CH. 2 OBJECT-ORIENTED PROGRAMMING

OBJECT-ORIENTED DESIGN PRINCIPLES

- **Object Oriented Programming** – paradigm for programming involving modularizing code into self contained **objects** that are a concise and consistent view of a "thing" without exposing unnecessary detail like the inner workings of the object
  - Abstraction – What makes up an object? The model
  - Composition – Objects can own other objects, "has-a" relationships
  - Encapsulation – Hiding implementation details, only exposing the "public interface"
  - Inheritance – Types and subtypes, "is-a" relationships
  - Polymorphism – Provision of a single interface to entities of different types

OBJECT-ORIENTED DESIGN PRINCIPLES
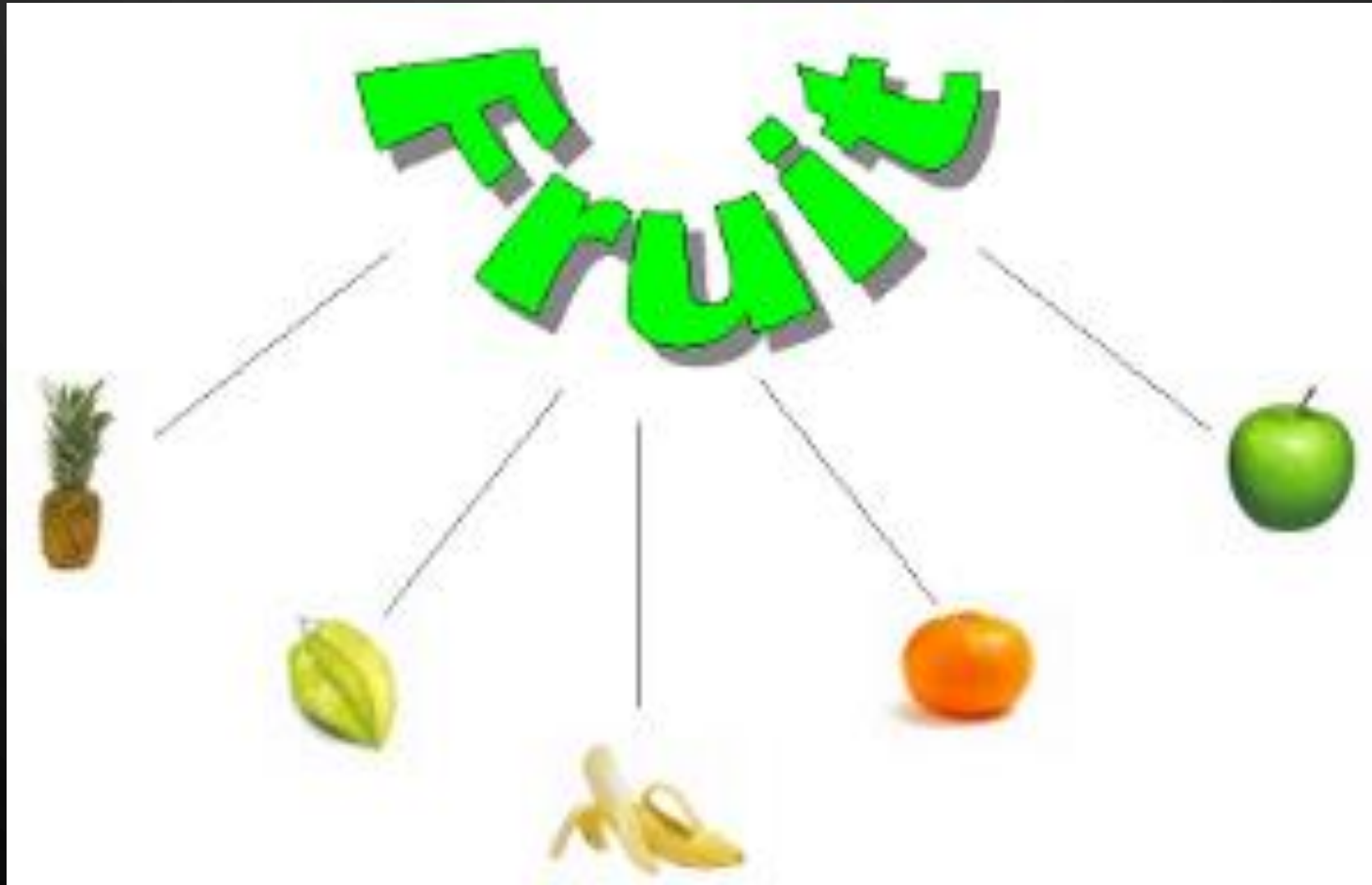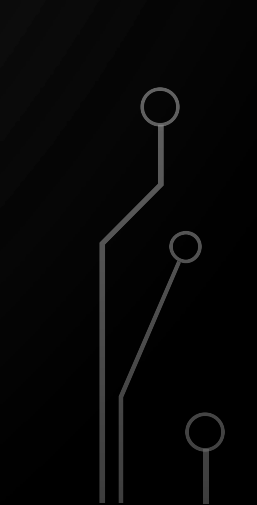
GOALS

