CompSci 230 S2
Object Oriented Software Development

Polymorphism

Review Quizzes

- Given the following code, which of these constructors can be added to MySub class without causing a compile-time error?
  - MySub() {}
  - MySub(int cnt) { count = cnt; }
  - MySub(int cnt) { super(); count = cnt; }
  - MySub(int cnt) { count = cnt; super(cnt); }
  - MySub(int cnt) { this(cnt, cnt); }
  - MySub(int cnt) { super(cnt); this(cnt, 0); }

```
class MySuper {
    int number;
    MySuper(int i) { number = i; }
}
```

```
class MySub extends MySuper {
    int count;
    MySub(int cnt, int num) {
        super(num);
        count=cnt;
    }
// ...
}
```


Agenda & Reading

- Topics:
  - Introduction
  - Polymorphism Example
  - The Instanceof operator
  - Packages
  - Access Modifiers
- Reading
  - Java how to program Late objects version (D & D)
    - Chapter 10
  - The Java Tutorial :
    - https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
    - Multiple Inheritance of State, Implementation, and Type
    - Overriding and Hiding Methods
    - Polymorphism
    - Hiding Fields
    - Using the Keyword super
    - Object as a Superclass
    - Writing Final Classes and Methods

1.Introduction

- This occurs when **one** method name in a method call can cause **different actions** depending on which type of instance is invoking the method. !
  - At runtime, the actual method corresponding to the instance type is executed (i.e., the method defined in the class whose constructor created the object)!
- Relying on each object to know how to "do the right thing" in response to the **same method call** is the key concept of **polymorphism**.
- The same message sent to a variety of objects has "**many forms**" of results—hence the term polymorphism.

1.Introduction Extensibility

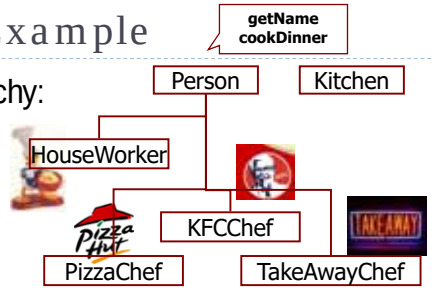
- With polymorphism, we can design and implement systems that are easily **extensible**
  - **New classes** can be **added** with little or no modification to the general portions of the program, as long as the new classes are part of the **inheritance hierarchy** that the program processes generically.
  - The **new** classes simply "plug right in."
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

1.Introduction Polymorphism Definition

- Definition:
  - Polymorphism is the ability of objects belonging to different types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behaviour. The program does not have to know the exact type of the object in advance, so this behavior can be implemented at run time (this is called late binding or dynamic binding).
  - Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics.

2.Polymorphism Example

- Consider the following hierarchy:



- There is a person in my kitchen:
  - The program issues the same message (i.e., cookDinner) to each person object, but each person has it's own way to cook for my dinner.
    - Each object responds in a unique way.
    - The same message (in this case, cookDinner) sent to a variety of objects has "many forms" of results.

2.Polymorphism Example

- Person
  - The Person class is the superclass
    - Instance variables: surname and first name
    - Methods:
      - getName() get surname and first name
      - cookDinner() to prepare food for dinner
        - An ordinary person prepares dinner using a microwave!

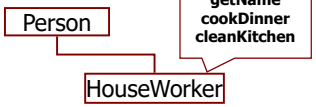
```
class Person {
  protected String surname;
  protected String firstname;
  ...
  public void cookDinner() {
   System.out.println("Microwave Dinner");
  }
}
```

```
Person p = Person("Dick", "Smith");
System.out.println(p.getName());
p.cookDinner();
```

```
Dick Smith
Microwave Dinner
```

2.Polymorphism Example HouseWorker


- Now our Kitchen has a HouseWorker person
  - Inherited method: getName
    - To get her name
  - Additional method: cleanKitchen
    - To clean our kitchen
  - Overridden method: cookDinner
    - A HouseWorker makes Roast chicken with Potato.



