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

Udaya Parampalli & Thuan Pham
Semester 1 2021, 12-04-2021

- 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

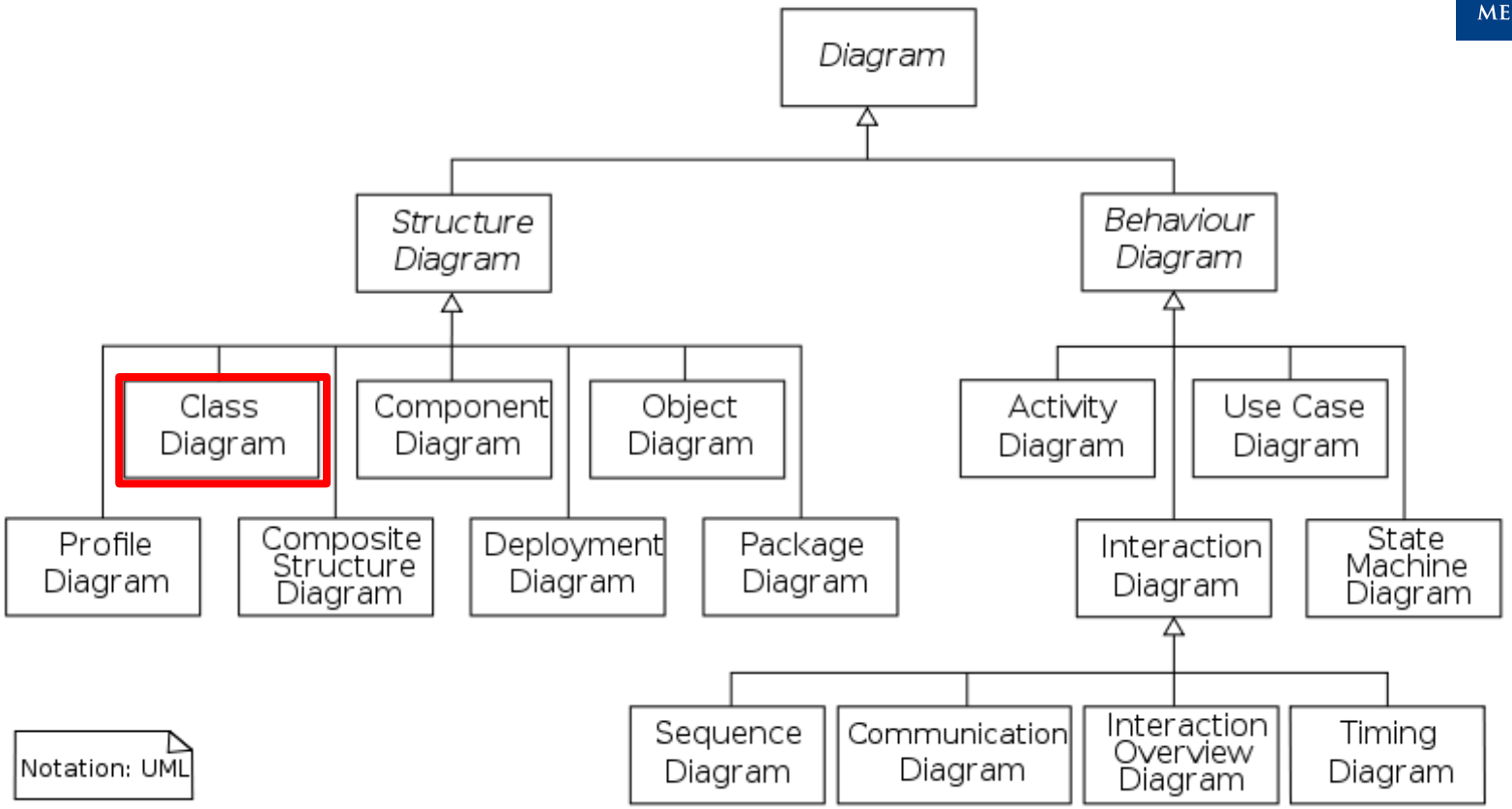


Image from Wikipedia

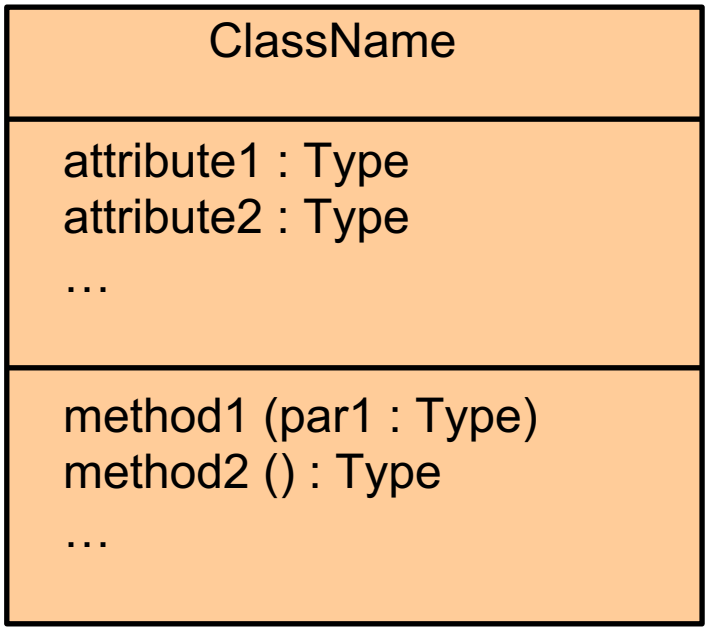
UML Diagrams

- A type of *structure diagram*

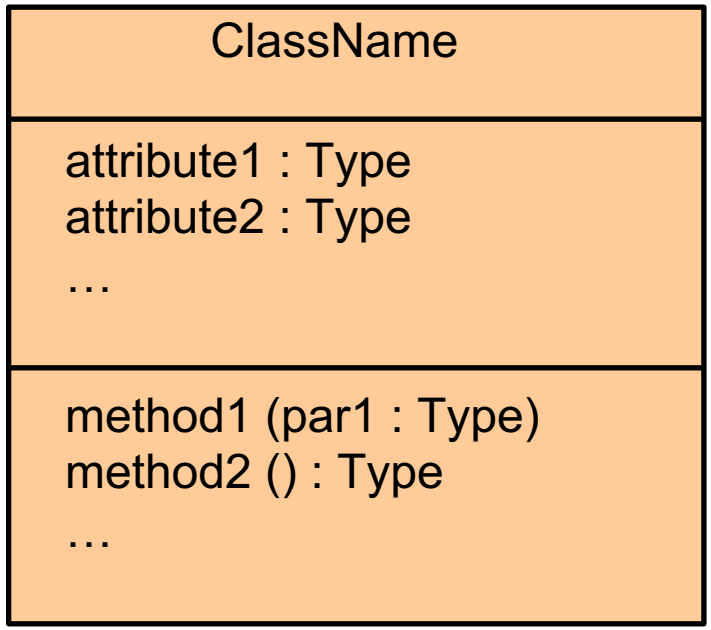
- Describes the structure of a system by showing the system's:
  - Classes
  - Their attributes (and the accessibility)
  - Methods
  - The relationships among the classes

