and the possible exception is caught in a **catch** block within the same method
  2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **throws** clause

## The Catch or Declare Rule: Two ways

- The first technique handles an exception in a **catch** block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a **catch** block in some method that does not just declare the exception class in a **throws** clause

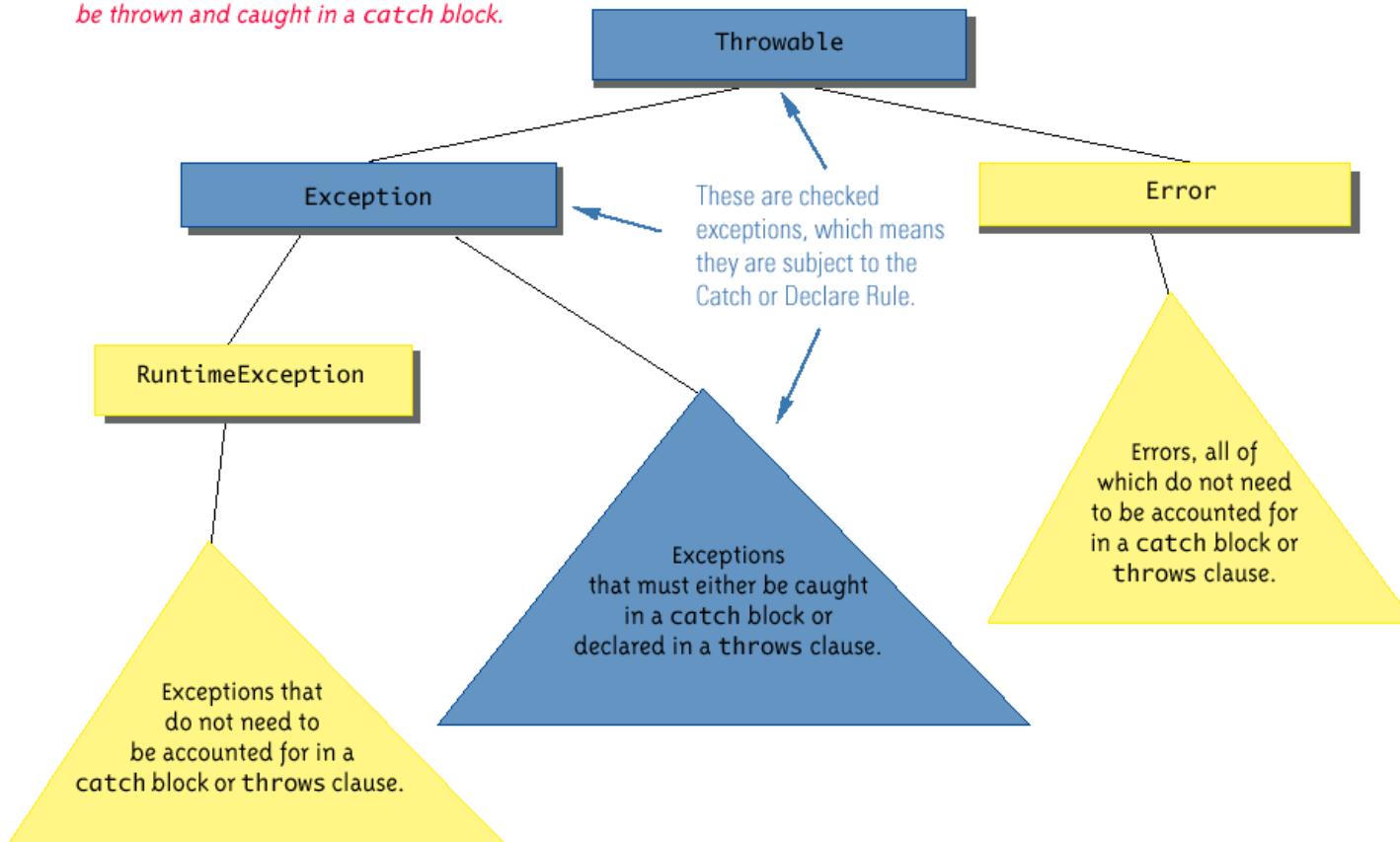
## The Catch and Declare Rule: An Exception must be handled somewhere

- Two techniques can be **mixed**
  - Some exceptions may be caught, and others may be declared in a **throws** clause
- However, these techniques must be used consistently with a given exception
  - If an exception is not declared, then it must be handled within the method
  - If an exception is declared, then the responsibility for handling it is shifted to some other calling method
  - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

## The Catch and Declare Rule: Mixed Usage

## Display 9.10 Hierarchy of Throwable Objects

All descendants of the class `Throwable` can be thrown and caught in a catch block.



# 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

## Checked and Unchecked Exceptions

- 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 **throws** Clause in Derived Classes

- If every method up to and including the main method simply includes a **throws** clause for an exception, that exception may be thrown but never caught
  - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable
  - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a **catch** block in some method

What happens if  
an exception is never caught?

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- How exceptions are handled depends on how a method is called.

## When to use Exceptions?

- Exception handling is an example of a programming methodology known as *event-driven programming*
- When using event-driven programming, objects are defined so that they send events to other objects that handle the events
  - An event is an object also
  - Sending an event is called *firing an event*

# Event Driven Programming

- In exception handling, the event objects are the exception objects
  - They are fired (thrown) by an object when the object invokes a method that throws the exception
  - An exception event is sent to a **catch** block, where it is handled

# Event Driven Programming

- It is possible to place a **try** block and its following catch blocks inside a larger **try** block, or inside a larger **catch** block
  - If a set of **try-catch** blocks are placed inside a larger **catch** block, **different names** must be used for the **catch** block parameters in the inner and outer blocks, just like any other set of nested blocks
  - If a set of **try-catch** blocks are placed inside a larger **try** block, and an exception is thrown in the **inner try** block that is **not caught**, then the exception is thrown to the **outer try** block for processing, and may be caught in one of its **catch** blocks

## Pitfall: Nested try-catch Blocks

- The **finally** block contains code to be executed whether or not an exception is thrown in a **try** block
  - If it is used, a **finally** block is placed after a **try** block and its following **catch** blocks

```
try
{
 ...
}
catch(ExceptionClass1 e)
{
 ...
}

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

# The finally Block

- If the **try-catch-finally** blocks are inside a method definition, there are three possibilities when the code is run:
  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

## The finally Block

- When a program contains an assertion check, and the assertion check fails, an object of the class **AssertionError** is thrown
  - This causes the program to end with an error message
- The class **AssertionError** is derived from the class **Error**, and therefore is an unchecked Throwable
  - In order to prevent the program from ending, it could be handled, but this is not required

# The AssertionError 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

## Exception Handling with the Scanner Class

- 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

# The InputMismatchException

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
 try
 {
 CodeThatMayThrowAnException
 done = true;
 }
 catch (SomeExceptionClass e)
 {
 SomeMoreCode
 }
}
```

## Tip: Exception Controlled Loops

### Display 9.11 An Exception Controlled Loop

```
1 import java.util.Scanner;
2 import java.util.InputMismatchException;
3
4 public class InputMismatchExceptionDemo
5 {
6 public static void main(String[] args)
7 {
8 Scanner keyboard = new Scanner(System.in);
9 int number = 0; //to keep compiler happy
10 boolean done = false;
```

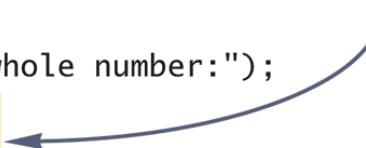
(continued)

# An Exception Controlled Loop (Part 1 of 3)

## Display 9.11 An Exception Controlled Loop

```
10 while (! done)
11 {
12 try
13 {
14 System.out.println("Enter a whole number:");
15 number = keyboard.nextInt();
16 done = true;
17 }
18 catch(InputMismatchException e)
19 {
20 keyboard.nextLine();
21 System.out.println("Not a correctly written whole number.");
22 System.out.println("Try again.");
23 }
24 }
25
26 System.out.println("You entered " + number);
27 }
```

If `nextInt` throws an exception, the try block ends and so the boolean variable `done` is not set to `true`.



(continued)

## An Exception Controlled Loop (Part 2 of 3)

## Display 9.11 An Exception Controlled Loop

### SAMPLE DIALOGUE

Enter a whole number:

**forty two**

Not a correctly written whole number.

Try again.

Enter a whole number:

**fortytwo**

Not a correctly written whole number.

Try again.

Enter a whole number:

**42**

You entered 42

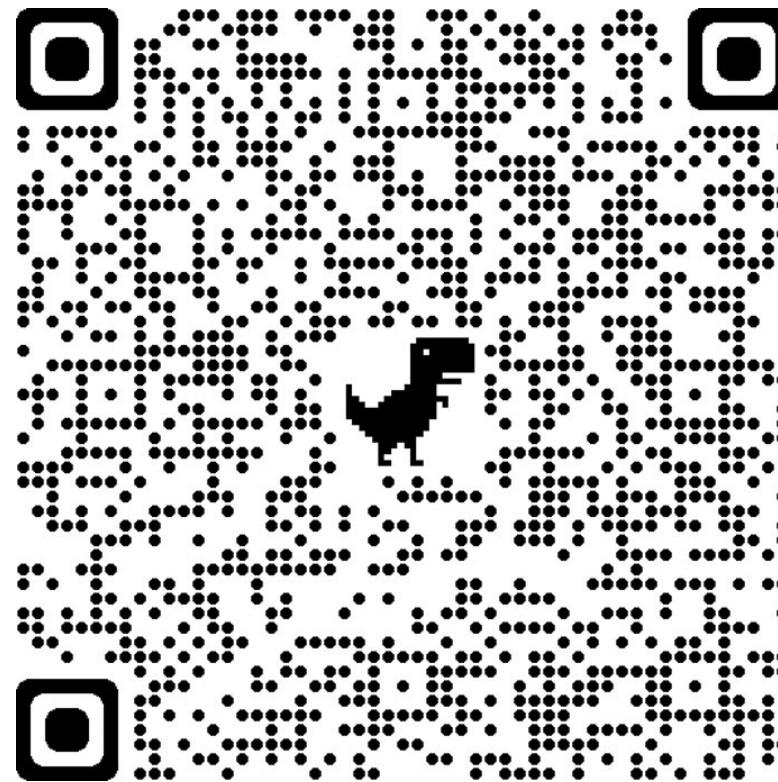
# An Exception Controlled Loop (Part 3 of 3)

- An **ArrayIndexOutOfBoundsException** is thrown whenever a program attempts to use an array index that is out of bounds
  - This normally causes the program to end
- Like all other descendants of the class **RuntimeException**, it is an unchecked exception
  - There is no requirement to handle it
- When this exception is thrown, it is an indication that the program contains an error
  - Instead of attempting to handle the exception, the program should simply be fixed

# ArrayIndexOutOfBoundsException

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

## Class Reflections



[http://go.unimelb.edu.au/5o8i.](http://go.unimelb.edu.au/5o8i)

## Class Reflections



# Programming and Software Development

## COMP90041

### Lecture 10

# File I/O

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- \* the Textbook resources

- Interfaces
- Handling Exceptions

# Review: Week 8-9

- File I/O
- Buffered Reader
- Binary Files

# Outline

- **File I/O**
- **Buffered Reader**
- **Binary Files**

# Outline

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - If the data flows into a program, then the stream is called an *input stream*
  - If the data flows out of a program, then the stream is called an *output stream*

# Streams

- Input streams can flow from the keyboard or from a file
  - **System.in is an input stream that connects to the keyboard**  
**Scanner keyboard = new Scanner(System.in);**
- Output streams can flow to a screen or to a file
  - **System.out is an output stream that connects to the screen**  
**System.out.println("Output stream");**

# Streams

- Files that are designed to be read by human beings, and that can be read or written with an editor are called ***text files***
  - An advantage of text files is that they are usually the same on all computers, so that they can move from one computer to another
- Files that are designed to be read by programs and that consist of a sequence of binary digits are called ***binary files***
  - An advantage of binary files is that they are ***more efficient to process*** than text files

## Text Files and Binary Files

- The class **PrintWriter** is a stream class that can be used to write to a text file
  - An object of the class **PrintWriter** has the methods **print**, **println**, and **printf**
  - These are similar to the **System.out** methods of the same names, but are used for text file output, not screen output

## Writing to a Text File: PrintWriter

- All the file I/O classes that follow are in the package **java.io**, so a program that uses **PrintWriter** will start with a set of **import** statements:  
**import java.io.PrintWriter;**  
**import java.io.FileOutputStream;**  
**import java.io.FileNotFoundException;**