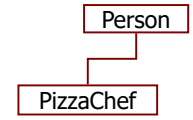

```
p = new HouseWorker("Theresa",
"Thompson");
System.out.println(p.getName());
p.cookDinner();
```

```
class HouseWorker extends Person {
  public HouseWorker(String firstname, String surname)
{
   super(firstname, surname);
  }
  public void cookDinner() {
   System.out.println("Roast Chicken with Potato");
  }
  public void cleanKitchen() {
   System.out.println("Cleaning now");
 }
}
```

Theresa Thompson
Roast Chicken with Potato

2.Polymorphism Example PizzaChef


- Now our Kitchen has a PizzaChef person
  - Inherited method: getName
    - To get her name
  - Overridden method: cookDinner
    - A pizzachef makes Pizza




```
p = new PizzaChef("Michael", "Hill");
System.out.println(p.getName());
p.cookDinner();
```

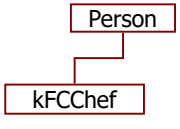
```
class PizzaChef extends Person {
  public PizzaChef(String firstname, String surname) {
   super(firstname, surname);
  }

  public void cookDinner() {
   System.out.println("Pizza");
  }
}
```

Pizza

2.Polymorphism Example KFCChef


- Now our Kitchen has a KFCChef person
  - Inherited method: getName
    - To get her name
  - Overridden method: cookDinner
    - A KFCChef makes KFC fried chicken




```
p = new KFCChef("Peter", "Wong");
System.out.println(p.getName());
p.cookDinner();
```

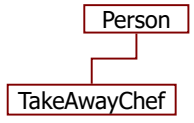
```
class KFCChef extends Person {
  public KFCChef(String firstname, String surname) {
   super(firstname, surname);
  }

  public void cookDinner() {
   System.out.println("KFC Chicken");
  }
}
```

Peter Wong
KFC Chicken

2.Polymorphism Example TakeAwayChef

- Now our Kitchen has a TakeAwayChef person
  - Inherited method: getName
    - To get her name
  - Overridden method: cookDinner
    - A TakeAwaychef makes fried rice




```
p = new TakeAwayChef("Kevin", "Chan");
System.out.println(p.getName());
p.cookDinner();
```

```
class TakeAwayChef extends Person {

  public TakeAwayChef(String firstname, String surname) {
   super(firstname, surname);
  }

  public void cookDinner() {
   System.out.println("Fried Rice");
  }
}
```

Kevin Chan
Fried Rice

2.Polymorphism Example
An Array of Person objects

- Now, it is more interesting if we have an array of superclass variables that refer to objects of many subclass types.
  - This example demonstrates that an object of a subclass can be treated as an **object of its superclass**, enabling various interesting manipulations.
  - For example:
    - We have an array holding five objects of type **Person**.
    - Because of their inheritance relationship with the Person class, the HouseWorker, PizzaChef, KFCChef and TakeAwayChef classes can be assigned to the array.
    - Within the for-loop, the **cookDinner** method is invoked on each element of the array.





```
Person[] pList = new Person[5];
pList[0] = new Person("Dick", "Smith");
pList[1] = new HouseWorker("Theresa",
"Thompson");
pList[2] = new PizzaChef("Michael", "Hill");
pList[3] = new KFCChef("Peter", "Wong");
pList[4] = new TakeAwayChef("Kevin", "Chan");
for (int i=0; i<pList.length; i++)
  pList[i].cookDinner();
```

The result depends on the object stored, not on the type of the variable.