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.

- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.

- Reusability
  - The same code should be usable as a component of different systems in various applications.
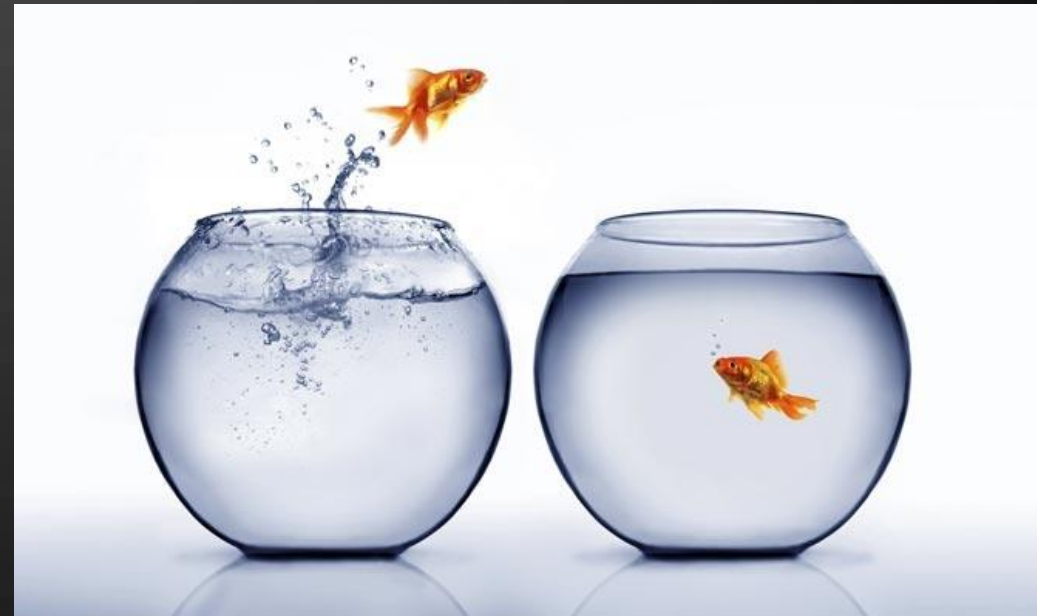
OBJECT-ORIENTED SOFTWARE DESIGN

- Responsibilities
  - Divide the work into different actors, each with a different responsibility.

- Independence
  - Define the work for each class to be as independent from other classes as possible.