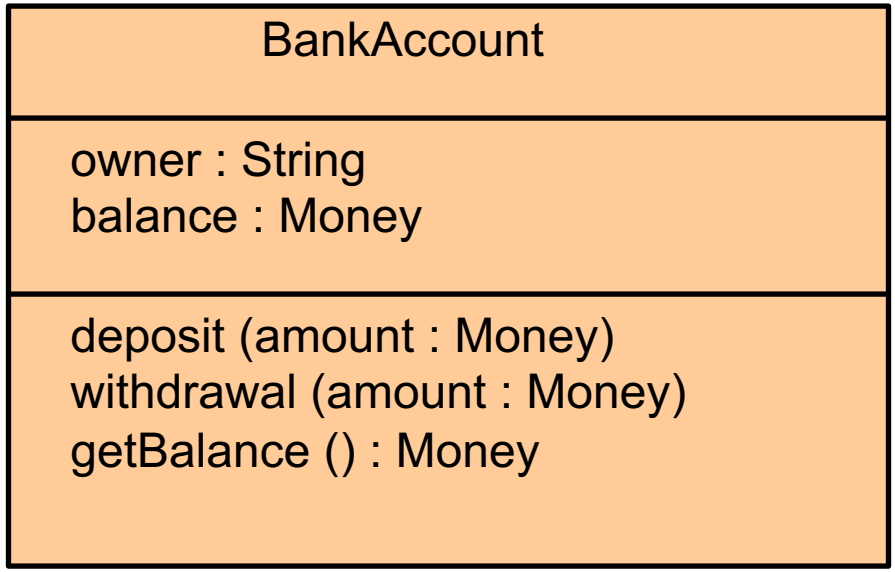
UML Class Diagrams

- 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



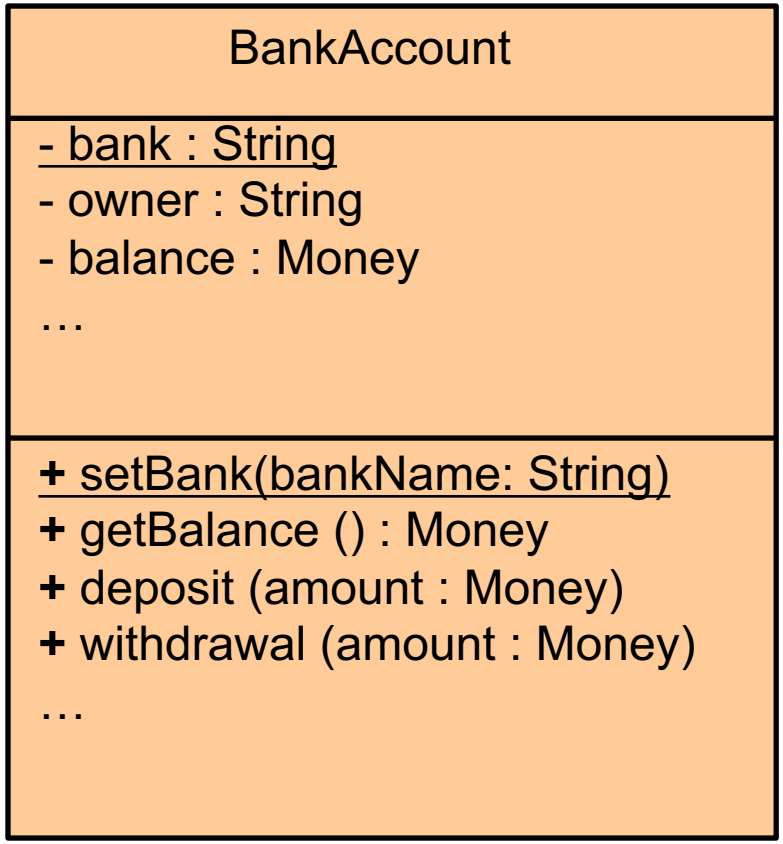


A UML Class

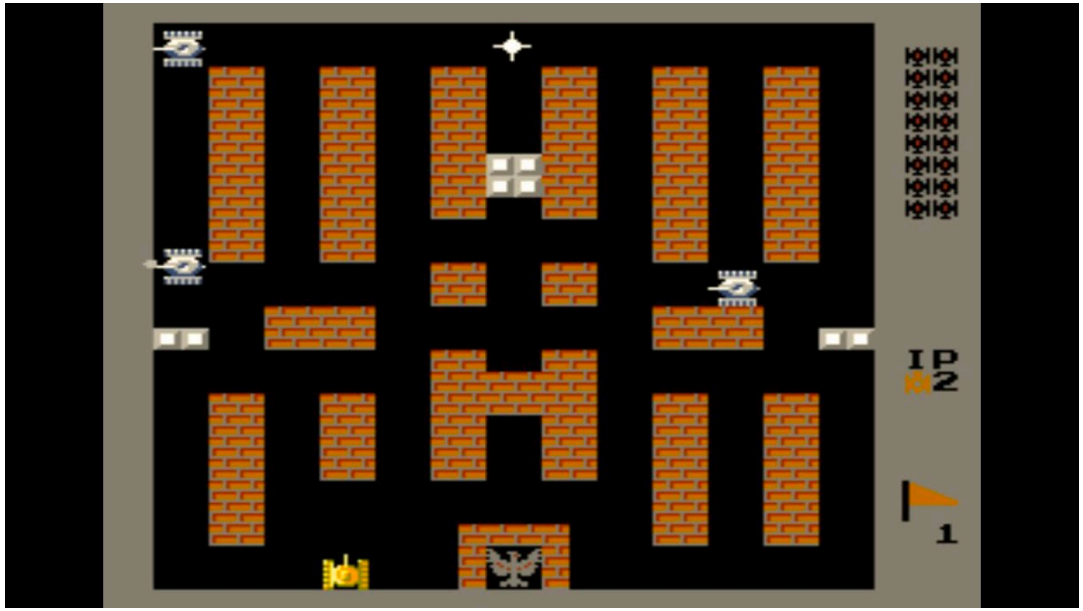
- The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type
- Each name is preceded by a character that specifies its ***access type***:
  - A minus sign (-) indicates private access
  - A plus sign (+) indicates public access
  - A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