## Writing to a Text File: imports

- A stream of the class **PrintWriter** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
FileOutputStream(fileName));
```

- The class **FileOutputStream** takes a string representing the file name as its argument
- The class **PrintWriter** takes the anonymous **FileOutputStream** object as its argument

## Writing to a Text File: syntax

- This produces an object of the class **PrintWriter** that is connected to the file **FileName** (Eg. **TextFileOutputDemo**)
  - The process of connecting a stream to a file is called *opening the file*
  - If the file already exists, then doing this causes the old contents to be lost
  - If the file does not exist, then a new, empty file named **FileName** is created
- After doing this, the methods **print** and **println** can be used to write to the file

## Writing to a Text File: open a file

- When a text file is opened in this way, a **FileNotFoundException** can be thrown
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file
- It is therefore necessary to enclose this code in exception handling blocks
  - The file should be opened inside a **try** block
  - A **catch** block should catch and handle the possible exception
  - The variable that refers to the **PrintWriter** object should be declared outside the block (and initialized to **null**) so that it is not local to the block

## Writing to a Text File: exceptions

- When a program is finished writing to a file, it should always close the stream connected to that file

***outputStreamName.close();***

- This allows the system to release any resources used to connect the stream to the file
- If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

## Writing to a Text File: close a file

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method **flush** is invoked, the buffered data is written to the file all at once
  - This is more **efficient**, since physical writes to a file can be slow

## Writing to a Text File: buffer

- The method **close** invokes the method **flush**, thus insuring that all the data is written to the file
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway
  - The sooner a file is closed after writing to it, the less likely it is that there will be a problem

## Writing to a Text File: **close** & **flush**

- The rules for how file names should be formed depend on a given operating system, not Java
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., `"fileName.txt"`)
  - Any suffix used, such as `.txt` has no special meaning to a Java program

# File Names

- Every input file and every output file used by a Java program has two names:
  1. The real file name used by the operating system
  2. The name of the stream that is connected to the file
- The actual file name is used to connect to the stream
- The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

## A File Has Two Names

- When performing file I/O there are many situations in which an exception, such as **FileNotFoundException**, may be thrown
- Many of these exception classes are subclasses of the class **IOException**
  - The class **IOException** is the root class for a variety of exception classes having to do with input and/or output
- These exception classes are all checked exceptions
  - Therefore, they must be caught or declared in a throws clause

# IOException

- In contrast, the exception classes **NoSuchElementException**, **InputMismatchException**, and **IllegalStateException** are all unchecked exceptions
  - Unchecked exceptions are not required to be caught or declared in a throws clause

# Unchecked Exceptions

- Since opening a file can result in an exception, it should be placed inside a **try** block
- If the variable for a **PrintWriter** object needs to be used outside that block, then the variable must be declared outside the block
  - Otherwise it would be local to the block, and could not be used elsewhere
  - If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier

## Pitfall: a try Block is a Block

- To create a **PrintWriter** object and connect it to a text file for *appending*, a second argument, set to **true**, must be used in the constructor for the **FileOutputStream** object

```
outputStreamName = new PrintWriter(new
FileOutputStream(FileName, true));
```

- After this statement, the methods **print**, **println** and/or **printf** can be used to write to the file
- The new text will be written *after the old text* in the file

## Appending to a Text File

- If a class has a suitable `toString()` method, and `anObject` is an object of that class, then `anObject` can be used as an argument to `System.out.println`, and it will produce sensible output
- The same thing applies to the methods `print` and `println` of the class `PrintWriter`  
`outputStreamName.println(anObject);`

## toString Helps with Text File Output

## Display 10.2 Some Methods of the Class PrintWriter

PrintWriter and FileOutputStream are in the `java.io` package.

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new PrintWriter(new FileOutputStream(File_Name))
```

When the constructor is used in this way, a blank file is created. If there already was a file named `File_Name`, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
new PrintWriter(new FileOutputStream(File_Name, true))
```

(For an explanation of the argument `true`, read the subsection "Appending to a Text File.")

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the `File_Name`. (The `File` class will be covered in Section 10.3. We discuss it here so that you will have a more complete reference in this display, but you can ignore the reference to the class `File` until after you've read that section.)

(continued)

# Some Methods of the PrintWriter (Part 1 of 3)

## Display 10.2 Some Methods of the Class PrintWriter

```
public void println(Argument)
```

The *Argument* can be a string, character, integer, floating-point number, boolean value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

```
public void print(Argument)
```

This is the same as `println`, except that this method does not end the line, so the next output will be on the same line.

(continued)

# Some Methods of the PrintWriter (Part 2 of 3)

## Display 10.2 Some Methods of the Class PrintWriter

```
public PrintWriter printf(Arguments)
```

This is the same as `System.out.printf`, except that this method sends output to a text file rather than to the screen. It returns the calling object. However, we have always used `printf` as a void method.

```
public void close()
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush()
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

# Some Methods of the PrintWriter (Part 3 of 3)

- The class **Scanner** can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument **System.in** (to the **Scanner** constructor) with a suitable stream that is connected to the text file
- **Scanner StreamObject =  
new Scanner(new FileInputStream(FileName));**
- Methods of the **Scanner** class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the **nextInt** and **nextLine** methods

## Reading From a Text File Using Scanner

### Display 10.3 Reading Input from a Text File Using Scanner

```
1 import java.util.Scanner;
2 import java.io.FileInputStream;
3 import java.io.FileNotFoundException;
4
5 public class TextFileScannerDemo
6 {
7 public static void main(String[] args)
8 {
9 System.out.println("I will read three numbers and a line");
10 System.out.println("of text from the file morestuff.txt.");
11
12 Scanner inputStream = null;
13
14 try
15 {
16 inputStream =
17 new Scanner(new FileInputStream("morestuff.txt"));
18 }
```

(continued)

## Example (Part 1 of 4)

### Display 10.3 Reading Input from a Text File Using Scanner

```
19 catch(FileNotFoundException e)
20 {
21 System.out.println("File morestuff.txt was not found");
22 System.out.println("or could not be opened.");
23 System.exit(0);
24 }
25 int n1 = inputStream.nextInt();
26 int n2 = inputStream.nextInt();
27 int n3 = inputStream.nextInt();
28
29 inputStream.nextLine(); //To go to the next line
30
31 String line = inputStream.nextLine();
32
```

(continued)

## Example (Part 2 of 4)

### Display 10.3 Reading Input from a Text File Using Scanner

```
33 System.out.println("The three numbers read from the file are:");
34 System.out.println(n1 + ", " + n2 + ", and " + n3);
35
36 System.out.println("The line read from the file is:");
37 System.out.println(line);
38
39 inputStream.close();
40 }
41 }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

*This file could have been made with a  
text editor or by another Java  
program.*

(continued)

## Example (Part 3 of 4)

### Display 10.3    Reading Input from a Text File Using Scanner

#### SCREEN OUTPUT

I will read three numbers and a line  
of text from the file morestuff.txt.  
The three numbers read from the file are:  
1, 2, and 3  
The line read from the file is:  
Eat my shorts.

## Example (Part 4 of 4)

- A program that tries to read beyond the end of a file using methods of the **Scanner** class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the **Scanner** class provides methods such as **hasNextInt** and **hasNextLine**
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

## Testing for the End of a Text File

**Display 10.4 Checking for the End of a Text File with hasNextLine**

```
1 import java.util.Scanner;
2 import java.io.FileInputStream;
3 import java.io.FileNotFoundException;
4 import java.io.PrintWriter;
5 import java.io.FileOutputStream;
6
7 public class HasNextLineDemo
8 {
9 public static void main(String[] args)
10 {
11 Scanner inputStream = null;
12 PrintWriter outputStream = null;
```

(continued)

# Checking for the end of a Text File with hasNextLine method (Part 1 of 4)

**Display 10.4 Checking for the End of a Text File with hasNextLine**

```
13 try
14 {
15 inputStream =
16 new Scanner(new FileInputStream("original.txt"));
17 outputStream = new PrintWriter(
18 new FileOutputStream("numbered.txt"));
19 }
20 catch(FileNotFoundException e)
21 {
22 System.out.println("Problem opening files.");
23 System.exit(0);
24 }

25 String line = null;
26 int count = 0;
```

(continued)

# Checking for the end of a Text File with hasNextLine method (Part 2 of 4)

## Display 10.4 Checking for the End of a Text File with hasNextLine

```
27 while (inputStream.hasNextLine())
28 {
29 line = inputStream.nextLine();
30 count++;
31 outputStream.println(count + " " + line);
32 }
33 inputStream.close();
34 outputStream.close();
35 }
36 }
```

(continued)

# Checking for the end of a Text File with hasNextLine method (Part 3 of 4)

### Display 10.4 Checking for the End of a Text File with hasNextLine

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

## Checking for the end of a Text File with hasNextLine method (Part 4 of 4)

### Display 10.5 Checking for the End of a Text File with hasNextInt

```
1 import java.util.Scanner;
2 import java.io.FileInputStream;
3 import java.io.FileNotFoundException;
4
5 public class HasNextIntDemo
6 {
7 public static void main(String[] args)
8 {
9 Scanner inputStream = null;
10
11 try
12 {
13 inputStream =
14 new Scanner(new FileInputStream("data.txt"));
15 }
16 catch(FileNotFoundException e)
17 {
18 System.out.println("File data.txt was not found");
19 System.out.println("or could not be opened.");
20 System.exit(0);
21 }
22 }
23 }
```

(continued)

## Checking for the end of a Text File with hasNextInt method (Part 1 of 2)

### Display 10.5 Checking for the End of a Text File with hasNextInt

```
20 int next, sum = 0;
21 while (inputStream.hasNextInt())
22 {
23 next = inputStream.nextInt();
24 sum = sum + next;
25 }
26
27 inputStream.close();
28
29 }
System.out.println("The sum of the numbers is " + sum);
```

File data.txt

1 2  
3 4 hi 5

Reading ends when either the end of the file is reached or a token that is not an int is reached. So, the 5 is never read.

#### SCREEN OUTPUT

The sum of the numbers is 10

## Checking for the end of a Text File with hasNextInt method (Part 2 of 2)

- File I/O
- **Buffered Reader**
- Binary Files

# Outline

- The class **BufferedReader** is a stream class that can be used to read from a text file
  - An object of the class **BufferedReader** has the methods **read** and **readLine**
- A program using **BufferedReader**, like one using **PrintWriter**, will start with a set of **import** statements:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

# Introduction to BufferedReader

- Like the classes **PrintWriter** and **Scanner**, **BufferedReader** has no constructor that takes a file name as its argument
  - It needs to use another class, **FileReader**, to convert the file name to an object that can be used as an argument to its (the **BufferedReader**) constructor
- A stream of the class **BufferedReader** is created and connected to a text file as follows:  
**BufferedReader readerObject;**  
**readerObject = new BufferedReader(new**  
**FileReader(fileName));**
  - This opens the file for reading

# Introduction to BufferedReader

- After these statements, the methods **read** and **readLine** can be used to read from the file
  - The **readLine** method is the same method used to read from the keyboard, but in this case it would read from a file
  - The **read** method reads a single character, and returns a value (of type **int**) that corresponds to the character read
  - Since the read method does not return the character itself, a type cast must be used:  
**char next = (char)(readerObject.read());**

# Reading From a Text File

## Display 10.7 Reading Input from a Text File Using BufferedReader

```
1 import java.io.BufferedReader;
2 import java.io.FileReader;
3 import java.io.FileNotFoundException;
4 import java.io.IOException;

5 public class TextInputDemo
6 {
7 public static void main(String[] args)
8 {
9 try
10 {
11 BufferedReader inputStream =
12 new BufferedReader(new FileReader("morestuff2.txt"));

13 String line = inputStream.readLine();
14 System.out.println(
15 "The first line read from the file is:");
16 System.out.println(line);
```

(continued)

# Example (Part 1 of 3)

### Display 10.7    Reading Input from a Text File Using BufferedReader

```
17
18 line = inputStream.readLine();
19 System.out.println(
20 "The second line read from the file is:");
21 System.out.println(line);
22 inputStream.close();
23 }
24 catch(FileNotFoundException e)
25 {
26 System.out.println("File morestuff2.txt was not found");
27 System.out.println("or could not be opened.");
28 }
29 catch(IOException e)
30 {
31 System.out.println("Error reading from morestuff2.txt.");
32 }
33 }
34 }
```

(continued)

## Example (Part 2 of 3)

## Display 10.7 Reading Input from a Text File Using BufferedReader

File morestuff2.txt

```
1 2 3
Jack jump over
the candle stick.
```

*This file could have been made with a text editor or by another Java program.*

### SCREEN OUTPUT

The first line read from the file is:

```
1 2 3
```

The second line read from the file is:

```
Jack jump over
```

## Example (Part 3 of 3)

- A program using a **BufferedReader** object in this way may throw two kinds of exceptions
  - An attempt to open the file may throw a **FileNotFoundException** (which in this case has the expected meaning)
  - An invocation of **readLine** may throw an **IOException**
  - Both of these exceptions should be handled

## Reading from a Text File

## Display 10.8 Some Methods of the Class BufferedReader

BufferedReader and FileReader are in the `java.io` package.

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new BufferedReader(new FileReader(File_Name))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

The `File` class will be covered in the section entitled “The `File` Class.” We discuss it here so that you will have a more complete reference in this display, but you can ignore the following reference to the class `File` until after you’ve read that section.

If you want to create a stream using an object of the class `File`, you use

```
new BufferedReader(new FileReader(File_Object))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

(continued)

# Some Methods of BufferedReader (Part 1 of 2)

### Display 10.8 Some Methods of the Class BufferedReader

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, null is returned. (Note that an EOFException is not thrown at the end of a file. The end of a file is signaled by returning null.)

```
public int read() throws IOException
```