- Behaviors
  - Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.
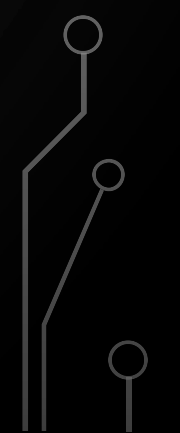
CRASH COURSE IN USING AND MAKING OBJECTS

REVIEW OF CMSC 150

TERMINOLOGY

- Object type, i.e., **class** – specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute

- Object **instance**, i.e. **object** – variable of that object type

USING A CLASS (QUICK AND DIRTY REFRESHER)

- Initialize a variable of an object with the keyword **new** followed by a call to a **constructor** of the object:

  `String s = new String("Hello");`

- Use a method of the class to execute a computation:

  `int l = s.length();`

CLASS DEFINITIONS

- A class serves as the primary means for abstraction in object-oriented programming.

- Data fields, i.e., members – defines the state of an object instance and its size/layout in memory

  - Can be primitive types or other objects (composition – "has-a" relationship)

- Member functions, i.e., methods – set of behaviors that act upon the state of an object instance

CLASS TEMPLATE (QUICK AND DIRTY REFRESHER)

```
1. public class ClassName {
2.     /* All instance variables declared private*/
3.     private int i = 0;
4.     /* Any public static final variables – these model constants */
5.     /* All constructors – constructors initialize all member data,
          and must be named the same as the class */
6.     public ClassName() {}
7.     /* All accessor (getters) and simple modifiers (setters) needed
          for the object */
8.     public int getI() {return i;}
9.     public int setI(int i) {this.i = i;}
10.    /* All other public methods */
11.    /* Any and all private methods */
12.    /* Any and all static methods */
13.}
```

EXAMPLE

- Lets program (in pairs) a class for a bank account, Account.java
  - Have getters and setters for private member data (name and balance)
  - Have a deposit method to operate on an account
- Program a simple test to exercise all of the methods of circle
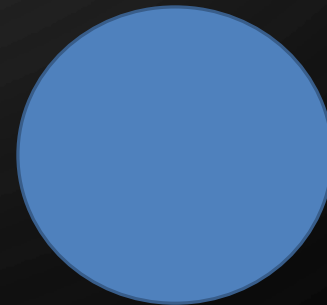
ABSTRACT DATA TYPES

ABSTRACT DATA TYPES

- **Abstraction** is to distill a system to its most fundamental parts.

- An **abstract data type (ADT)** is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.

  - This would essentially be the "public interface" of a class

- *An ADT specifies what each operation does, but not how it does it*

  - Lets repeat, an ADT is the operations not the implementation!

  - We will see that we can implement ADTs in many, many ways

NESTED CLASSES

- Java allows a class definition to be nested inside the definition of another class.

- The main use is in defining a class that is strongly affiliated with another class to increase encapsulation

- Nested classes are a valuable technique when implementing data structures. A instance of the nested class could represent:
  - A small portion of the larger data structure
  - An auxiliary class to help navigation of the data structure.

```java
public class A {
    // Can be public or private
    // Can be static or non-static
    public class B {
    }
    // We will use this form
    // most often
    private static class C {
    }
}
```

INHERITANCE

MOTIVATIONS

- Suppose you will want to model objects for shapes. Many of the objects will have common features, maybe colors, or the ability to compute their areas, or computing overlap between them. BUT, is there a way to reduce the amount of repeated code? Improve the robustness (correctness) of the model? Design this type of model hierarchy?

- How about an example of allied characters in a game? Some help you by healing, some help offensively, some help defensively. However, all of these types of allies have commonality. So the same questions exist!

- The answer is to use **inheritance** – modeling types and subtypes in a way that reduces duplicated components
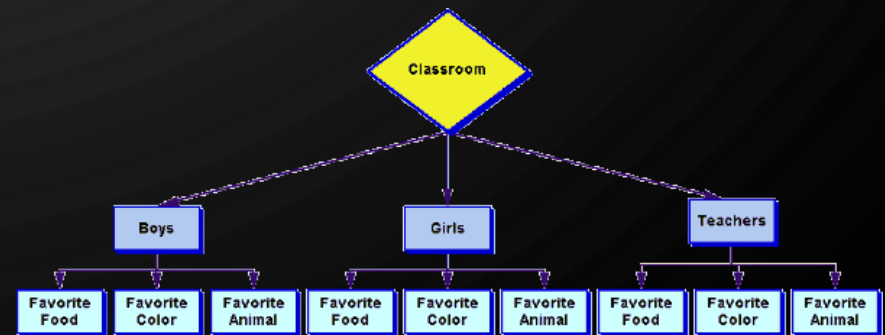
INHERITANCE