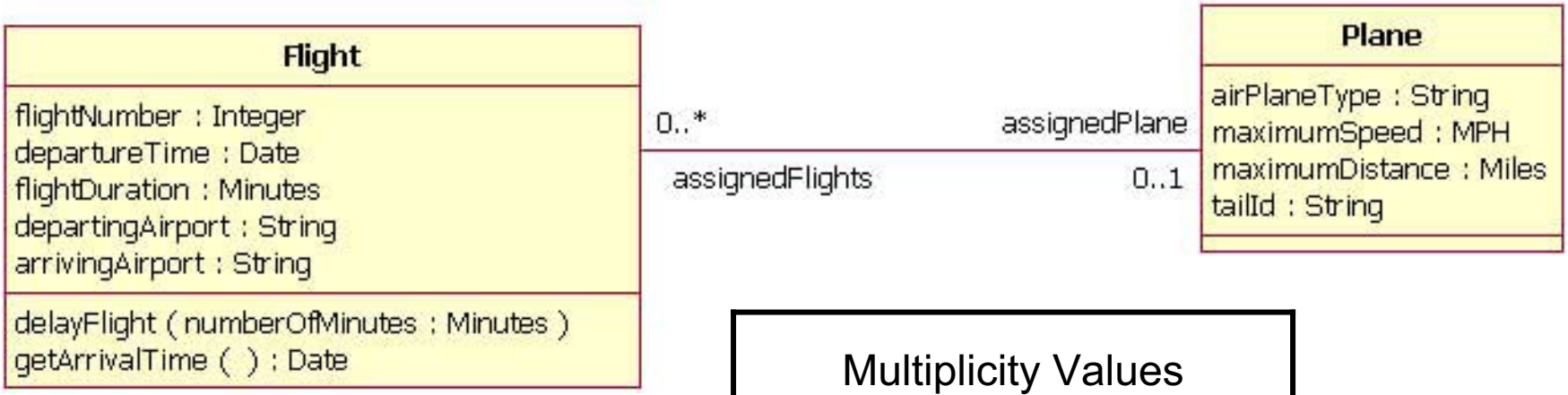
Battle City Tank - Gameplay:

- 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

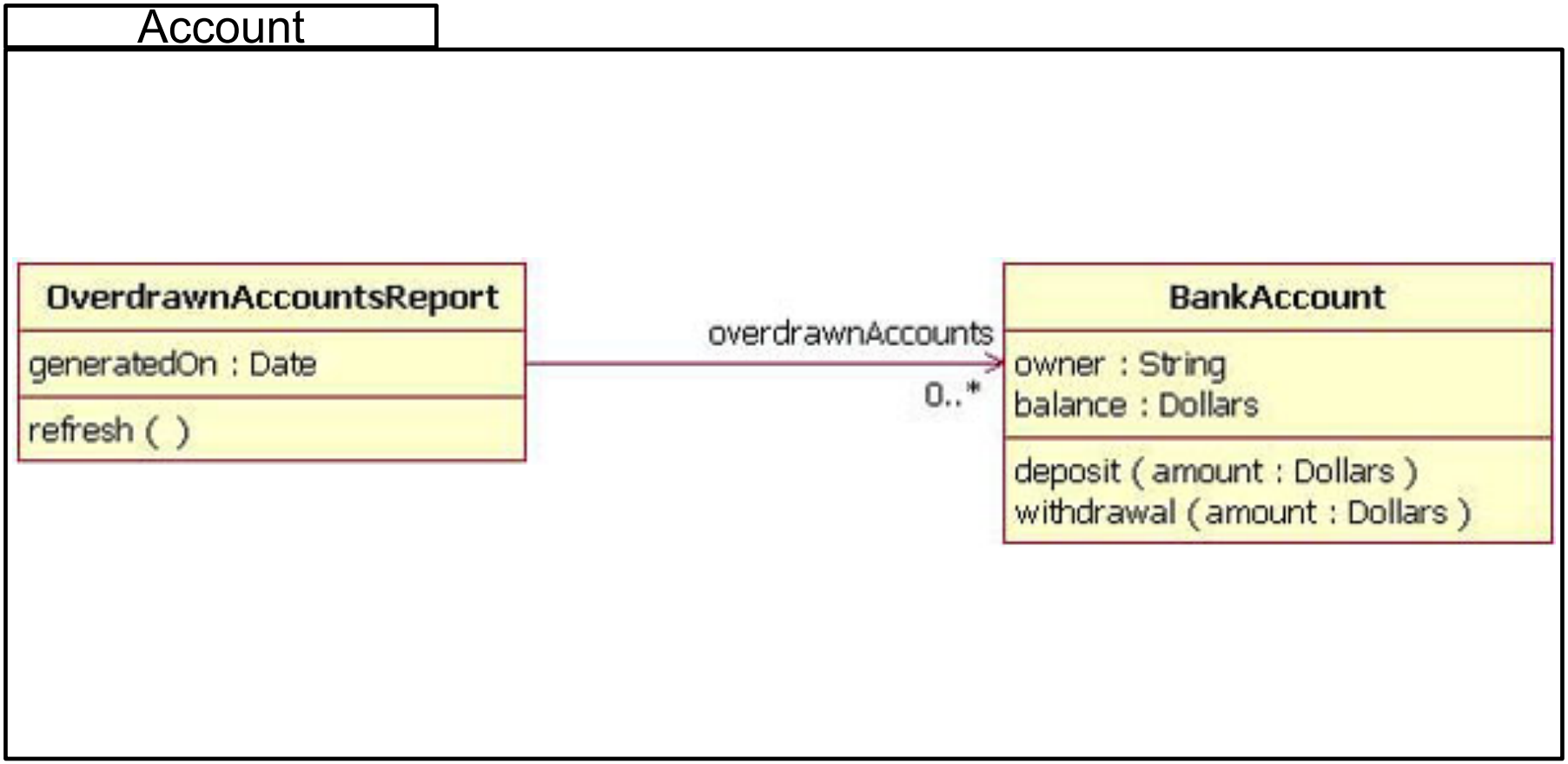
- Rather than show just the interface of a class, class diagrams are primarily designed to show the relationships among classes
- UML has various ways to indicate the information flow from one class object to another using different sorts of ***annotated arrows***
- UML has annotations for class groupings 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