Reads a single character from the input stream and returns that character as an int value. If the read goes beyond the end of the file, then -1 is returned. Note that the value is returned as an int. To obtain a char, you must perform a type cast on the value returned. The end of a file is signaled by returning -1. (All of the "real" characters return a positive integer.)

```
public long skip(long n) throws IOException
```

Skips n characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

# Some Methods of BufferedReader (Part 1 of 2)

- Unlike the **Scanner** class, the class **BufferedReader** has no methods to read a number from a text file
  - Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
  - To read in a single number on a line by itself, first use the method **readLine**, and then use **Integer.parseInt**, **Double.parseDouble**, etc. to convert the string into a number
  - If there are multiple numbers on a line, **StringTokenizer** can be used to decompose the string into tokens, and then the tokens can be converted as described above

# Reading Numbers

- The method **readLine** of the class **BufferedReader** returns **null** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **null** when using **readLine**
- The method **read** of the class **BufferedReader** returns **-1** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **-1** when using **read**

## Testing for the End of a Text File

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

## Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the program is located

# Path Names

- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is  
**"/user/sallyz/data/data.txt"**
  - A **BufferedReader** input stream connected to this file is created as follows:  
**BufferedReader inputStream =  
new BufferedReader(new  
FileReader("/user/sallyz/data/data.txt"));**

# Path Names

- The Windows operating system specifies path names in a different way
  - A typical Windows path name is the following:  
**C:\dataFiles\goodData\data.txt**
  - A **BufferedReader** input stream connected to this file is created as follows:  
**BufferedReader inputStream = new  
BufferedReader(new FileReader  
("C:\\dataFiles\\goodData\\\\data.txt"));**
  - Note that in Windows \\ must be used in place of \, since a single backslash denotes an the beginning of an escape sequence

## Path Names

- A double backslash (\\\) must be used for a Windows path name enclosed in a quoted string
  - This problem does not occur with path names read in from the keyboard
- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

## Path Names

- Each of the Java I/O library classes serves only one function, or a small number of functions
  - Normally two or more class constructors are combined to obtain full functionality
- Therefore, expressions with two constructors are common when dealing with Java I/O classes

## Nested Constructor Invocations

## **new BufferedReader(new FileReader("stuff.txt"))**

- Above, the anonymous **FileReader** object establishes a connection with the **stuff.txt** file
  - However, it provides only very primitive methods for input
- The constructor for **BufferedReader** takes this **FileReader** object and adds a richer collection of input methods
  - This transforms the inner object into an instance variable of the outer object

# Nested Constructor Invocations

- The standard streams **System.in**, **System.out**, and **System.err** are automatically available to every Java program
  - **System.out** is used for normal screen output
  - **System.err** is used to output error messages to the screen
- The **System** class provides three methods (**setIn**, **setOut**, and **setErr**) for redirecting these standard streams:

```
public static void setIn(InputStream inStream)
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
```

# System.in, System.out, and System.err

- Using these methods, any of the three standard streams can be redirected
  - For example, instead of appearing on the screen, error messages could be redirected to a file
- In order to redirect a standard stream, a new stream object is created
  - Like other streams created in a program, a stream object used for redirection must be closed after I/O is finished
  - Note, standard streams do not need to be closed

## System.in, System.out, and System.err

- Redirecting **System.err**:

```
public void getInput()
{
 ...
 PrintStream errStream = null;
 try
 {
 errStream = new PrintStream(new
 FileOutputStream("errMessages.txt"));
 System.setErr(errStream);
 ... //Set up input stream and read
 }
```

## System.in, System.out, and System.err

- Redirecting **System.err**:

```
public void getInput() {
 ...
 PrintStream errStream = null;
 try {
 errStream = new PrintStream(new
 FileOutputStream("errMessages.txt"));
 System.setErr(errStream);
 ... //Set up input stream and read
 } catch (FileNotFoundException e) {
 System.err.println("Input file not found");
 } finally {
 ...
 errStream.close();
 }
}
```

## System.in, System.out, and System.err

- The **File** class is like a wrapper class for file names
  - The constructor for the class **File** takes a name, (known as the *abstract name*) as a string argument, and produces an object that represents the file with that name
  - The **File** object and methods of the class **File** can be used to determine information about the file and its properties
  - Eg. **FileClassDemo**

# The File Class

## Display 10.12 Some Methods in the Class File

File is in the java.io package.

```
public File(String File_Name)
```

Constructor. *File\_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File\_Name* is referred to as the **abstract path name**.

```
public boolean exists()
```

Tests whether there is a file with the abstract path name.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns true if the file named by the abstract path name exists and is readable by the program; otherwise returns false.

(continued)

# Some Methods of the Class File (Part 1 of 5)

## Display 10.12 Some Methods in the Class File

```
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

```
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

(continued)

# Some Methods of the Class File (Part 2 of 5)

## Display 10.12 Some Methods in the Class File

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns true if successful, and returns false otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a String value.

```
public boolean renameTo(File New_Name)
```

Renames the file represented by the abstract path name to *New\_Name*. Returns true if successful; otherwise returns false. *New\_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

# Some Methods of the Class File (Part 3 of 5)

### Display 10.12 Some Methods in the Class File

```
public boolean isFile()
```

Returns true if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns false. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

```
public boolean isDirectory()
```

Returns true if a directory (folder) exists that is named by the abstract path name; otherwise returns false.

(continued)

## Some Methods of the Class File (Part 4 of 5)

## Display 10.12 Some Methods in the Class File

```
public boolean mkdir()
```

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

```
public boolean mkdirs()
```

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

```
public long length()
```

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.

# Some Methods of the Class File (Part 5 of 5)

- File I/O
- Buffered Reader
- **Binary Files**

# Outline

- Binary files store data in the same format used by computer memory to store the values of variables
  - No conversion needs to be performed when a value is stored or retrieved from a binary file
- Java binary files, unlike other binary language files, are portable
  - A binary file created by a Java program can be moved from one computer to another
  - These files can then be read by a Java program, but only by a Java program

# Binary Files

- The class **ObjectOutputStream** is a stream class that can be used to write to a binary file
  - An object of this class has methods to write strings, values of primitive types, and objects to a binary file
- A program using **ObjectOutputStream** needs to import several classes from package **java.io**:  
**import java.io.ObjectOutputStream;**  
**import java.io.FileOutputStream;**  
**import java.io.IOException;**

## Writing Simple Data to a Binary File

- An **ObjectOutputStream** object is created and connected to a binary file as follows:

```
ObjectOutputStream outputStreamName = new
ObjectOutputStream(new
FileOutputStream(fileName));
```

- The constructor for **FileOutputStream** may throw a **FileNotFoundException**
- The constructor for **ObjectOutputStream** may throw an **IOException**
- Each of these must be handled

## Opening a Binary File for Output

- After opening the file, **ObjectOutputStream** methods can be used to write to the file
  - Methods used to output primitive values include **writeInt**, **writeDouble**, **writeChar**, and **writeBoolean**
- *UTF is an encoding scheme used to encode Unicode characters that favors the ASCII character set*
  - The method **writeUTF** can be used to output values of type **String**
- The stream should always be closed after writing

## Opening a Binary File for Output

## Display 10.14 Some Methods in the Class ObjectOutputStream

`ObjectOutputStream` and `FileOutputStream` are in the `java.io` package.

```
public ObjectOutputStream(OutputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

This creates a blank file. If there already is a file named `File_Name`, then the old contents of the file are lost.

If you want to create a stream using an object of the class `File`, you use

```
new ObjectOutputStream(new FileOutputStream(File_Object))
```

The constructor for `FileOutputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileOutputStream` constructor succeeds, then the constructor for `ObjectOutputStream` may throw a different `IOException`.

(continued)

# Some Methods in ObjectOutputStream (Part 1 of 5)

### Display 10.14 Some Methods in the Class ObjectOutputStream

```
public void writeInt(int n) throws IOException
```

Writes the int value n to the output stream.

```
public void writeShort(short n) throws IOException
```

Writes the short value n to the output stream.

```
public void writeLong(long n) throws IOException
```

Writes the long value n to the output stream.

```
public void writeDouble(double x) throws IOException
```

Writes the double value x to the output stream.

(continued)

## Some Methods in ObjectOutputStream (Part 2 of 5)

### Display 10.14 Some Methods in the Class ObjectOutputStream

```
public void writeFloat(float x) throws IOException
```

Writes the float value x to the output stream.

```
public void writeChar(int n) throws IOException
```

Writes the char value n to the output stream. Note that it expects its argument to be an int value. However, if you simply use the char value, then Java will automatically type cast it to an int value. The following are equivalent:

```
outputStream.writeChar((int)'A');
```

and

```
outputStream.writeChar('A');
```

(continued)

## Some Methods in ObjectOutputStream (Part 3 of 5)

**Display 10.14 Some Methods in the Class ObjectOutputStream**

```
public void writeBoolean(boolean b) throws IOException
```

Writes the boolean value b to the output stream.

```
public void writeUTF(String aString) throws IOException
```

Writes the String value aString to the output stream. UTF refers to a particular method of encoding the string. To read the string back from the file, you should use the method readUTF of the class ObjectInputStream.

(continued)

# Some Methods in ObjectOutputStream (Part 4 of 5)

### Display 10.14 Some Methods in the Class ObjectOutputStream

```
public void writeObject(Object anObject) throws IOException
```

Writes its argument to the output stream. The object argument should be an object of a serializable class, a concept discussed later in this chapter. Throws various IOExceptions.

```
public void close() throws IOException
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush() throws IOException
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

## Eg. BinaryOutputDemo

# Some Methods in ObjectOutputStream (Part 5 of 5)

- The class **ObjectInputStream** is a stream class that can be used to read from a binary file
  - An object of this class has methods to read strings, values of primitive types, and objects from a binary file
- A program using **ObjectInputStream** needs to import several classes from package **java.io**:

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

## Reading Simple Data from a Binary File

- An **ObjectInputStream** object is created and connected to a binary file as follows:

```
ObjectInputStream inStreamName = new
 ObjectInputStream(new
 FileInputStream(fileName));
```

- The constructor for **FileInputStream** may throw a **FileNotFoundException**
- The constructor for **ObjectInputStream** may throw an **IOException**
- Each of these must be handled

## Opening a Binary File for Reading

- After opening the file, **ObjectInputStream** methods can be used to read to the file
  - Methods used to input primitive values include **readInt**, **readDouble**, **readChar**, and **readBoolean**
  - The method **readUTF** is used to input values of type **String**
- If the file contains multiple types, each item type must be read in exactly the same order it was written to the file
- The stream should be closed after reading

# Opening a Binary File for Reading

### Display 10.15 Some Methods in the Class ObjectInputStream

The classes `ObjectInputStream` and `FileInputStream` are in the `java.io` package.

```
public ObjectInputStream(InputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectInputStream(new FileInputStream(File_Name))
```

Alternatively, you can use an object of the class `File` in place of the `File_Name`, as follows:

```
new ObjectInputStream(new FileInputStream(File_Object))
```

The constructor for `FileInputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileInputStream` constructor succeeds, then the constructor for `ObjectInputStream` may throw a different `IOException`.

(continued)

## Some Methods in ObjectInputStream (Part 1 of 5)

### Display 10.15 Some Methods in the Class ObjectInputStream

```
public int readInt() throws IOException
```

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file and that value was not written using the method `writeInt` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public int readShort() throws IOException
```

Reads a `short` value from the input stream and returns that `short` value. If `readShort` tries to read a value from the file and that value was not written using the method `writeShort` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

(continued)

## Some Methods in ObjectInputStream (Part 2 of 5)

**Display 10.15 Some Methods in the Class ObjectInputStream**

```
public long readLong() throws IOException
```

Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file and that value was not written using the method `writeLong` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public double readDouble() throws IOException
```

Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file and that value was not written using the method `writeDouble` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public float readFloat() throws IOException
```

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file and that value was not written using the method `writeFloat` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

(continued)

# Some Methods in ObjectInputStream (Part 3 of 5)

### Display 10.15 Some Methods in the Class ObjectInputStream

```
public char readChar() throws IOException
```

Reads a char value from the input stream and returns that char value. If readChar tries to read a value from the file and that value was not written using the method writeChar of the class ObjectOutputStream (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public boolean readBoolean() throws IOException
```

Reads a boolean value from the input stream and returns that boolean value. If readBoolean tries to read a value from the file and that value was not written using the method writeBoolean of the class ObjectOutputStream (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

(continued)

## Some Methods in ObjectInputStream (Part 4 of 5)

### Display 10.15 Some Methods in the Class ObjectInputStream

```
public String readUTF() throws IOException
```

Reads a String value from the input stream and returns that String value. If readUTF tries to read a value from the file and that value was not written using the method writeUTF of the class ObjectOutputStream (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
Object readObject() throws ClassNotFoundException, IOException
```

Reads an object from the input stream. The object read should have been written using writeObject of the class ObjectOutputStream. Throws a ClassNotFoundException if a serialized object cannot be found. If an attempt is made to read beyond the end of the file, an EOFException is thrown. May throw various other IOExceptions.