- **Inheritance** is a type/sub-type relationship (parent/child) denoted with an arrow pointed to the type in a UML diagram
  - A **superclass (base class)** is the inherited object type
  - A **subclass (derived class)** is the inheriting object type
  - All of the state (data fields) and behavior (methods) of the superclass is inherited ("handed-down") to the subclass
    - The superclass constructors are NOT inherited

GeometricObject

- color: String

+ GeometricObject(color: String)
+ getColor(): String
+ setColor(color: String): void
+getArea(): double

Circle

- radius: double

+ Circle(color: String, radius: double)
+ getRadius(): double
+ setRadius(radius: double): void
+ getArea(): double
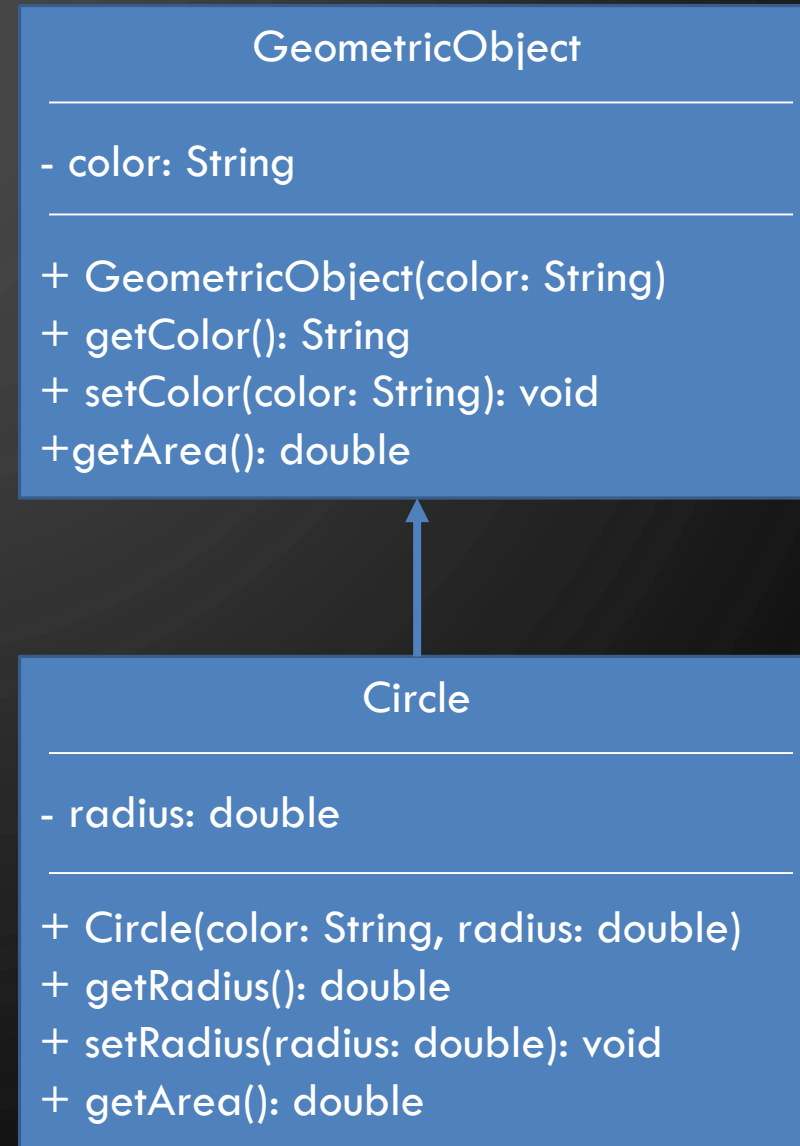
INHERITANCE IN JAVA

```
1. public class A {
2.    private int a;
3. }
4. public class B extends A {
5.    private int b;
6. }
```