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

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

The package Statement

- 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*

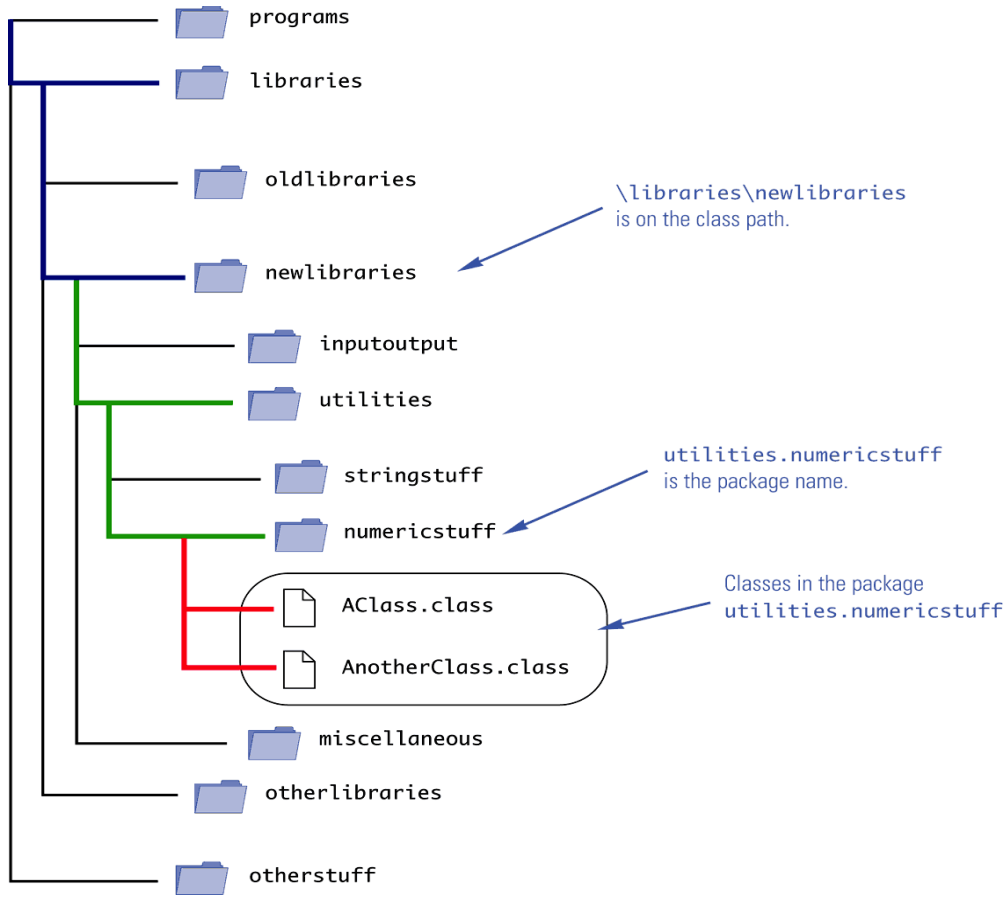
The package java.lang

- 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

Package Names and Directories

- 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

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

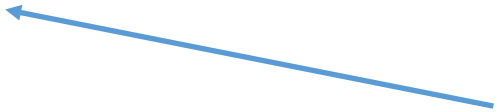
- The class path can be manually specified when a class is compiled
  - Just add `-classpath` followed by the desired class path
  - This will compile the class, overriding any previous `CLASSPATH` setting
- You should use the `-classpath` option again when the class is run

Specifying a Class Path When You Compile

Name Clashes

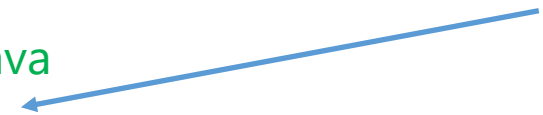
- In addition to keeping class libraries organized, packages provide a way to deal with *name clashes*: a situation in which two classes have the same name
  - Different programmers writing different packages may use the same name for one or more of their classes
  - This ambiguity can be resolved by 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)

- 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

Commenting Classes for 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

- 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

- @ 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

- 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`**

`–link` *Link_To_Other_Docs*	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.
`–d` *Documentation_Directory*	Specifies a directory to hold the documentation generated. *Documentation_Directory* may be a relative or absolute path name.
`–author`	Includes author information (from `@author` tags). This information is omitted unless this option is set.
`–version`	Includes version information (from `@version` tags). This information is omitted unless this option is set.
`–classpath` *List_of_Directories*	Overrides the CLASSPATH environment variable and makes *List_of_Directories* the CLASSPATH for the execution of this invocation of `javadoc`. Does not permanently change the CLASSPATH variable.
`–private`	Includes private members as well as public members in the documentation.

Options for javadoc

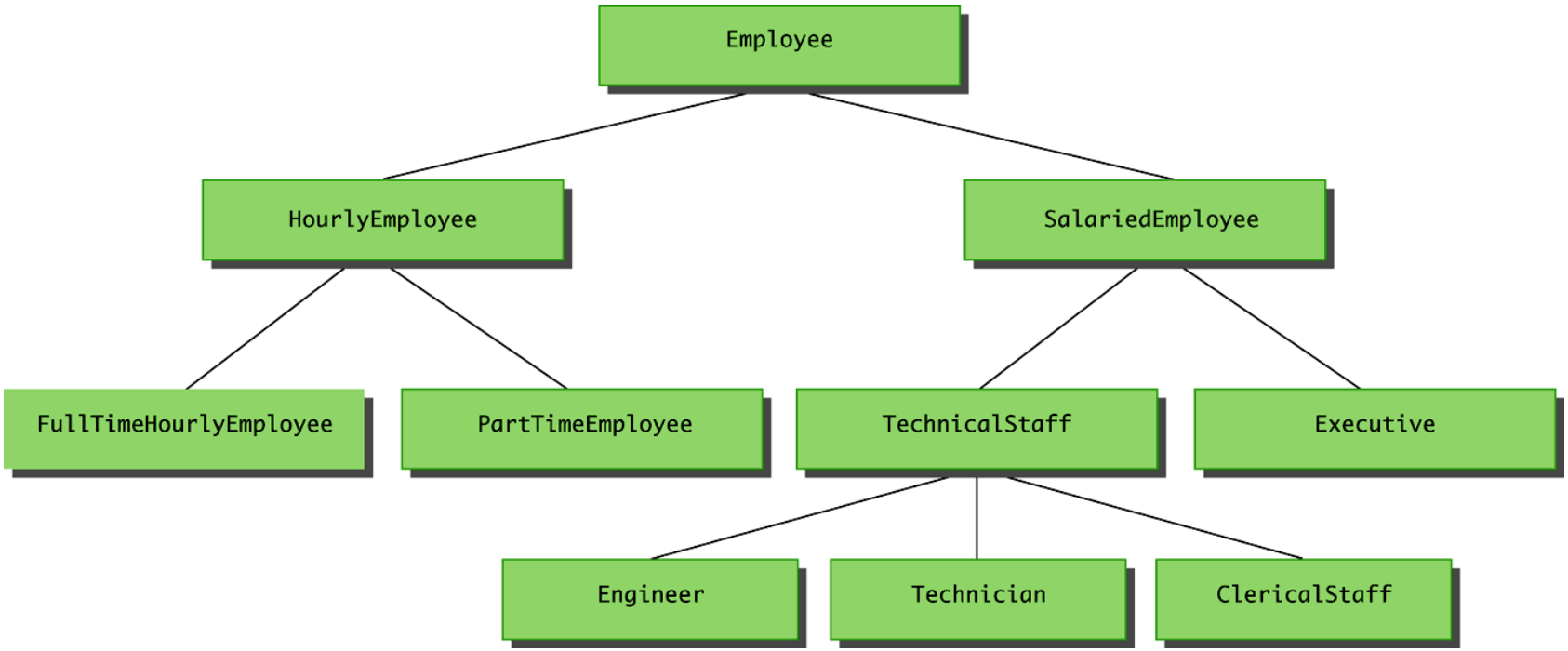
- Introduction to UML & Packages & javadoc
- **Inheritance**
- Access
- Polymorphism
- Abstract Classes