```
public int skipBytes(int n) throws IOException
```

Skips n bytes.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

## Some Methods in ObjectInputStream (Part 5 of 5)

- All of the **ObjectInputStream** methods that read from a binary file throw an **EOFException** when trying to read beyond the end of a file
  - This can be used to end a loop that reads all the data in a file
- Note that different file-reading methods check for the end of a file in different ways
  - Testing for the end of a file in the wrong way can cause a program to go into an infinite loop or terminate abnormally
  - Eg. **EOFDemo**

## Checking for the End of a Binary File

- Objects can also be input and output from a binary file
  - Use the **writeObject** method of the class **ObjectOutputStream** to write an object to a binary file
  - Use the **readObject** method of the class **ObjectInputStream** to read an object from a binary file
  - In order to use the value returned by **readObject** as an object of a class, it must be type cast first:

```
SomeClass someObject =
(SomeClass) objectInputStream.readObject();
```

# Binary I/O of Objects

- It is best to store the data of only one class type in any one file
  - Storing objects of multiple class types or objects of one class type mixed with primitives can lead to loss of data
- In addition, the class of the object being read or written must implement the **Serializable** interface
  - The **Serializable** interface is easy to use and requires no knowledge of interfaces
  - A class that implements the **Serializable** interface is said to be a *serializable class*

# Binary I/O of Objects

- In order to make a class serializable, simply add **implements Serializable** to the heading of the class definition  
`public class SomeClass implements Serializable`
- When a serializable class has instance variables of a class type, then all those classes must be serializable also
  - A class is not serializable unless the classes for all instance variables are also serializable for all levels of instance variables within classes

# The Serializable Interface

- Since an array is an object, arrays can also be read and written to binary files using **readObject** and **writeObject**
  - If the base type is a class, then it must also be serializable, just like any other class type
  - Since **readObject** returns its value as type **Object** (like any other object), it must be type cast to the correct array type:

```
SomeClass[] someObject =
(SomeClass[]) objectInputStream.readObject();
```

# Array Objects in Binary Files

- The streams for sequential access to files are the ones most commonly used for file access in Java
- However, some applications require very rapid access to records in very large databases
  - These applications need to have random access to particular parts of a file

## Random Access to Binary Files

- The stream class **RandomAccessFile**, which is in the **java.io** package, provides both read and write random access to a file in Java
- A random access file consists of a sequence of numbered bytes
  - There is a kind of marker called the *file pointer* that is always positioned at one of the bytes
  - All reads and writes take place starting at the *file pointer location*
  - The file pointer can be moved to a new location with the method **seek**

## Reading and Writing to the Same File

- Although a random access file is byte oriented, there are methods that allow for reading or writing values of the primitive types as well as string values to/from a random access file
  - These include **readInt**, **readDouble**, and **readUTF** for input, and **writeInt**, **writeDouble**, and **writeUTF** for output
  - It does not have **writeObject** or **readObject** methods, however

## Reading and Writing to the Same File

- The constructor for **RandomAccessFile** takes either a string file name or an object of the class **File** as its first argument
- The second argument must be one of four strings:
  - "**rw**", meaning the code can both read and write to the file after it is open
  - "**r**", meaning the code can read from the file, but not write to it
  - "**rws**" or "**rwd**" (See Table of methods from **RandomAccessFile**)

# Opening a File

- If the file already exists, then when it is opened, the length is not reset to 0, and the file pointer will be positioned at the start of the file
  - This ensures that old data is not lost, and that the file pointer is set for the most likely position for reading (not writing)
- The length of the file can be changed with the **setLength** method
  - In particular, the **setLength** method can be used to empty the file

## Pitfall: A Random Access File Need Not Start Empty

## Display 10.21 Some Methods of the Class RandomAccessFile

The class RandomAccessFile is in the java.io package.

```
public RandomAccessFile(String fileName, String mode)
public RandomAccessFile(File fileObject, String mode)
```

Opens the file, does not delete data already in the file, but does position the file pointer at the first (zeroth) location.

The mode must be one of the following:

- "r" Open for reading only.
- "rw" Open for reading and writing.
- "rws" Same as "rw", and also requires that every update to the file's content or metadata be written synchronously to the underlying storage device.
- "rwd" Same as "rw", and also requires that every update to the file's content be written synchronously to the underlying storage device.
- "rws" and "rwd" are not covered in this book.

(continued)

# Some Methods of RandomAccessFile (Part 1 of 7)

## Display 10.21 Some Methods of the Class RandomAccessFile

```
public long getFilePointer() throws IOException
```

Returns the current location of the file pointer. Locations are numbered starting with 0.

```
public void seek(long location) throws IOException
```

Moves the file pointer to the specified location.

```
public long length() throws IOException
```

Returns the length of the file.

```
public void setLength(long newLength) throws IOException
```

Sets the length of this file.

If the present length of the file as returned by the `length` method is greater than the `newLength` argument, then the file will be truncated. In this case, if the file pointer location as returned by the `getFilePointer` method is greater than `newLength`, then after this method returns, the file pointer location will be equal to `newLength`.

If the present length of the file as returned by the `length` method is smaller than `newLength`, then the file will be extended. In this case, the contents of the extended portion of the file are not defined.

(continued)

# Some Methods of RandomAccessFile (Part 2 of 7)

### Display 10.21 Some Methods of the Class RandomAccessFile

```
public void close() throws IOException
```

Closes the stream's connection to a file.

```
public void write(int b) throws IOException
```

Writes the specified byte to the file.

```
public void write(byte[] a) throws IOException
```

Writes `a.length` bytes from the specified byte array to the file.

```
public final void writeByte(byte b) throws IOException
```

Writes the byte `b` to the file.

```
public final void writeShort(short n) throws IOException
```

Writes the short `n` to the file.

(continued)

## Some Methods of RandomAccessFile (Part 3 of 7)

### Display 10.21 Some Methods of the Class RandomAccessFile

```
public final void writeInt(int n) throws IOException
```

Writes the int n to the file.

```
public final void writeLong(long n) throws IOException
```

Writes the long n to the file.

```
public final void writeDouble(double d) throws IOException
```

Writes the double d to the file.

```
public final void writeFloat(float f) throws IOException
```

Writes the float f to the file.

(continued)

## Some Methods of RandomAccessFile (Part 4 of 7)

## Display 10.21 Some Methods of the Class RandomAccessFile

```
public final void writeChar(char c) throws IOException
```

Writes the char c to the file.

```
public final void writeBoolean(boolean b) throws IOException
```

Writes the boolean b to the file.

```
public final void writeUTF(String s) throws IOException
```

Writes the String s to the file.

```
public int read() throws IOException
```

Reads a byte of data from the file and returns it as an integer in the range 0 to 255.

```
public int read(byte[] a) throws IOException
```

Reads a.length bytes of data from the file into the array of bytes a. Returns the number of bytes read or -1 if the end of the file is encountered.

(continued)

# Some Methods of RandomAccessFile (Part 5 of 7)

### Display 10.21 Some Methods of the Class RandomAccessFile

```
public final byte readByte() throws IOException
```

Reads a byte value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final short readShort() throws IOException
```

Reads a short value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final int readInt() throws IOException
```

Reads an int value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final long readLong() throws IOException
```

Reads a long value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

(continued)

## Some Methods of RandomAccessFile (Part 6 of 7)

### Display 10.21 Some Methods of the Class RandomAccessFile

```
public final double readDouble() throws IOException
```

Reads a double value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final float readFloat() throws IOException
```

Reads a float value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final char readChar() throws IOException
```

Reads a char value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

```
public final boolean readBoolean() throws IOException
```

Reads a boolean value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

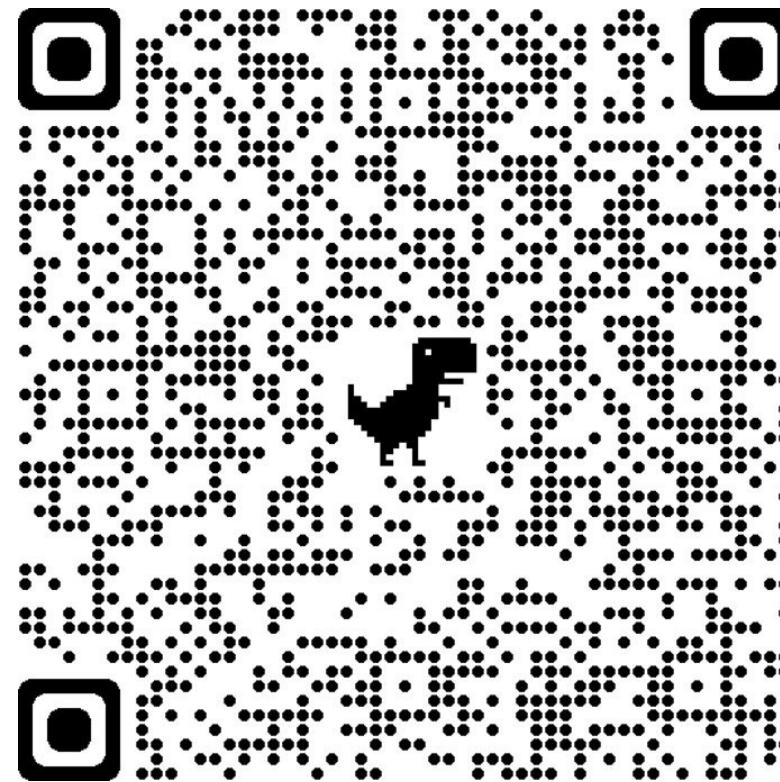
```
public final String readUTF() throws IOException
```

Reads a String value from the file and returns that value. If an attempt is made to read beyond the end of the file, an EOFException is thrown.

## Some Methods of RandomAccessFile (Part 5 of 7)

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

## Class Reflections



<http://go.unimelb.edu.au/5o8i>

## Class Reflections



# Programming and Software Development

## COMP90041

### Lecture 11

# Generics

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- \* the Textbook resources

- File I/O
- Buffered Reader
- Binary Files

# Review: Week 10

- Introduction to Generics & Generic Classes
- Bounds for Type Parameters
- Generic Methods
- Inheritance with Generic Classes

# Outline

- **Introduction to Generics & Generic Classes**
- Bounds for Type Parameters
- Generic Methods
- Inheritance with Generic Classes

# Outline

- Beginning with version 5.0, 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

# Introduction to Generics

```
1 public class SampleInteger
2 {
3 private Integer data;
4
5 public void setData(Integer newData)
6 {
7 data = newData;
8 }
9
10 public Integer getData()
11 {
12 return data;
13 }
14 }
```

```
1 public class SampleDouble
2 {
3 private Double data;
4
5 public void setData(Double newData)
6 {
7 data = newData;
8 }
9
10 public Double getData()
11 {
12 return data;
13 }
14 }
```

# Example

## Display 14.4 A Class Definition with a Type Parameter

```
1 public class Sample<T>
2 {
3 private T data;
4
5 public void setData(T newData)
6 {
7 data = newData; T is a parameter for a type.
8 }
9
10 public T getData()
11 {
12 return data;
13 }
14}
```

## Example

- Classes and methods can have a type parameter (a.k.a type variable)
  - 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
  - Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used
    - A naming convention:
      - E: Element (e.g., ArrayList<E>)
      - K: Key (e.g., HashMap<K, V>)
      - N: Number
      - T: Type
      - V: Value
      - S, U, V, and so on: Second, third, and fourth types

# Generics

- A class that is defined with a parameter for a type is called a ***generic class*** or a ***parameterized class***
  - The type parameter is included in angle brackets after the class name in the class definition heading
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter
  - The type parameter can be used like other types used in the definition of a class (e.g., instance variable declarations, method parameters)

# Class Definitions with a Type Parameter

- A class definition with a type parameter is stored in a file and compiled just like any other class
- Once a parameterized class is compiled, it can be used like any other class
  - However, the class type plugged in for the type parameter **must be specified before it can be used in a program**
  - Doing this is said to *instantiate* the generic class

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

## Generics: Instantiation

- There are many pitfalls that can be encountered when using type parameters
- Compiling with the **-Xlint** option will provide more informative diagnostics of any problems or potential problems in the code, including warnings

**javac -Xlint Sample.java**

**Tip: Compile with the -Xlint Option**

### Display 14.5 A Generic Ordered Pair Class

```
1 public class Pair<T>
2 {
3 private T first;
4 private T second;
5
6 public Pair()
7 {
8 first = null;
9 second = null;
10
11 public Pair(T firstItem, T secondItem)
12 {
13 first = firstItem;
14 second = secondItem;
15 }
16 }
```

Constructor headings do not include the type parameter in angular brackets.