Memory

A

B

Object of type A

int a

Object of type B

int a   int b
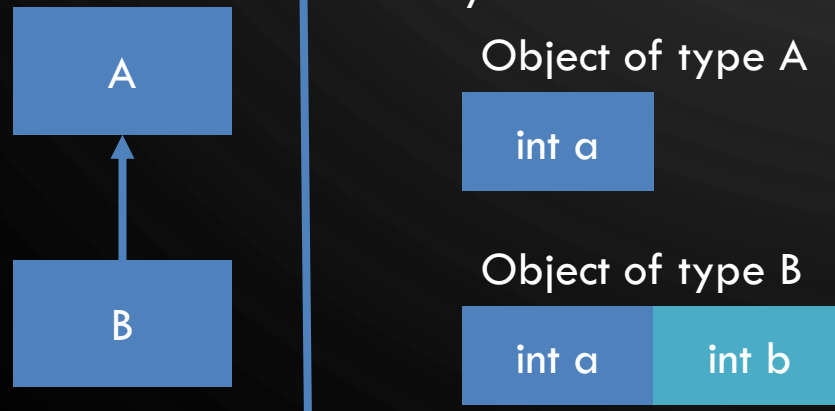
- In Java, the keyword **extends** denotes an inheritance relationship

- In this example, by inheritance **B** is an object whose state is defined by two **int**s, the one in **A** and the one in **B**

- In this relationship, the superclass is responsible for constructing (initializing) the superclass's data fields, while the subtype is responsible for the subclass's data fields

CONSTRUCTION IN INHERITANCE

- The superclass constructor is not inherited, so how do we construct it's part of memory?

- They are invoked explicitly (by the programmer) or implicitly (by the Java compiler)

- We use the **super** keyword to invoke explicitely

- The Java compiler will always attempt to invoke the no-arg constructor implicitely

- Caveats:
  - We must use the keyword super, otherwise error
  - It must be the very first line of the constructor, otherwise error

- Explicitely:

```
public B() {
    super(); //note this is like any
             //constructor, we are
             //free to pass
             //parameters as well!
}
```

- Implicitely:

```
public B() {
    //java inserts super() – always
    //calling the no-arg constructor
}
```

SUPER

- A reference to the superclass
  - Synonymous to `this`
- Can be used to
  - Call superclass constructor
  - Call methods/data fields of superclass

```
1.  public class A {
2.     int x;
3.     public A(int a) {x = a;}
4.     public void printA() {System.out.print(x);}
5.  }
6.  public class B extends A {
7.     int y;
8.     public B(int a, int b) {
9.        super(a); //Example of construction
10.       y = b;
11.    }
12.    public void printB() {
13.       super.printA(); //Example of method
                         //invocation
14.       System.out.print(", " + y);
15.    }
16.}
```

DEFINING A SUBCLASS

- A subclass inherits from a superclass. You can also:
  - Add new properties
  - Add new methods
  - Override the methods of the superclass
- Conceptually a subclass represents a smaller set of things, so we make our subclass *more detailed* to model this

Super

Sub
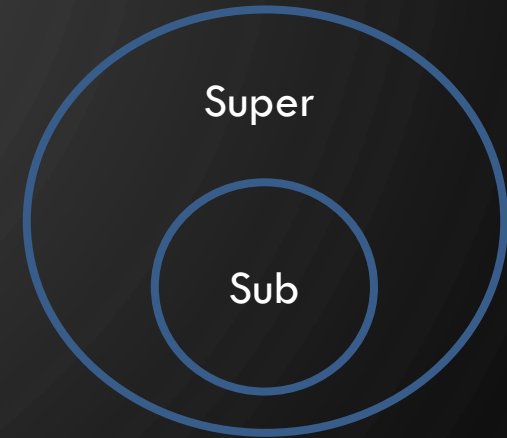
OVERRIDING

- A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as **method overriding**.