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
```

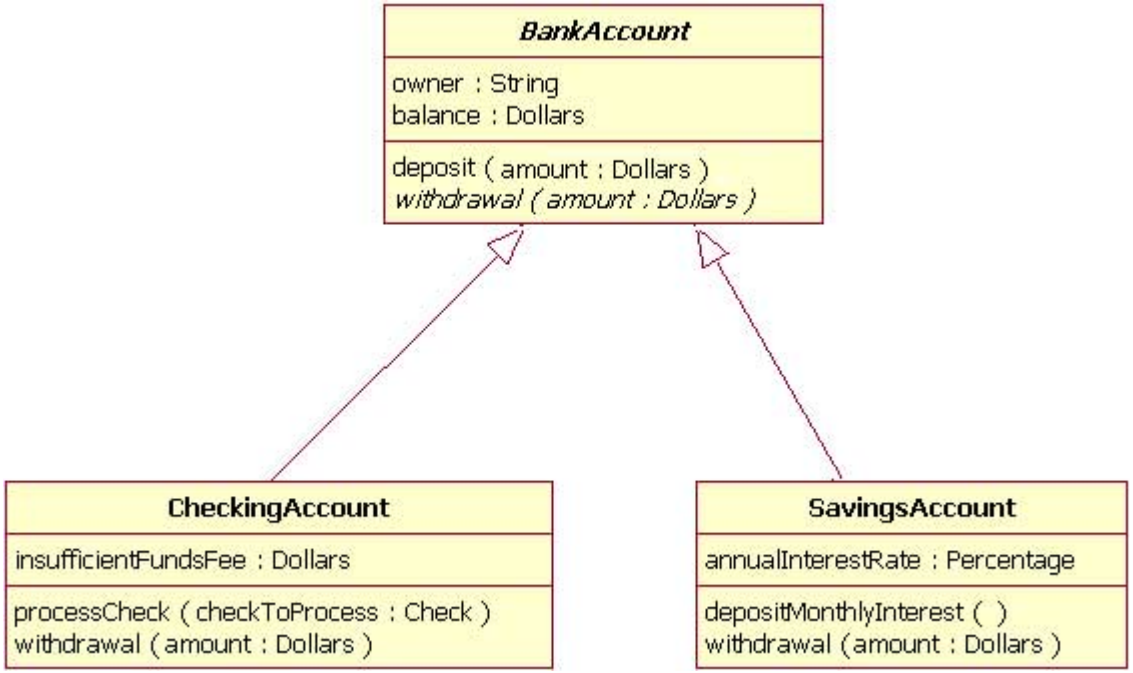
Derived Classes: Syntax

- 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
- It 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

The super Constructor

- 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

- 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

The super Constructor: default

- 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

- 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 anyplace 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, **`EnhancedStringTokenizer`** 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 overriden

An Enhanced StringTokenizer Class

Display 7.7 **EnhancedStringTokenizer**

```
1  import java.util.StringTokenizer;
2
3  public class EnhancedStringTokenizer extends StringTokenizer
4  {
5      private String[] a;
6      private int count;

7      public EnhancedStringTokenizer(String theString)
8      {
9          super(theString);
10         a = new String[countTokens()];
11         count = 0;
12     }

13     public EnhancedStringTokenizer(String theString, String delimiters)
14     {
15         super(theString, delimiters);
16         a = new String[countTokens()];
17         count = 0;
18     }
```

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

(continued)

An Enhanced StringTokenizer Class (Part 1 of 4)

Display 7.7 EnhancedStringTokenizer

```
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.nextTokens` *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 EnhancedStringTokenizer

```
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      {
37          String token = super.nextToken(delimiters);
38          a[count] = token;
39          count++;
40          return token;
41      }
```

This method `nextToken` *also has its definition overridden.*

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

(continued)

An Enhanced StringTokenizer Class (Part 3 of 4)