2.Polymorphism Example
Compile time

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this "crossover" because an object of a subclass is an object of its superclass (but not vice versa).
- Type checking is done at compile time:
  - When the compiler encounters **a method call** made through a variable, the **compiler** determines if the method can be called by checking the variable's class **type**.
    - If that class contains the proper method declaration or inherits one, i.e. if the method does not exist, then check the superclass (work from the child to the parent class),
    - the call is compiled.
    - For example:

Can we find the cookDinner defined in the Person(the type of the variable) class?

```
Person[] pList = new Person[5];
...
for (int i=0; i<pList.length; i++)
  pList[i].cookDinner();
```


2.Polymorphism Example
Execution time

- At execution time, the type of the object to which the variable refers determines the **actual** method to use.
  - This process is called dynamic binding.
  - i.e. the method is executed based on the object, not on the type of variable.
    - The derived classes override the cookDinner method of the Person class. This makes the overridden cookDinner() methods of the **derived** classes execute when the cookDinner() method is called using the base class reference from the array

Invoke the cookDinner from the Person class
Invoke the cookDinner from the HouseWorker class
and so on…

```
pList[0] = new Person("Dick", "Smith");
pList[1] = new HouseWorker("Theresa",
"Thompson");
pList[2] = new PizzaChef("Michael", "Hill");
pList[3] = new KFCChef("Peter", "Wong");
pList[4] = new TakeAwayChef("Kevin", "Chan");
```


2.Polymorphism Example
Compile-time Error?

- Note:The cleanKitchen method is only defined in the HouseWorker class.
  - For example, there are two objects:

```
HouseWorker w = new HouseWorker("Theresa",
"Thompson");
Person p1 = new HouseWorker("Theresa",
"Thompson");
```

a subclass object with a **subclass** reference

a subclass object with a **superclass** reference

  - Method Calls:

```
w.cookDinner();
w.cleanKitchen();
```

```
p1.cookDinner();
p1.cleanKitchen(); //Compile erro
```

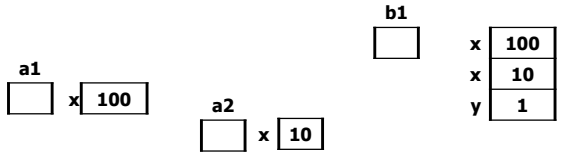
No cleanKitchen method declared in Person class.

  - A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.
    - Java decides which class's cookDinner method to call at execution time rather than at compile time
  - Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

Exercise 1

- What is the output?

```
A a1 = new A();
System.out.println(a1);
A a2 = new A(10);
System.out.println(a2);
B b1 = new B();
System.out.println(b1);
```




```
class A {
  int x;
  public A() {
    this(100);
  }
  public A(int x) {
    this.x = x;
  }
  public String toString() {
    return "A:" + x;
  }
}
```

```
class B extends A {
  int x = 10;
  int y = 1;
  public B() {}
  public B(int y) {
    this.y = this.y + y;
  }
  public B(int x, int y) {
    super(x);
    this.y = this.y + y;
  }
  public String toString() {
    return "A:" + super.x + ", B:("+x+","+y + ")";
  }
}
```


Exercise 2

- What is the output of the following program?

```
public class L11Ex2{
  public static void main( String args[] ) {
    Cat      theCat      = new Cat( );
    Dog      theDog      = new Dog( );
    Frog     theFrog     = new Frog( );
    Flamingo theFlamingo = new Flamingo( );

    System.out.println("The CAT says :" + theCat.speak());
    System.out.println("The DOG says :" + theDog.speak());
    System.out.println("The FROG says :" + theFrog.speak());
    System.out.println("The FLAMINGO says :" + theFlamingo.speak());
  }
}
```


Exercise 2: Structure

- Pet, Dog, Cat etc

```
class Pet {
  String speak()  {
    return "Hi, I'm a happy and contented pet";
  }
}
```

```
class Dog extends Pet {
  String speak()  {
    return super.speak() + ". Arf, Arf";
  }
}
```

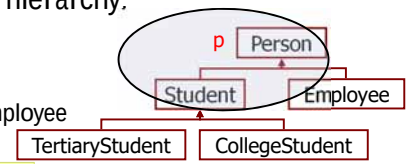
```
class Cat extends Pet {
  String speak()  {
    return super.speak() + ". Meow, meow";
  }
}
```

```
class Frog extends Pet{}
```

```
class Flamingo extends Pet {
  String speak() {
    return "They're holding me captive. You must help!";
  }
}
```