- Note this is different than **method overloading** – two functions named identically with different signatures

OVERRIDING

```
1.  public class Shape {
2.     private Color c;

3.     /** other parts omitted
4.         for brevity */

5.     public void draw() {
6.        StdDraw.setPenColor(c);
7.     }
8.  }
```

```
1.  public class Circle extends Shape {
2.     private double x, y;
3.     private double radius;
4.
5.     /** other parts omitted
6.         for brevity */

7.     public void draw() {
8.        super.draw();
9.        StdDraw.filledCircle(
10.        x, y, radius);
11.    }
12. }
```

Circle overrides the implementation of draw

THE JAVA OBJECT CLASS

- Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

- Java Object provides for a few basic functions, like `toString()`.

- We will use others as we go.

POLYMORPHISM

POLYMORPHISM

- **Polymorphism** means that a variable of a superclass (supertype) can refer to a subclass (subtype) object

  ```
  Shape s = new Circle(5);
  ```

- Under the context of polymorphism, the supertype here is the **declared type** and the subtype is the **actual type**

- Polymorphism implies that an object of a subtype can be used wherever its supertype value is required

WHY WOULD YOU EVER DO THIS?

- Allow types to be defined at runtime, instead of at compile time:

```
1.  Scanner s = new Scanner(System.in);
2.  Shape shape = null;
3.  String tag = s.next();
4.  if(tag.equals("Circle")) {                      //user wants a circle
5.      double r = s.nextDouble();
6.      shape = new Circle(r, Color.red);
7.  }
8.  else if(tag.equals("Rectangle")) {              //User wants a rectangle
9.      double w = s.nextDouble(), h = s.nextDouble();
10.     shape = new Rectangle(w, h, Color.red);
11. }
12. System.out.println("Area: " + shape.area());  //works no matter what!
```

WHY WOULD YOU EVER DO THIS?

- Arrays can only store one type

1. **Circle**[] circles; //all circles
2. **Rectangle**[] rects; //all rectangles
3. **Shape**[] shapes; //depends on subtypes! Can have some circles and some rectangles.

WHY WOULD YOU EVER DO THIS?

- Lets say we have an array of Shape shapes then we can do something like:

```
1. double total = 0;
2. for(int i = 0; i < shapes.length; ++i)
3.     total += shapes[i].area(); //Uses specific
4.                                //instance's function
5. return total;
```

POLYMORPHISM DEMO

```
1.  public class PolymorphismDemo {
2.    public static void main(String[] args) {
3.      m(new Student());
4.      m(new Person());
5.      m(new Object());
6.    }
7.    public static void m(Object x) {
8.      System.out.println(x.toString());
9.    }
10. }
11.
12. class Student extends Person {
13.   public String toString() {
14.     return "Student";
15.   }
16. }
17.
18. class Person {
19.   public String toString() {
20.     return "Person";
21.   }
22. }
```

- Method m takes a parameter of the **Object** type. You can invoke it with any object.

- When the method m(**Object** x) is executed, the argument x's toString method is invoked. Classes **Student, Person,** and **Object** have their own implementation of the toString method.

- The correct implementation is dynamically determined by the Java Virtual Machine. This is called **dynamic binding**.

- Polymorphism allows superclass methods to be used generically for a wide range of object arguments (any possible subclass). This is known as **generic programming**.

POLYMORPHISM AND TYPE CONVERSION

- So when assigning a value of a subtype to a variable of a supertype, the conversion is implicit:

  `Shape s = new Circle(5); //implicit conversion from Circle to Shape`

  This is called **upcasting**.

- When going from a supertype value to a subtype variable, the conversion must be explicit:

  `Circle c = (Circle)s; //explicit conversion from Shape to circle`

  This is called **downcasting**. This type of casting might not always succeed, why?