(continued)

# A Generic Ordered Pair Class (Part 1 of 4)

### 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)

# A Generic Ordered Pair Class (Part 2 of 4)

**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)

# A Generic Ordered Pair Class (Part 3 of 4)

### 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 Pair<T> otherPair = (Pair<T>)otherObject;
46 return (first.equals(otherPair.first)
47 && second.equals(otherPair.second));
48 }
49 }
50 }
```

# A Generic Ordered Pair Class (Part 4 of 4)

## 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)

# Using Our Ordered Pair Class (Part 1 of 3)

### Display 14.6 Using Our Ordered Pair Class

```
15 if (inputPair.equals(secretPair))
16 {
17 System.out.println("You guessed the secret words");
18 System.out.println("in the correct order!");
19 }
20 else
21 {
22 System.out.println("You guessed incorrectly.");
23 System.out.println("You guessed");
24 System.out.println(inputPair);
25 System.out.println("The secret words are");
26 System.out.println(secretPair);
27 }
28 }
29 }
```

(continued)

# Using Our Ordered Pair Class (Part 2 of 3)

## Display 14.6 Using Our Ordered Pair Class

### SAMPLE DIALOGUE

Enter two words:

**two words**

You guessed incorrectly.

You guessed

first: two

second: words

The secret words are

first: Happy

second: Day

# Using Our Ordered Pair Class (Part 3 of 3)

- The type plugged in for a type parameter must always be a reference type
  - It cannot be a primitive type such as **int**, **double**, or **char**
  - However, now that Java has automatic boxing, this is not a big restriction
  - Note: reference types can include arrays

## Pitfall: A Primitive Type Cannot be Plugged in for a Type Parameter

### 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)

# Using Ordered Pair Class and Automatic Boxing (Part 1 of 2)

**Display 14.7 Using Our Ordered Pair Class and Automatic Boxing**

```
15 if (inputPair.equals(secretPair))
16 {
17 System.out.println("You guessed the secret numbers");
18 System.out.println("in the correct order!");
19 }
20 else
21 {
22 System.out.println("You guessed incorrectly.");
23 System.out.println("You guessed");
24 System.out.println(inputPair);
25 System.out.println("The secret numbers are");
26 System.out.println(secretPair);
27 }
28 }
```

**Display 14.7 Using Our Ordered Pair Class and Automatic Boxing****SAMPLE DIALOGUE**

Enter two numbers:

42 24

You guessed the secret numbers  
in the correct order!

# Using Ordered Pair Class and Automatic Boxing (Part 2 of 2)

- Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed
- In particular, the type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

**T object = new T();  
T[] a = new T[10];**

## Pitfall: A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

- Arrays such as the following are illegal:

```
Pair<String>[] a =
new Pair<String>[10];
```

- Although this is a reasonable thing we want to do, it is not allowed given the way that Java implements generic classes

## Pitfall: An Instantiation of a Generic Class Cannot be an Array Base Type

- It is not permitted to create a generic class with **Exception**, **Error**, **Throwable**, or any descendent class of **Throwable**
  - A generic class cannot be created whose objects are throwable  
**public class GEx<T> extends Exception**
  - The above example will generate a compiler error message

## Pitfall: A Generic Class Cannot Be an Exception Class

- A generic class definition can have any number of type parameters
  - Multiple type parameters are listed in angle brackets just as in the single type parameter case, but are separated by commas

**Tip: A Class Definition Can Have More Than One Type Parameter/Type Variable**

## Display 14.8 Multiple Type Parameters

---

```
1 public class TwoTypePair<T1, T2>
2 {
3 private T1 first;
4 private T2 second;
5
5 public TwoTypePair()
6 {
7 first = null;
8 second = null;
9 }
10
10 public TwoTypePair(T1 firstItem, T2 secondItem)
11 {
12 first = firstItem;
13 second = secondItem;
14 }
```

(continued)

# Multiple Type Parameters (Part 1 of 4)

### 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)

## Multiple Type Parameters (Part 2 of 4)

### 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)

## Multiple Type Parameters (Part 3 of 4)

### Display 14.8 Multiple Type Parameters

```
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 TwoTypePair<T1, T2> otherPair =
46 (TwoTypePair<T1, T2>)otherObject;
47 return (first.equals(otherPair.first)
48 && second.equals(otherPair.second));
49 }
50 }
51 }
```

The first equals is the equals of the type T1. The second equals is the equals of the type T2.



## Multiple Type Parameters (Part 4 of 4)

### 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)

## Using a Generic Class with Two Type Parameters (Part 1 of 2)

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

```
15 System.out.println(
16 "Our new rating for " + rating.getFirst());
17 System.out.println(" is " + rating.getSecond());
18 }
19 }
```

#### SAMPLE DIALOGUE

Our current rating for The Car Guys  
is 8

How would you rate them?

10

Our new rating for The Car Guys  
is 10

## Using a Generic Class with Two Type Parameters (Part 2 of 2)

- Introduction to Generics & Generic Classes
- **Bounds for Type Parameters**
- Generic Methods
- Inheritance with Generic Classes

# Outline

- Sometimes it makes sense to **restrict the possible types** that can be plugged in for a type parameter **T**
  - For instance, to ensure that only classes that implement the **Comparable** interface are plugged in for **T**, define a class as follows:

```
public class RClass<T extends Comparable>
```

- "**extends Comparable**" serves as a *bound* on the type parameter **T**
- Any attempt to plug in a type for **T** which does not implement the **Comparable** interface will result in a compiler error message

## Bounds for Type Parameters

- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain ***multiple interfaces and up to one class***
- If there is more than one type parameter, the syntax is as follows:

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

## Bounds for Type Parameters

### Display 14.10 A Bounded Type Parameter

```
1 public class Pair<T extends Comparable>
2 {
3 private T first;
4 private T second;
5
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 14.5  
are also included as part of this generic class definition>

```
12 }
```

# A Bounded Type Parameter

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters

## Tip: Generic Interfaces

- Introduction to Generics & Generic Classes
- Bounds for Type Parameters
- **Generic Methods**
- Inheritance with Generic Classes

# Outline

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
  - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
  - The type parameter of a generic method is ***local*** to that method, not to the class

## Generic Methods (advanced)

- The type parameter must be placed (in angle brackets) after all the modifiers, and before the returned type

**public static <T> T genMethod(T[] a)**

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angle brackets

**String s = NonG.<String>genMethod(c);**

# Generic Methods

```
1 public class Utility
2 {
3 public static <T> T getMidPoint(T[] a)
4 {
5 return a[a.length/2];
6 }
7
8 public static <T> T getFirst(T[] a)
9 {
10 return a[0];
11 }
12
13 public static <T1, T2> boolean isSameClass(T1 a, T2 b)
14 {
15 return (a.getClass() == b.getClass());
16 }
17 }
```

## Example

- Introduction to Generics & Generic Classes
- Bounds for Type Parameters
- Generic Methods
- Inheritance with Generic Classes

# Outline

- A generic class can be defined as a derived class of an ordinary class or of another generic class
  - As in ordinary classes, an object of the subclass type would also be of the superclass type

## Inheritance with Generic Classes

## Display 14.11 A Derived Generic Class

```
1 public class UnorderedPair<T> extends Pair<T>
2 {
3 public UnorderedPair()
4 {
5 setFirst(null);
6 setSecond(null);
7 }
8
9 public UnorderedPair(T firstItem, T secondItem)
10 {
11 setFirst(firstItem);
12 setSecond(secondItem);
13 }
14 }
```

(continued)

# A Derived Generic Class (Part 1 of 2)

### Display 14.11 A Derived Generic Class

```
13 public boolean equals(Object otherObject)
14 {
15 if (otherObject == null)
16 return false;
17 else if (getClass() != otherObject.getClass())
18 return false;
19 else
20 {
21 UnorderedPair<T> otherPair =
22 (UnorderedPair<T>)otherObject;
23 return (getFirst().equals(otherPair.getFirst()))
24 && getSecond().equals(otherPair.getSecond()))
25 ||
26 (getFirst().equals(otherPair.getSecond()))
27 && getSecond().equals(otherPair.getFirst()));
28 }
29 }
30 }
```

## A Derived Generic Class (Part 2 of 2)

### Display 14.12 Using UnorderedPair

```
1 public class UnorderedPairDemo
2 {
3 public static void main(String[] args)
4 {
5 UnorderedPair<String> p1 =
6 new UnorderedPair<String>("peanuts", "beer");
7 UnorderedPair<String> p2 =
8 new UnorderedPair<String>("beer", "peanuts");
```

(continued)

# Using UnorderedPair (Part 1 of 2)

### Display 14.12 Using UnorderedPair

```
9 if (p1.equals(p2))
10 {
11 System.out.println(p1.getFirst() + " and " +
12 p1.getSecond() + " is the same as");
13 System.out.println(p2.getFirst() + " and " +
14 + p2.getSecond());
15 }
16 }
17 }
```

#### SAMPLE DIALOGUE<sup>2</sup>

peanuts and beer is the same as  
beer and peanuts

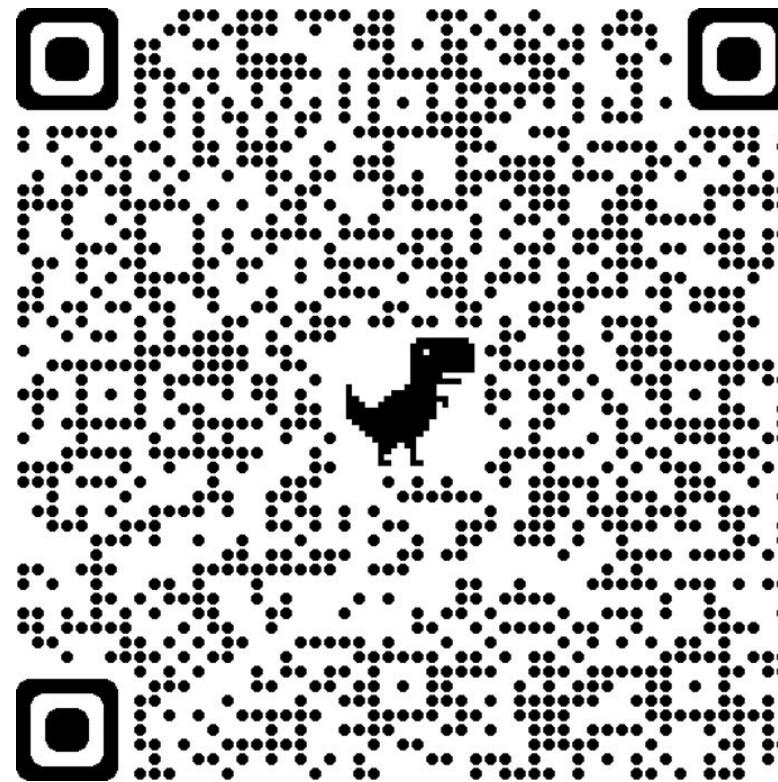
# Using UnorderedPair (Part 2 of 2)

- Given two classes: **A** and **B**, and given **G**: a generic class, there is no relationship between **G<A>** and **G<B>**
  - This is true regardless of the relationship between class **A** and **B**, e.g., if class **B** is a subclass of class **A**
- Example:
  - Suppose HourlyEmployee is a derived class of the class Employee, there is no relationship between **G<HourlyEmployee>** and **G<Employee>**

## Pitfall: Inheritance with Generic Classes

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

## Class Reflections



[http://go.unimelb.edu.au/5o8i.](http://go.unimelb.edu.au/5o8i)

## Class Reflections



Programming and Software Development  
COMP90041  
Lecture 12

# Automated Testing & Semester Review

NOTE: Some of the Material in these slides are adopted from  
\* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and  
\* the Textbook resources



# Programming and Software Development

## COMP90041

### Lecture 12

# Automated Testing

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- \* the Textbook resources



**~25,000**

**In Google products (e.g., Chrome)**



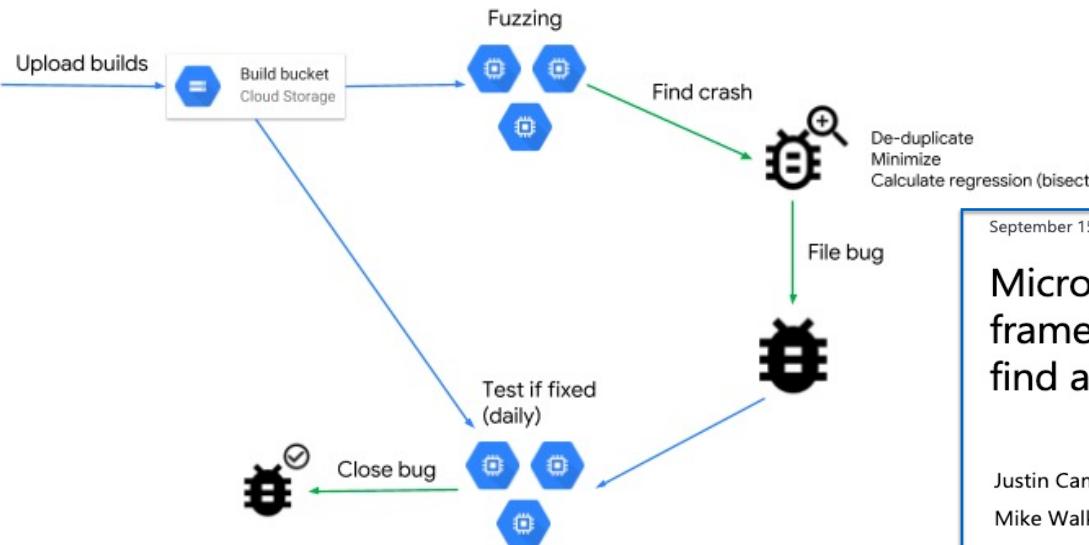
**~22,500**

**In 340+ open-source projects  
integrated with OSS-Fuzz**

## Bugs found by Fuzzing at Google

# Open sourcing ClusterFuzz

Thursday, February 7, 2019



September 15, 2020

Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale

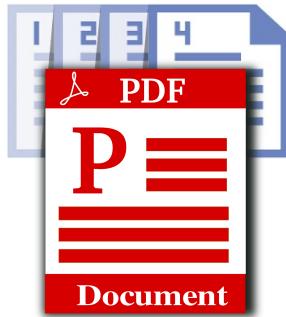
Justin Campbell Principal Security Software Engineering Lead, Microsoft Security

Mike Walker Senior Director, Special Projects Management, Microsoft Security

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a single command, which can be baked into CI/CD, developers can launch fuzz jobs from a few virtual machines to thousands of cores.