Display 7.7 EnhancedStringTokenizer

```
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

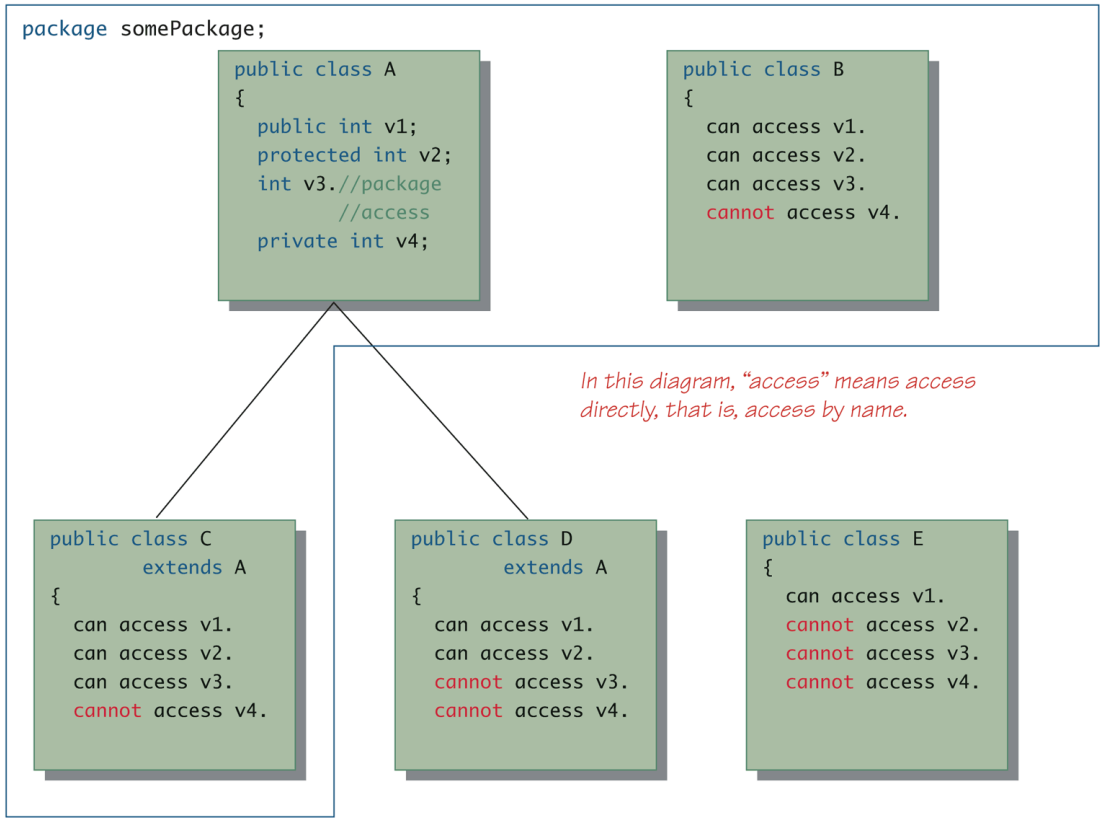
- If a method or instance variable is modified by **`protected`** (rather than **`public`** or **`private`**), then it can be accessed *by name*
  - Inside its own class definition
  - Inside any class derived from it
  - In the definition of any class in the same package
- The **`protected`** modifier 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

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

Display 7.9 **Access Modifiers**



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 form 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

The getClass() Method

- 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())
```

```
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

- 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`**

- 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 instancepf

- 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 instancepf

- **`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 instancepf

- 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);
```