THE INSTANCEOF OPERATOR

- Use the **instanceof** operator to test whether an object is an instance of a class:

```
1.Object myObject = new Circle();
2./** Perform downcasting only if myObject
3.    is an instance of Circle */
4.if (myObject instanceof Circle) {
5.  System.out.println("The circle diameter is " +
6.    ((Circle)myObject).getDiameter());
7.}
```

JAVA.LANG.OBJECT'S EQUALS METHOD

- The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
1. public boolean equals(
2.       Object obj) {
3.     return this == obj;
4. }
```

- What is the problem? How do we fix it?
  - == for objects compares their memory addresses, not their values.

- As an example of overriding the method for our Circle:

```
1. public boolean equals(
2.       Object o) {
3.     if (o instanceof Circle) {
4.        return radius ==
5.           ((Circle)o).radius;
6.     }
7.     else
8.        return false;
9. }
```

EXAMPLE

- Extend your account model to add two subtypes: one for a checking account and one for a savings account.

  - Add a method withdraw in your account super class, and override this functionality in checkings and savings.

  - Override the toString() method for the accounts

- Exemplify polymorphism in a main program that allows a user to create an account and make deposits/withdrawals from it

ADVANCED CONCEPTS OF INHERITANCE

THE PROTECTED VISIBILITY (SCOPE) MODIFIER

- The **protected** modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or *its subclasses*, even if the subclasses are in a different package.

Visibility Increases

$\longrightarrow$
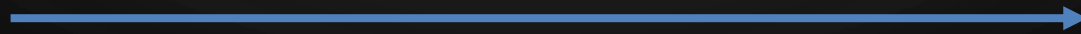
private, none (if no modifier is used), **protected**, **public**

ACCESSIBILITY SUMMARY

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

VISIBILITY MODIFIERS FULL EXAMPLE

```
package p1;

public class C1 {                 public class C2 {
   public int x;                     C1 o = new C1();
   protected int y;                  can access o.x;
   int z;                            can access o.y;
   private int u;                    can access o.z;
                                     cannot access o.u;
   protected void m() {
   }                                 can invoke o.m();
}                                 }


                                package p2;


public class C3               public class C4               public class C5 {
      extends C1 {                   extends C1 {               C1 o = new C1();
   can access x;                  can access x;                 can access o.x;
   can access y;                  can access y;                 cannot access o.y;
   can access z;                  cannot access z;              cannot access o.z;
   cannot access u;               cannot access u;              cannot access o.u;