# Fuzzing in CI/CD

**Input corpus  
(i.e., seeds)**



**Black-box  
Fuzzer**

**mutated inputs**



**System  
Under Test**



**Monitor  
(e.g. Crash  
detection)**

**crash  
report**



**crash  
input**

# How does Fuzzing work?

```
void Fn(char buf[4])
{
 if (buf[0] == 'b') {
 if (buf[1] == 'a') {
 if (buf[2] == 'd') {
 if (buf[3] == '!') {
 CRASH();
 }
 }
 }
 }
}
```

“good”

Black-box  
Fuzzing

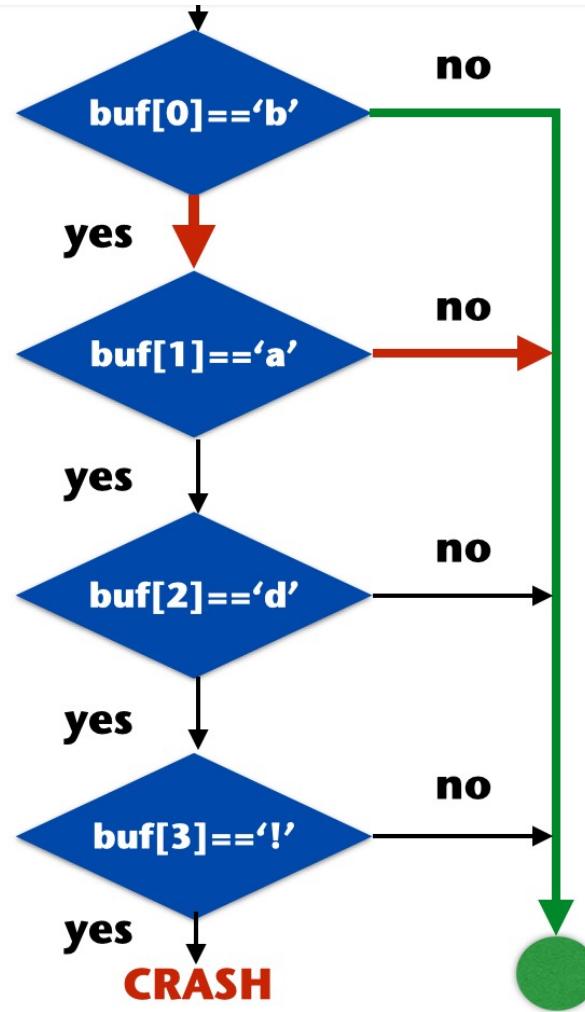
o!do  
god!o  
good  
?oo?  
cood!  
b?od  
ooooood  
godnHgggggggggggggg  
gggggggggggggggggggggg  
goad!  
go-590ooooood  
\$ggofd

# Example

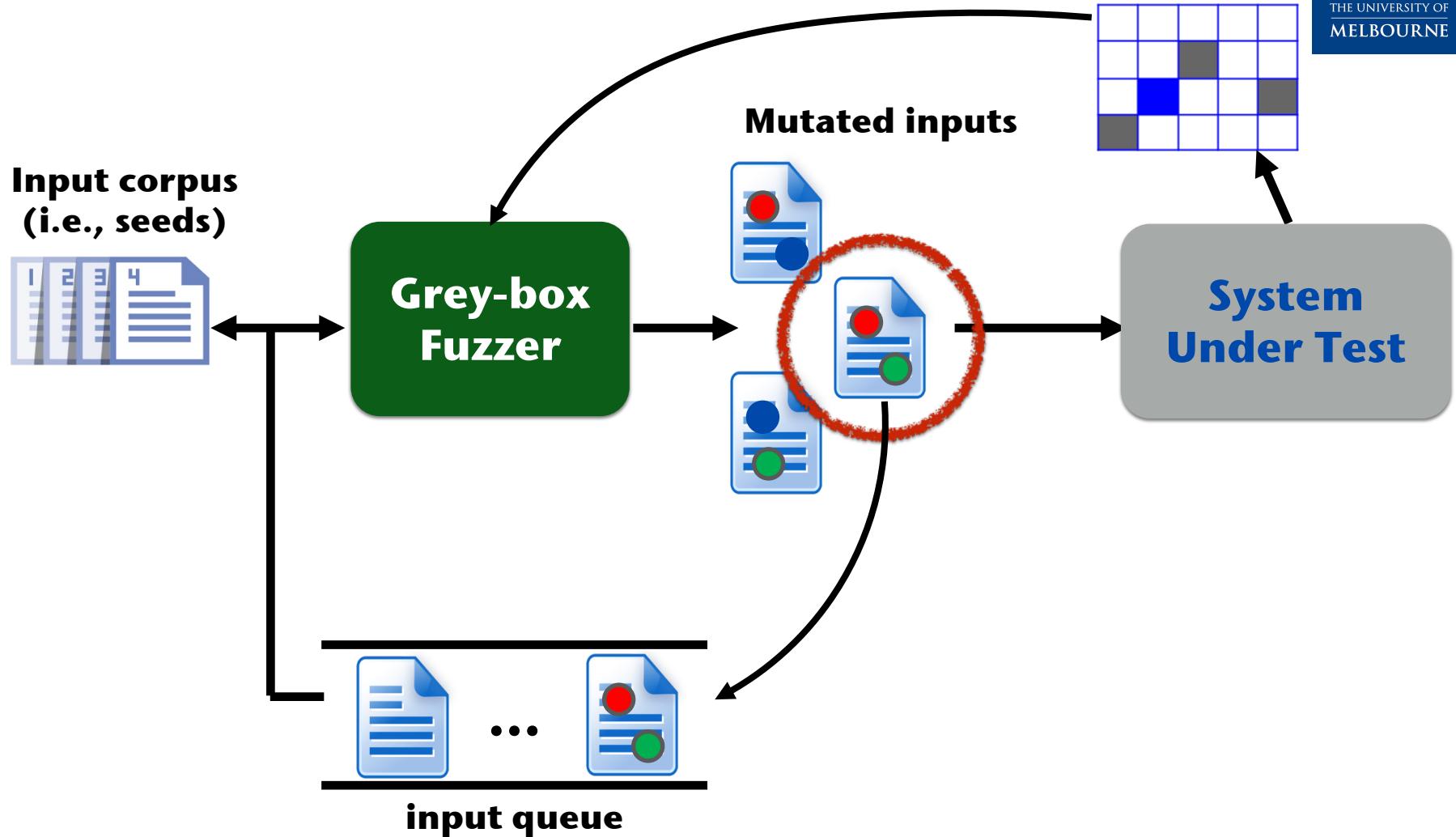
o!do  
god!o  
good  
?oo?  
cood!

**b?od**

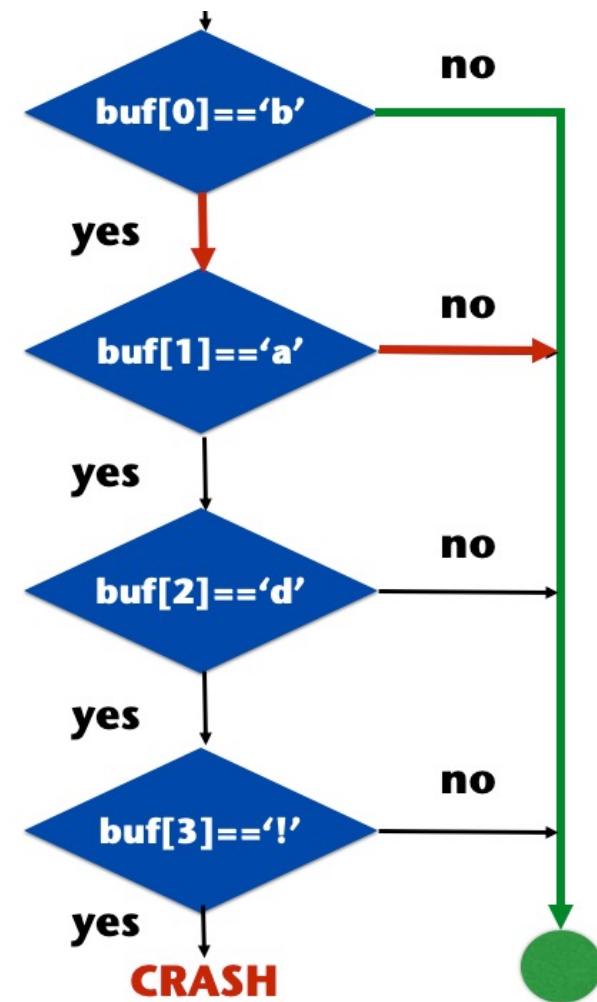
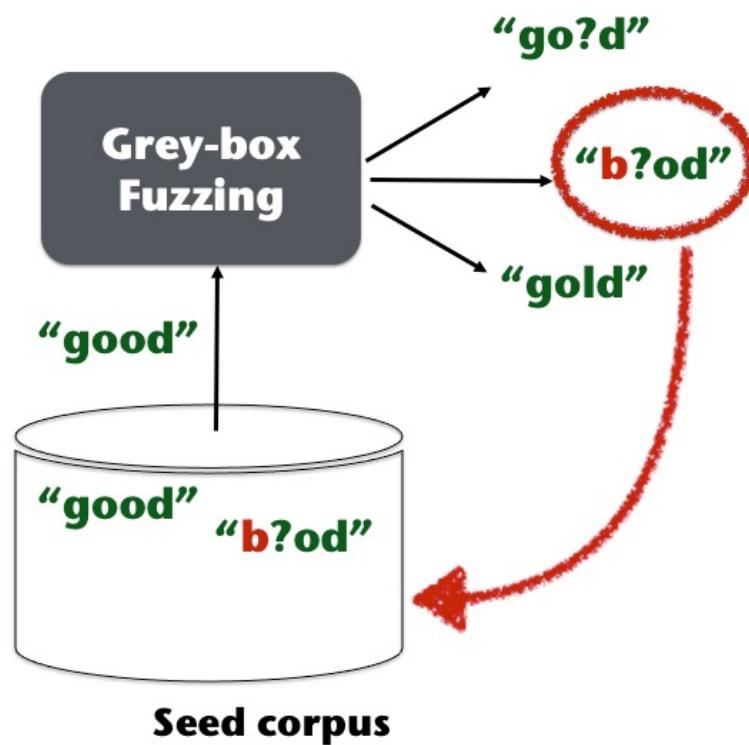
oooooooood  
godnHgggggggggggggg  
ggggggggggggggggggggggood  
goad!  
go-590ooooood  
\$ggofd