2.The instanceof operator

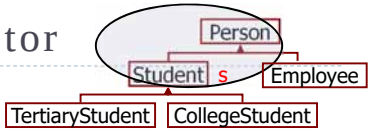
- The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).
  - The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either **true** or **false**.
  - If we apply the instanceof operator with any variable that has null value, it returns false.
- For example, consider the following hierarchy:
  - Declare and create a Person object, p.
    - p is an instance of Person
    - But p is NOT an instance of student/Employee




```
Person p = new Person();
System.out.println(p instanceof Person);
System.out.println(p instanceof Student);
System.out.println(p instanceof Employee);
```

```
true
false
false
```

2.The instanceof operator



- Declare and create a Student object: s.
  - s is an instance of Person/Student
  - But Student and Employee are incompatible types
    - i.e. we can't use instanceof between s and Employee class

```
Student s = new Student();
System.out.println(s instanceof Person);
System.out.println(s instanceof Student);
//System.out.println(s instanceof Employee);
```

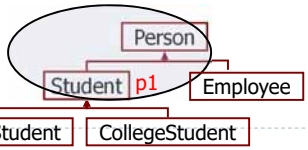
```
true
true
```

```
Error: incompatible types...
```

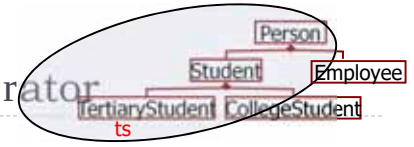
- Create a Student object but with a Person reference
  - P1 is an instance of Person/Student
  - But p1 is NOT an instance of Employee
    - But p1 declared as Person therefore we can use the instanceof operator between p1 and Employee

```
Person p1 = new Student();
System.out.println(p1 instanceof Person);
System.out.println(p1 instanceof Student);
System.out.println(p1 instanceof Employee);
```

```
true
true
false
```



2.The instanceof operator



- Declare and create a TertiaryStudent object
  - *ts* is an instance of Person/Student/TertiaryStudent
  - But TertiaryStudent and CollegeStudent are incompatible types
  - But TertiaryStudent and Employee are incompatible types

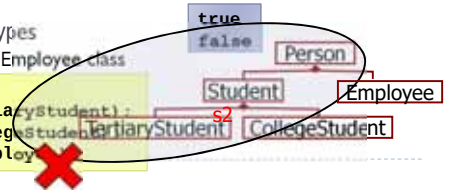
```
TertiaryStudent ts = new TertiaryStudent();
System.out.println(ts instanceof Student);
// System.out.println(ts instanceof CollegeSt...);
// System.out.println(ts instanceof Employee);
```

```
true
```

- Create a TertiaryStudent object but with a Student reference
  - s2 is an instance of Person/Student/TertiaryStudent
  - s2 is NOT an instance of CollegeStudent
    - But s2 declared as Student therefore we can use the instanceof operator between s2 and **CollegeStudent**
  - Student and Employee are incompatible types
    - i.e. we can't use instanceof between s2 and Employee class

```
Student s2 = new TertiaryStudent();
System.out.println(s2 instanceof TertiaryStudent);
System.out.println(s2 instanceof CollegeStudent);
//System.out.println(s2 instanceof Employee);
```

```
true
false
```



3.Packages

- **Definition:** A *package* is a grouping of related types providing access protection and name space management.
  - A package is a namespace that organizes a set of related classes and interfaces."
- **Purpose:** "To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages."
  - Conceptually you can think of packages as being similar to different folders on your computer.
  - Note that types refers to classes, interfaces …
- Examples:
  - java.io — file operations
  - java.lang — basic language functionality and fundamental types
  - java.util — collection data structure classes
  - java.awt — basic hierarchy of packages for native GUI components

3.Packages
Creating a Package

- "To create a package, you
  - choose a name for the package (folder) and
  - put a package statement with that name at the top of *every source file* that contains the types that you want to include in the package.

```
package X;
```

- Rules:
  - There can be only one package statement in each source file, and must be the first line in the source file.
    - If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.
    - You can include non-public types in the same file as a public type

This rule makes it easy for the class loader, and the human programmer, to find the definition for a public type.