   can invoke m();                can invoke m();               cannot invoke o.m();
}                             }                             }
```
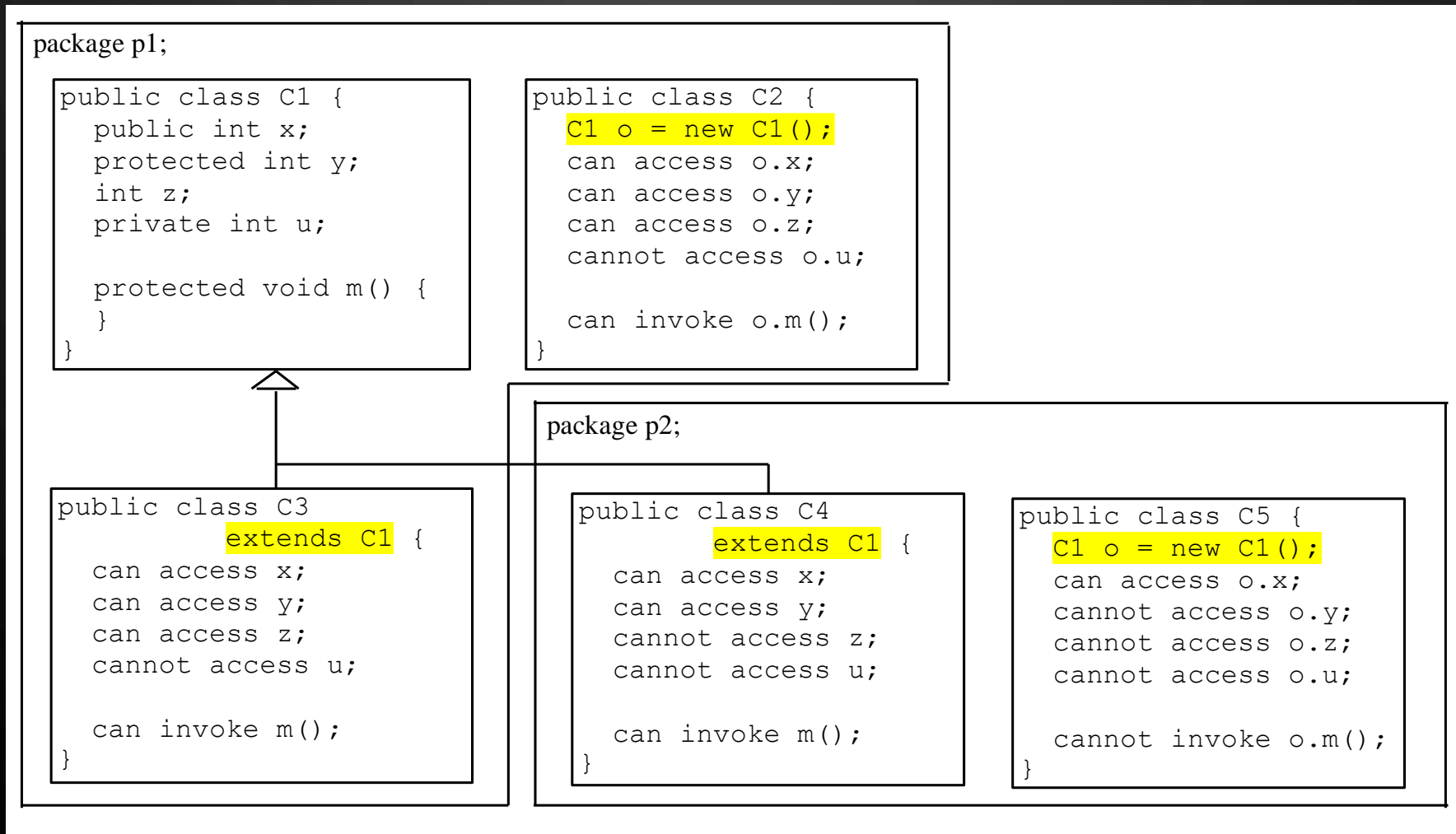
A SUBCLASS CANNOT WEAKEN THE ACCESSIBILITY

- A subclass may override a protected method in its superclass and change its visibility to public.

- However, a subclass cannot "weaken" the accessibility of a method defined in the superclass.

  - For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

THE FINAL MODIFIER

- The final modifier, introduced with variables to define constants, e.g., PI, has extended meaning in the context of inheritance:

- A final class **cannot** be extended:
```
final class Math {
    ...
}
```

- The final method cannot be overridden by its subclasses:
```
public final double getArea() {
    return Math.PI*radius*radius;
}
```

EXCEPTIONS

EXCEPTIONS

- **Exceptions** are unexpected events that occur during the execution of a program.

- An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.

- In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.

- An exception may also be **caught** by a surrounding block of code that "handles" the problem.

- If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.

CATCHING EXCEPTIONS

- The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.

- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch block** that contains the code to apply an appropriate resolution.

- If no exception occurs in the guarded code, all catch blocks are ignored.

```
1. …
2. try {
3.    /*Code that may
            generate exception*/
4. }
5. catch(ExceptionType1 e1) {
6. }
7. catch(ExceptionType2 e2) {
8. }
9. …
```

THROWING EXCEPTIONS

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.

- This is done by using the `throw` keyword followed by an instance of the exception type to be thrown:
  ```
  throw new ExceptionType(parameters);
  ```

THE THROWS CLAUSE

- When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.

- The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual throw statement).

```
public static int parseInt(String s)
    throws NumberFormatException;
```