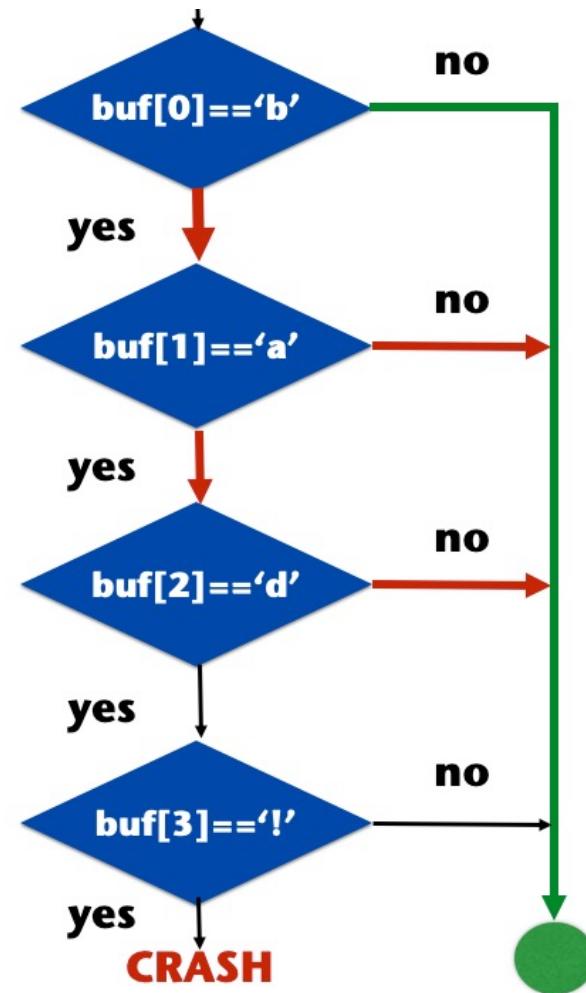
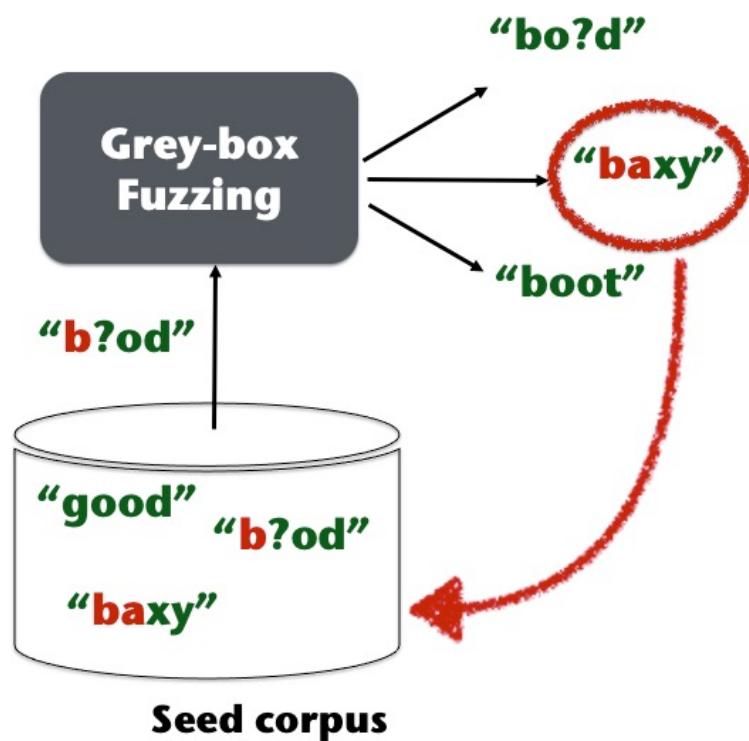
## A deeper look



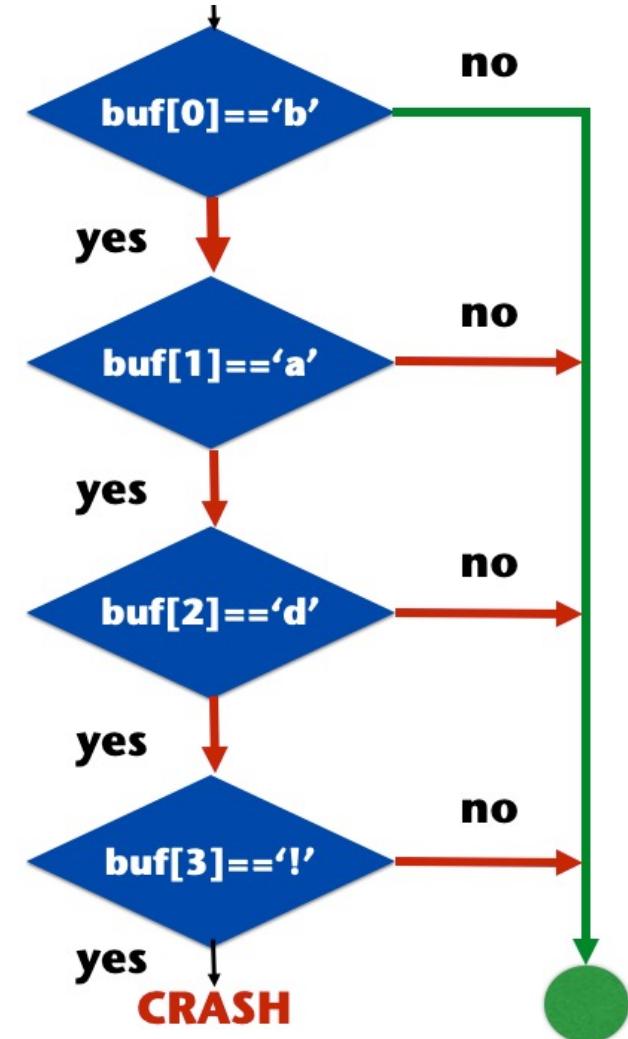
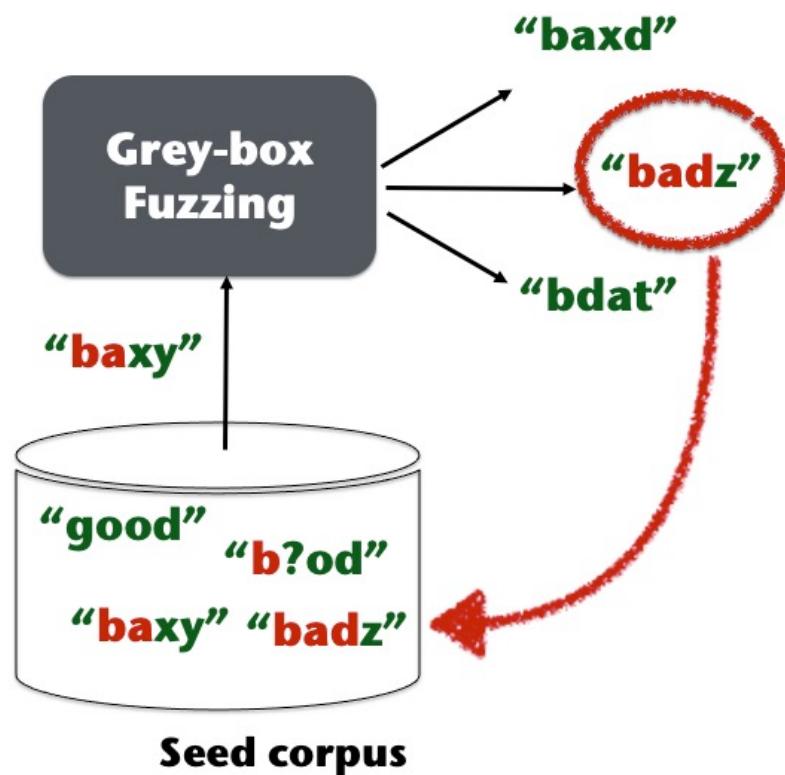
# Coverage-Guided Greybox Fuzzing



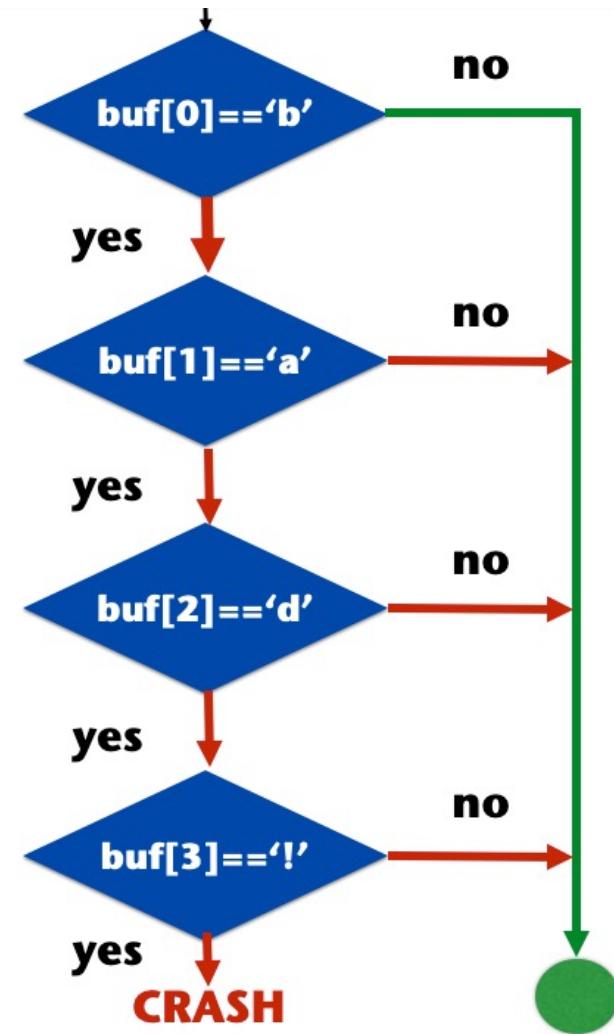
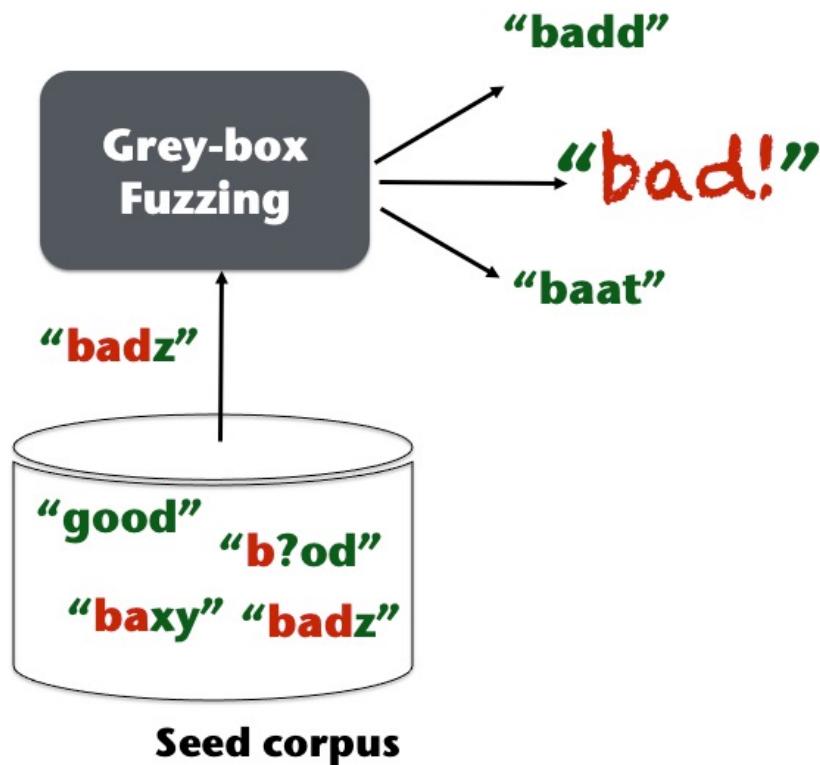
# Example



## Example (cont'd)



## Example (cont'd)



## Example (cont'd)

- Black-box fuzzing: JUnit-QuickCheck
  - <https://github.com/pholser/junit-quickcheck>
- Grey-box fuzzing: JQF
  - JUnit-QuickCheck + Code coverage feedback + Smart algorithms
  - <https://github.com/rohanpadhye/JQF>

# How do we fuzz Java programs?

- JUnit is a ***unit testing*** framework for Java
- JUnit promotes the idea of “first testing then coding”  
(a.k.a test-driven software development)
- It increases the productivity of programmers and the stability of the software systems
  - Find more bugs and find them ***earlier***
  - Support ***regression testing***

## Let's talk about JUnit first

Implement a function `classify` which takes 3 inputs that represent the lengths of the sides of a triangle and returns an integer where the return value

- 1 means it is equilateral (all sides equal length)
- 2 means it is isosceles (exactly 2 equal sides)
- 3 means it is scalene (no equal sides)
- 4 means it is an illegal triangle

```
public int classify(int a, int b, int c)
{
 if (a <= 0 && b <= 0 && c <= 0)
 return 4; //invalid
 if (a <= c - b || b <= a - c || c <= b - a)
 return 4; //invalid
 if (a == b && b == c)
 return 1; //equilateral
 if (a == b || b == c || c == a)
 return 2; //isosceles
 return 3; //scalene
}
```

## Example-How do we test this buggy program?

```
public static void main(String[] args)
{
 Triangle obj = new Triangle();
 Scanner keyboard = new Scanner(System.in);
 while(true)
 {
 int a, b, c;
 System.out.println("Enter the length of 3 edges:");
 System.out.print("a: ");
 a = keyboard.nextInt();
 System.out.print("b: ");
 b = keyboard.nextInt();
 System.out.print("c: ");
 c = keyboard.nextInt();

 System.out.printf("Given lengths: a=%d, b=%d, c=%d\n", a, b, c);
 int result = obj.classify(a, b, c);

 //print the result ...

 keyboard.nextLine();
 System.out.print("Test more cases? (Y/N): ");
 char selectedOption = Character.toLowerCase(keyboard.next().charAt(0));
 if (selectedOption == 'n') break;
 }
}
```

Triangle.java

# Option-1. Manual Testing

- Problems
  - Time consuming & tedious
  - The program should be tested after each (major) changes to
    - Check the correctness of new code
    - Prevent regression bugs
  - Likely to miss ***corner/edge cases***