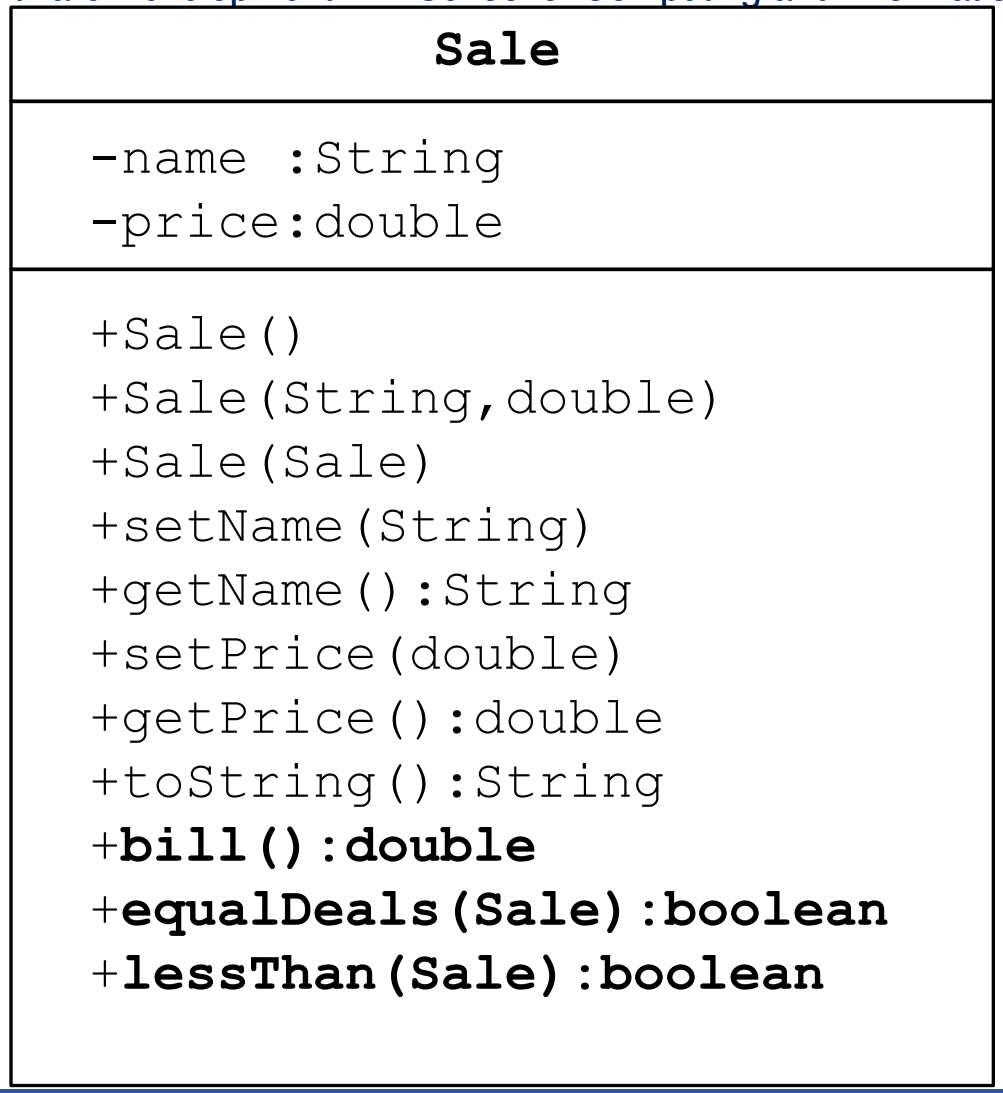
Example

- 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*

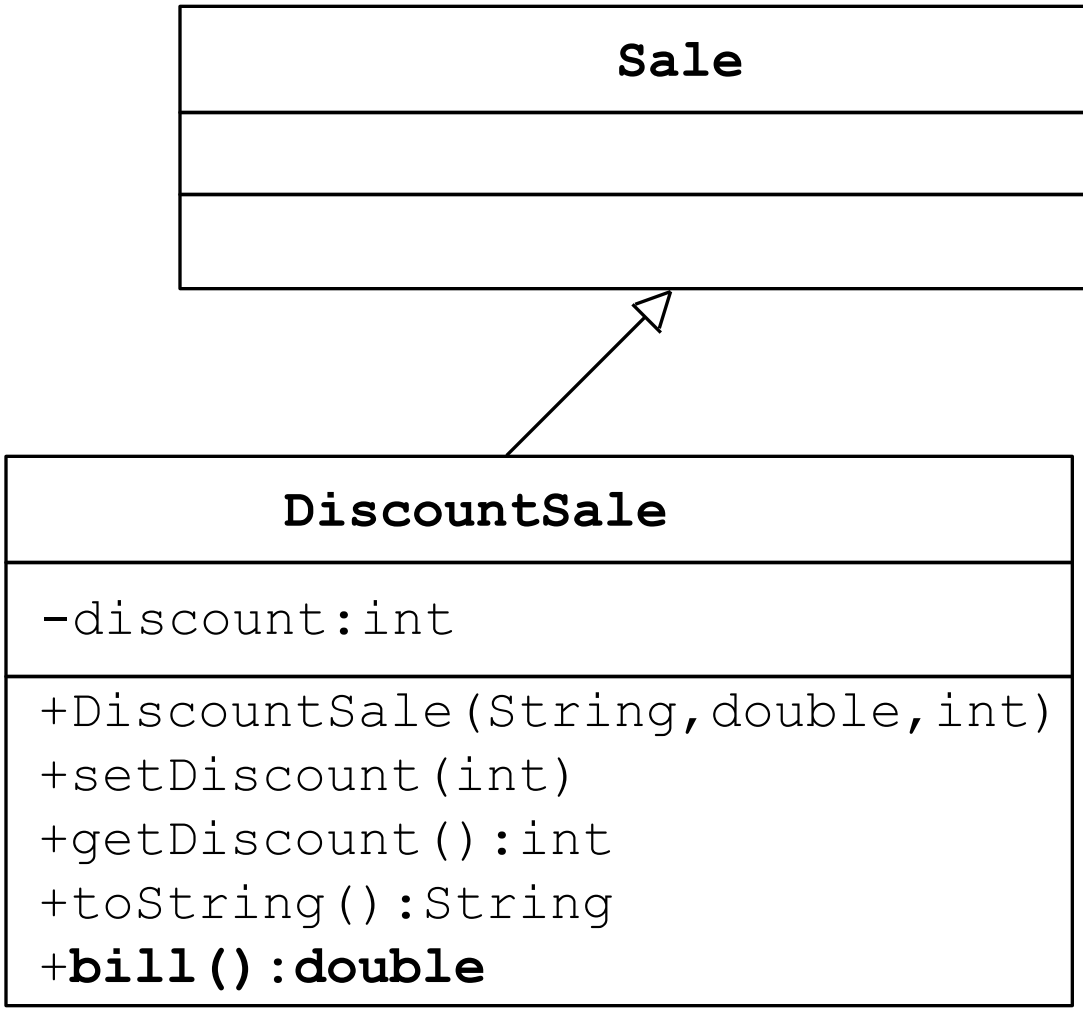
Late Binding

- 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 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

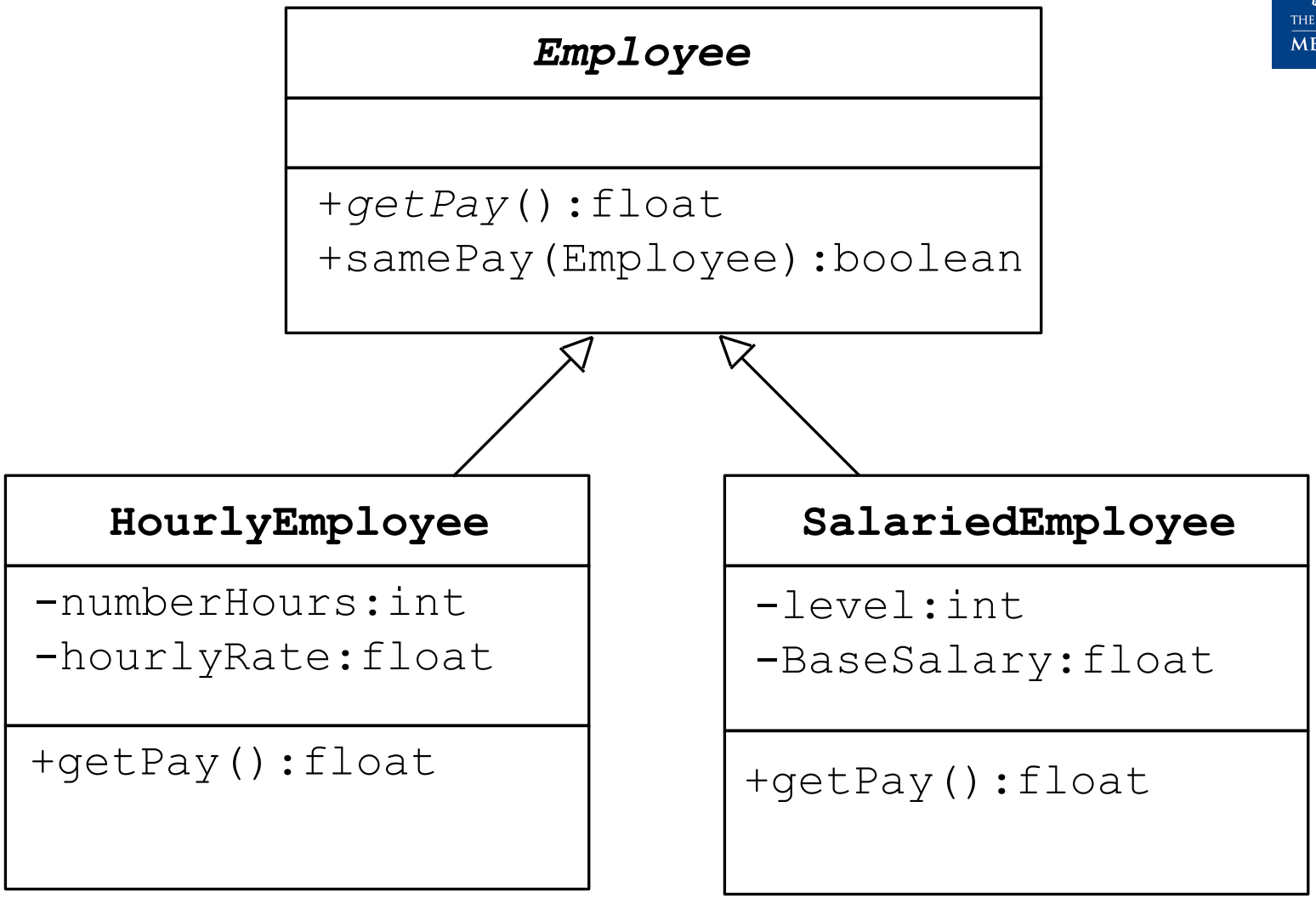
Pitfall: Downcasting

- 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 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***