3.Packages
One public type per file!

- "If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.
  - For example, you can
    - define **`public class Circle`** in the file **`Circle.java`**,
    - define **`public interface Draggable`** in the file **`Draggable.java`**,
    - define **`public enum Day`** in the file **`Day.java`**, and so forth.
- "You can include non-public types in the same file as a public type
  - (this is strongly discouraged, unless the non-public types are small and closely related to the public type),
  - but only the public type will be accessible from outside of the package.
  - All the top-level, non-public types will be *package private."*
- This rule makes it easy for the class loader, and the human programmer, to find the definition for a public type.
  - The **name** of a package determines the **directory** in which the files of this package *should* be stored.
  - The **name** of a public type determines the **name** of the file in which the type's definition *must* be found."

3.Packages
The default package

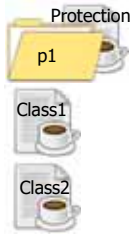
- "If you do not use a package statement, your type ends up in an unnamed package.
  - Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process.
  - Otherwise, classes and interfaces belong in named packages."

3.Packages
Package naming conflicts

- "With programmers worldwide writing classes and interfaces using the Java programming language,
  - it is likely that many programmers will use the same name for different types.
  - The compiler allows both classes to have the same name if they are in different packages.
- The fully qualified name of each **`Rectangle`** class includes the package name.
  - That is, the fully qualified name of the **`Rectangle`** class in the **`graphics`** package is **`graphics.Rectangle`**, and
  - the fully qualified name of the **`Rectangle`** class in the **`java.awt`** package is **`java.awt.Rectangle`**.

3.Packages
External references

- To use a public package member from outside its package, you must do one of the following:
  - The fully qualified name for class C in package p1 is p1.C
  - To import class C from package p1, you write import p1.C
  - To import an entire package p1, you write import p1.*
- Example:
  - Directory/package: p1
    - class: Protection
  - class: Class1
  - class: Class2




```
package p1;
public class Protection {
  int n_default = 1;
  public int n_public = 2;
  protected int n_protected = 3;
  private int n_private = 4;
}
```

```
public class Class1 {
  public static void main(String[] args) {
    p1.Protection c = new p1.Protection();
  }
}
```

```
import p1.*;
public class Class2 {
    public static void main(String[] args) {
        Protection c = new Protection();
    }
}
```

4.Access Modifiers

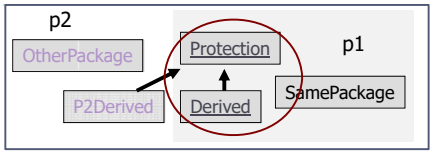
- Classes, and their fields and methods have access levels to specify how they can be used by other objects during execution
  - A **private** field or method is accessible only to the class in which it is defined.
  - A **protected** field or method is accessible to the class itself, its subclasses, and classes in the same package.
  - A **public** field or method is accessible to any class of any parentage in any package
  - **Default**: When no access modifier is specified – It is said to be having the default access modifier by default. i.e. having default access modifier are accessible only within the same package.

	Private	Protected	Public	Default
Class itself	**YES**	**YES**	**YES**	**YES**
Other Subclasses within a package	**No**	**YES**	**YES**	**YES**
Other classes within a package	**No**	**YES**	**YES**	**YES**
Other subclasses outside this package	**No**	**YES**	**YES**	**No**
Other classes outside this package	**No**	**No**	**YES**	**No**

4.Access Modifiers
Subclasses within a package

p1/Derived.java

```
package p1;
public class Protection {
  int n_default = 1;
  public int n_public = 2;
  protected int n_protected = 3;
  private int n_private = 4;
}
```



Within a package

javac p1/*.java
java p1.Derived

```
package p1;
public class Derived extends Protection {
  public static void main(String[] args) {
    Derived b = new Derived();
    System.out.println("n_default=" + b.n_default);
    System.out.println("n_public=" + b.n_public);
    System.out.println("n_protected=" + b.n_protected);
    // System.out.println("n_private" + b.n_private);
  }
}
```

```
n_default=1
n_public=2
n_protected=3
```