## Option-1. Manual Testing

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TriangleJUnitTest {

 @Test
 public void testInvalidTriangle() {
 Triangle obj = new Triangle();
 assertEquals(4, obj.classify(-1, 1, 1));
 assertEquals(4, obj.classify(1, 2, 3));
 assertEquals(4, obj.classify(-5, -5, -5));
 }
}
```

[TriangleJUnitTest.java](#)

- Bundle test cases in a test class
- The test class can be executed automatically once the code has been updated
- Problem: developers/testers still need to design test inputs

## Option-2. Test Automation with JUnit

- Test inputs are generated **automatically** in a **black-box manner**

```
import java.util.*;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.runner.RunWith;
import com.pholser.junit.quickcheck.Property;
import com.pholser.junit.quickcheck.runner.JUnitQuickcheck;

@RunWith(JUnitQuickcheck.class)
public class TriangleQCheckTest {

 @Property
 public void testInvalidTriangle(int a, int b, int c) {
 assumeTrue(a <= 0 || b <= 0 || c <= 0);
 Triangle obj = new Triangle();
 System.out.printf("\nGenerated lengths: a=%d, b=%d, c=%d\n", a, b, c);
 assertTrue("Invalid triangle", obj.classify(a, b, c) == 4);
 }
}
```

## TriangleQCheckTest.java

# Option-3. Junit-QuickCheck

- Test inputs are generated ***automatically*** in a ***grey-box manner***

```
import java.util.*;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.runner.RunWith;
import com.pholser.junit.quickcheck.*;
import com.pholser.junit.quickcheck.generator.*;
import edu.berkeley.cs.jqf.fuzz.*;

@RunWith(JQF.class)
public class TriangleJQFTest {

 @Fuzz
 public void testInvalidTriangle(int a, int b, int c) {
 assumeTrue(a <= 0 || b <= 0 || c <= 0);
 Triangle obj = new Triangle();
 //System.out.printf("\nGenerated lengths: a=%d, b=%d, c=%d\n", a, b, c);
 assertTrue("Invalid triangle", obj.classify(a, b, c) == 4);
 }
}
```

## TriangleJQFTest.java

# Option-4. JQF

- Clone the `java_test_automation` repository
  - `git clone https://github.com/thuanpv/java test automation.git`
- Follow the instructions in `README.md`
  - Build a Docker image
  - Run a Docker container
  - Compile the source files
  - Run tests
- To learn more about Docker, read this:  
<https://docs.docker.com/get-started/overview/>

**Demo – Run all examples  
in a Docker container**

- Add more tests e.g., `testEquilateralTriangle`, `testIsosceleTriangle`, `testScaleneTriangle`
- Fuzz these tests using JUnit-QuickCheck and JQF

# Homework



# Programming and Software Development

## COMP90041

### Lecture 12

# Semester Review

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte, Dr. Rose Williams, and
- \* the Textbook resources



# Semester Review

- Object-Oriented (OO) software development
  - Program design, implementation and testing
  - OO concept
    - classes
    - objects
    - encapsulation
    - inheritance
    - polymorphism
  - The Java programming language
  - Problem solving
    - data structures
    - algorithms

# Expectation

- 1: Introduction
  - What a java program looks like? How it works? How to compile and run it, etc
  - Basic operations: primitive types, identifiers, assignment statement, arithmetics, string, etc
- 2: Console Input and Output
  - System.out.println (printf, print), various formats
  - Input using the scanner class, nextInt (nextFloat...), nextLine, etc
- 3: Flow of Control
  - Boolean expressions: logical values, !, &&, ||, >, <, ==, precedence and association rules
  - Branching: if-else; multiway if-else; switch; break; continue
  - Loops: for, while, do-while; nested loop, infinite loop, debugging a loop

# Review

- 4: Classes I
  - Type, members (instance variables, methods), local/global variables, this, access permission (public, private), **overloading** (same name different signature), constructors
  - Modularity, information hiding, encapsulation
- 5: Classes II
  - Static methods and variables
  - Modular design
  - References, privacy leak, mutable and immutable classes, equals, toString, packages
  - UML
  - Wrapper classes
- 6: Memory and Arrays
  - Variables in memory
  - Privacy leaks
  - Mutable/imutable types
  - Packages and Javadoc
  - Arrays:
    - Basic operations, references, string array, **No** multidimensional array.
    - Sorting
- 7: Inheritance
  - Base/derived classes, **overriding**, super constructor
  - More access permission
  - Enumerations

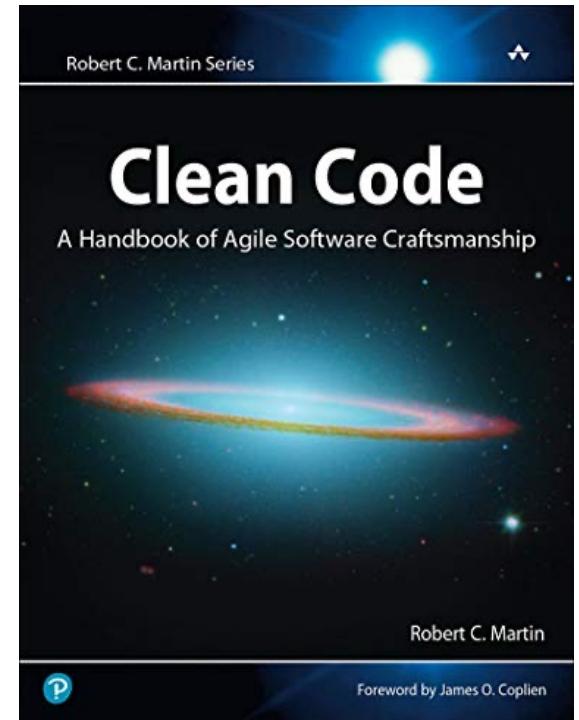
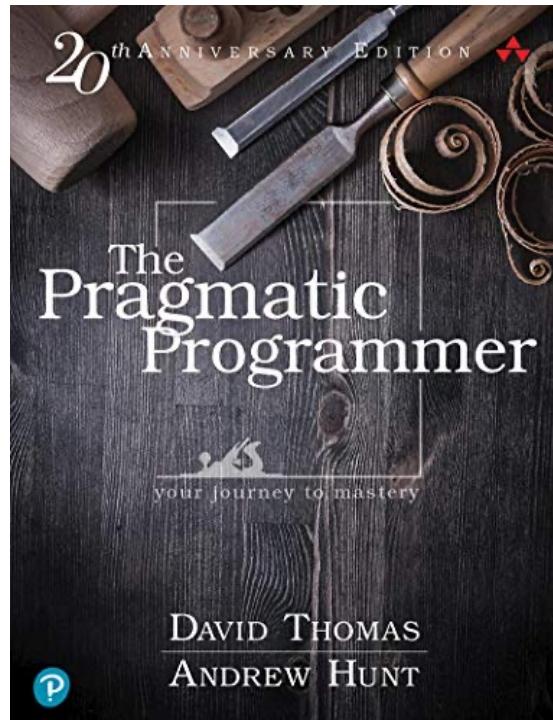
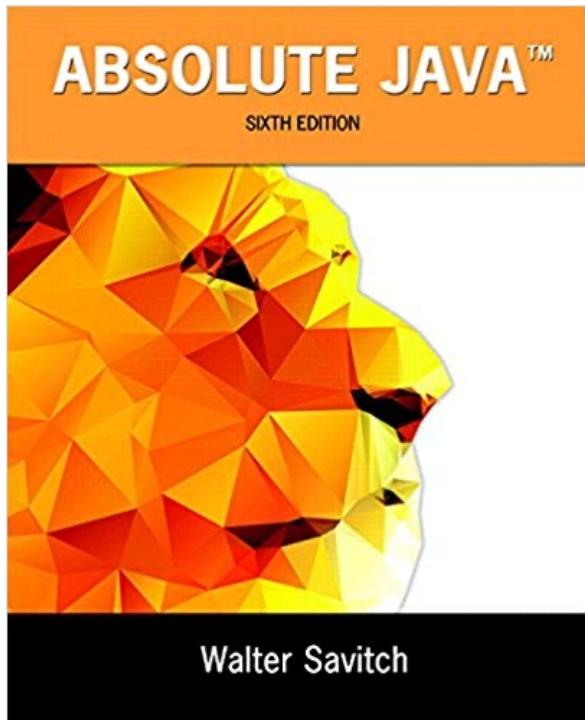
# Review

- 8: Polymorphism and Abstract classes
  - Late binding (`toString`): **except static, final, private methods**; downcasting/upcasting;  
**No clone methods**
  - Abstract class: a class containing an abstract method; cannot define an object of an abstract class
  - Interfaces
  - Handling exceptions
    - Try-throw-catch; Exception class; `getMessage()`; checked/unchecked exceptions
- 9: File I/O
  - Input/Output streams
  - Text file and binary file
  - Textfile: opening, writing (`PrintWriter`), reading (`Scanner`, `BufferedReader`)
  - Buffered Reader
  - Binary Files: `ObjectInputStream/ObjectOutputStream`, `Serializable` interface
  - File class (`getName`, `setReadonly`, `delete`, etc)
- 10: Generics & `ArrayList`
  - `ArrayList`: basic operations, methods (`add`, `set`, `get`, etc), for-each loop
  - Generics: parameterized classes/methods
  - **No coding required for generic methods**

# Review

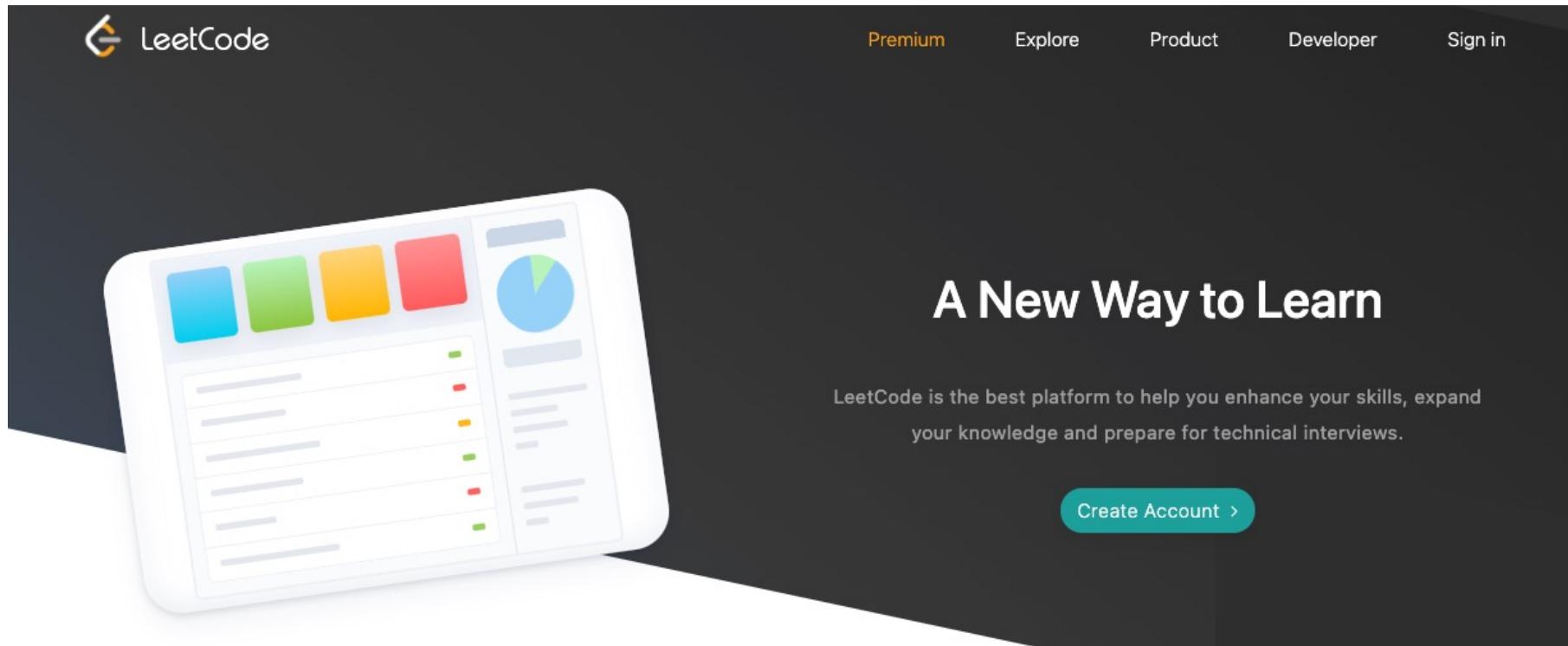
- More advanced subjects
  - Algorithms and complexity
  - Databases
  - Machine learning
  - Security & Software Testing
  - ...
- Research projects

# What's next?



# Readings

<https://leetcode.com/>



The screenshot shows the LeetCode homepage. At the top left is the LeetCode logo. To the right are navigation links: Premium, Explore, Product, Developer, and Sign in. A large graphic on the left depicts a smartphone displaying a dashboard with various data visualizations like a pie chart and bar graphs. To the right of the phone, the text "A New Way to Learn" is displayed in large white letters. Below this, a descriptive paragraph reads: "LeetCode is the best platform to help you enhance your skills, expand your knowledge and prepare for technical interviews." At the bottom right of the main content area is a teal button with the text "Create Account >".

**And writing more code!**



## Final Reflections



**Thank you all!**