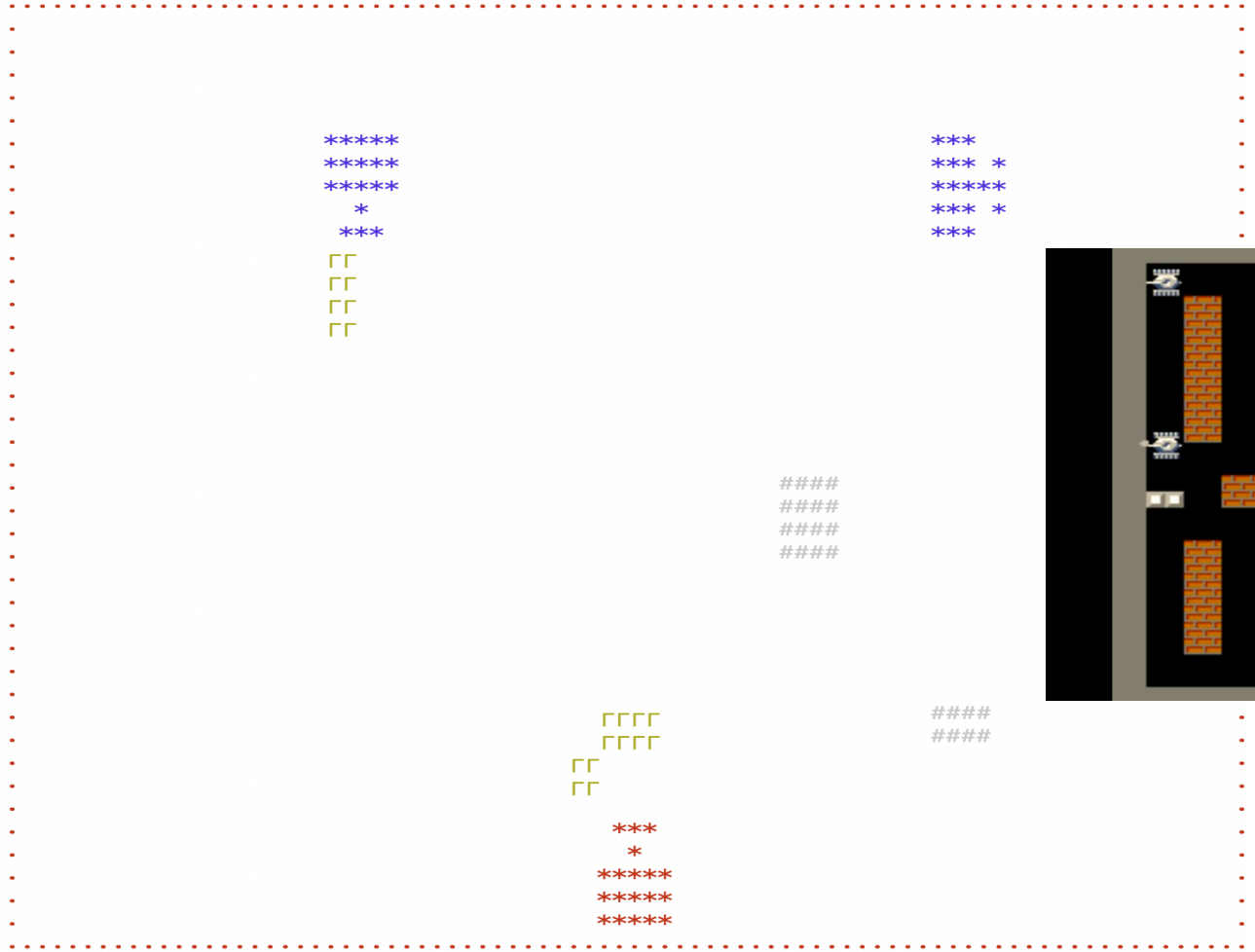
Abstract Class

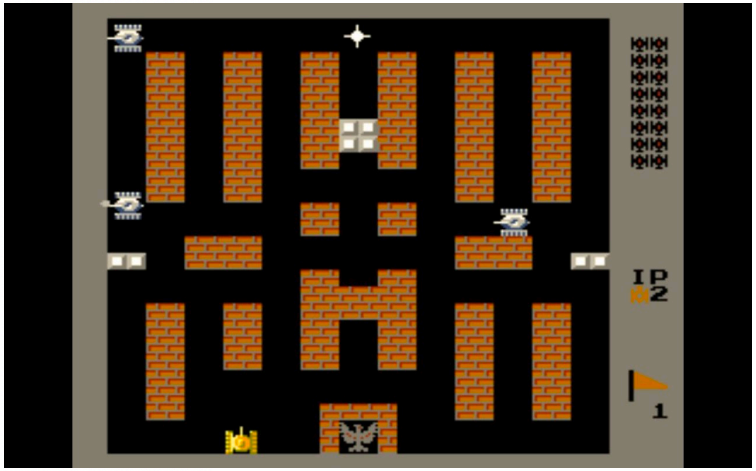
- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, 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

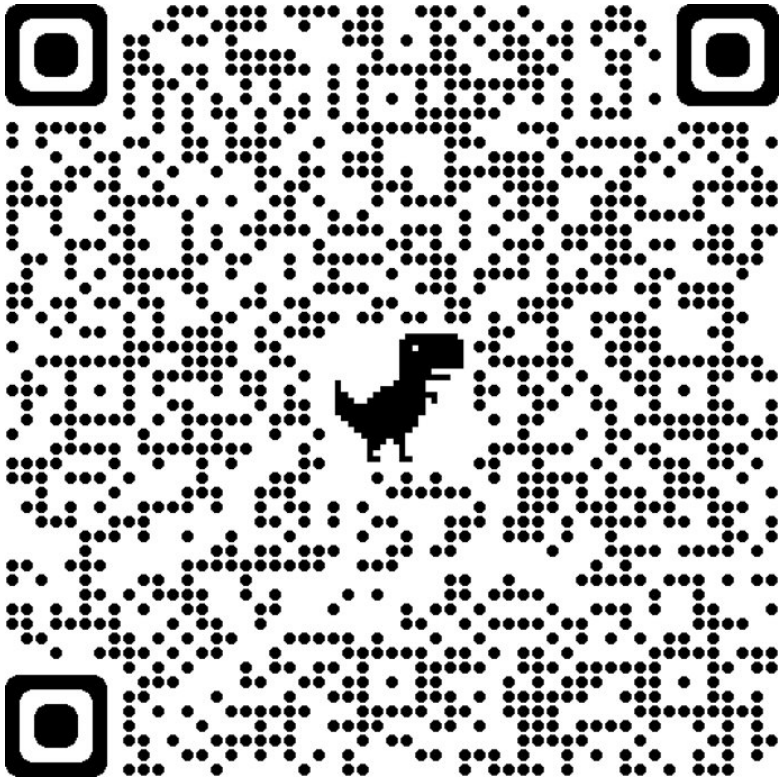




Example – A Simplified Version of The Battle City Tank Game

- 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