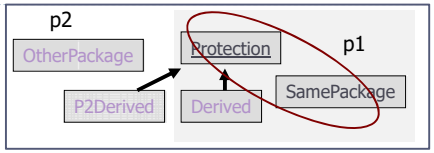
No access to Private variable

	Private	Protected	Public	Default
Other Subclasses within a package	No	YES	YES	YES

4.Access Modifiers
Other classes within a package

p1/SamePackage.java

```
package p1;
public class Protection {
  int n_default = 1;
  public int n_public = 2;
  protected int n_protected = 3;
  private int n_private = 4;
}
```



With a package

```
n_default=1
n_public=2
n_protected=3
```

javac p1/*.java
java p1.SamePackage

```
package p1;
public class SamePackage {
  public static void main(String[] args) {
    Protection b = new Protection();
    System.out.println("n_default=" + b.n_default);
    System.out.println("n_public=" + b.n_public);
    System.out.println("n_protected=" + b.n_protected);
   // System.out.println("n_private" + b.n_private);

  }
}
```

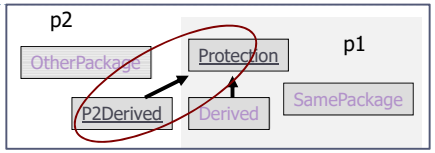
No access to Private variable

	Private	Protected	Public	Default
Other classes within a package	No	YES	YES	YES

4.Access Modifiers
Subclasses outside this package

p2/P2Derived.java

```
package p1;
public class Protection {
  int n_default = 1;
  public int n_public = 2;
  protected int n_protected = 3;
  private int n_private = 4;
}
```



Different packages

```
n_public=2
n_protected=3
```

```
package p2;
public class P2Derived extends p1.Protection {
   public static void main(String[] args) {
     P2Derived b = new P2Derived();
    //System.out.println("n_default=" + b.n_default);
    System.out.println("n_public=" + b.n_public);
    System.out.println("n_protected" + b.n_protected);
    //System.out.println("n_private" + b.n_private);
  }
}
```

No access to the default variable

No access to Private variable

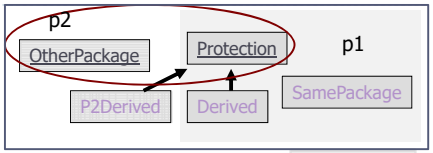
javac p1/*.java
javac p2/*.java
java p2.P2Derived

	Private	Protected	Public	Default
Other subclasses outside this package	No	YES	YES	No

4.Access Modifiers
Other classes outside this package

p2/OtherPackage.java

```
package p1;
public class Protection {
  int n_default = 1;
  public int n_public = 2;
  protected int n_protected = 3;
  private int n_private = 4;
}
```



Different packages

```
package p2;
public class OtherPackage {
 public static void main(String[] args) {
  p1.Protection b = new p1.Protection();
  // System.out.println("n_default=" + b.n_default);
  System.out.println("n_public=" + b.n_public);
  // System.out.println("n_protected" + b.n_protected);
  // System.out.println("n_private" + b.n_private);
  }
}
```

No access to
- **Default**
- **protected**
- **private**

javac p1/*.java
javac p2/*.java
java p2.OtherPackage

	Private	Protected	Public	Default
Other classes outside this package	No	No	YES	No

Exercise 3

- What is the output of the following program?

```
public class L11Ex3{
  public static void main( String args[] ) {
    Vehicle b = new Car (); // Vehicle reference but Car object
    b.move (); //Calls the method in Car class
  }
}

class Vehicle {
  public void move () {
  System.out.println ("Vehicles are used for moving from one place to another ");
  }
}

class Car extends Vehicle {
    public void move () {
      super. move (); // invokes the super class method
      System.out.println ("Car is a good medium of transport ");
    }
}
```