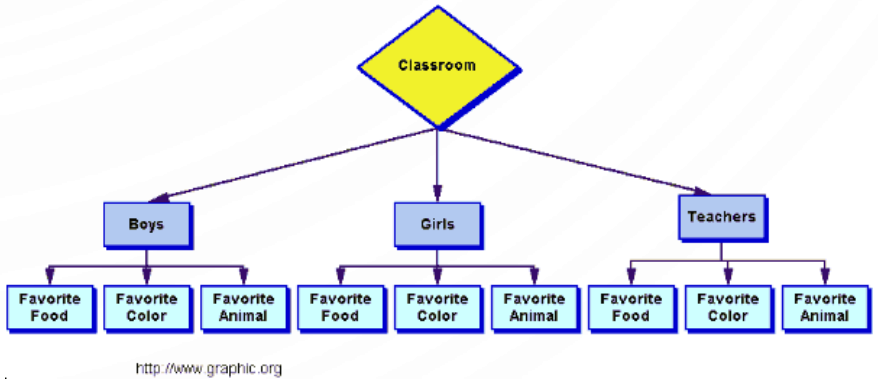
CHAPTER 11 INHERITANCE AND POLYMORPHISM

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO JAVA PROGRAMMING, LIANG (PEARSON 2014)

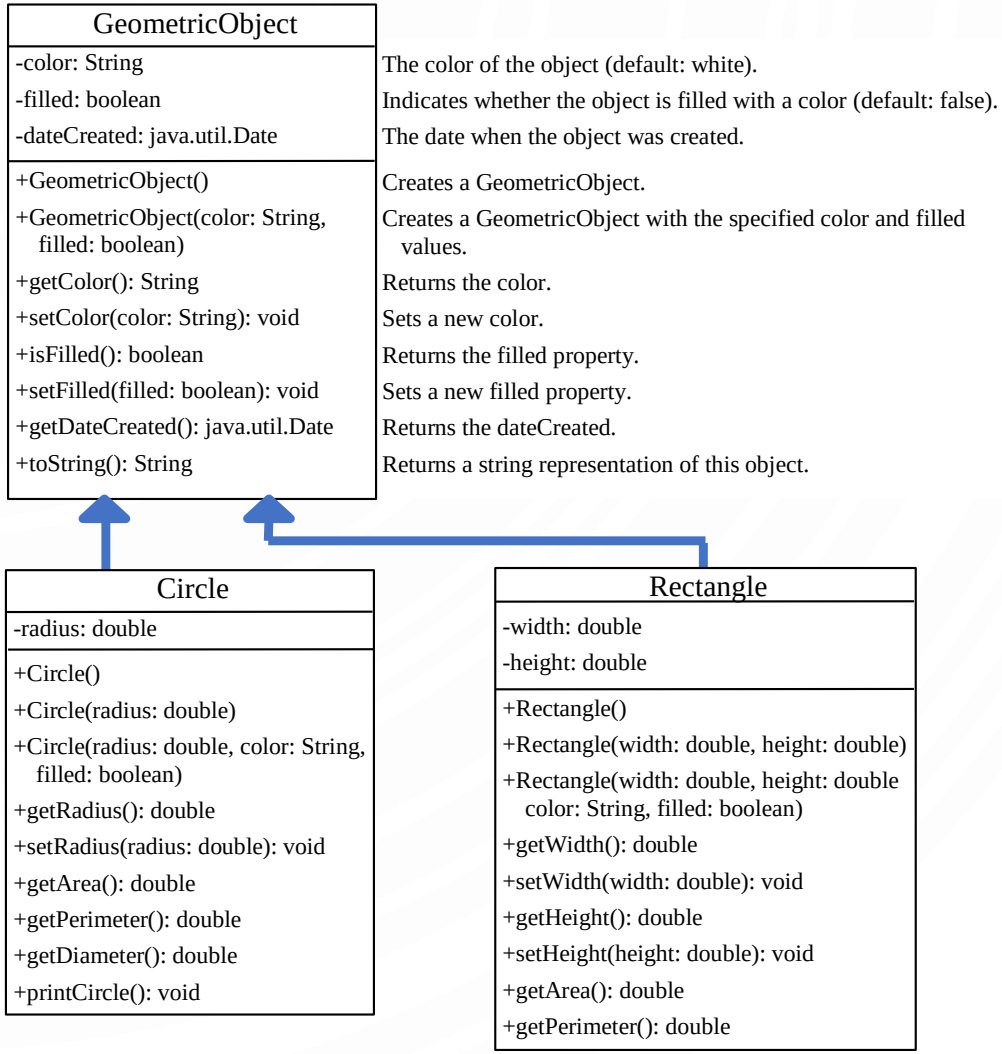
MOTIVATIONS

- 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



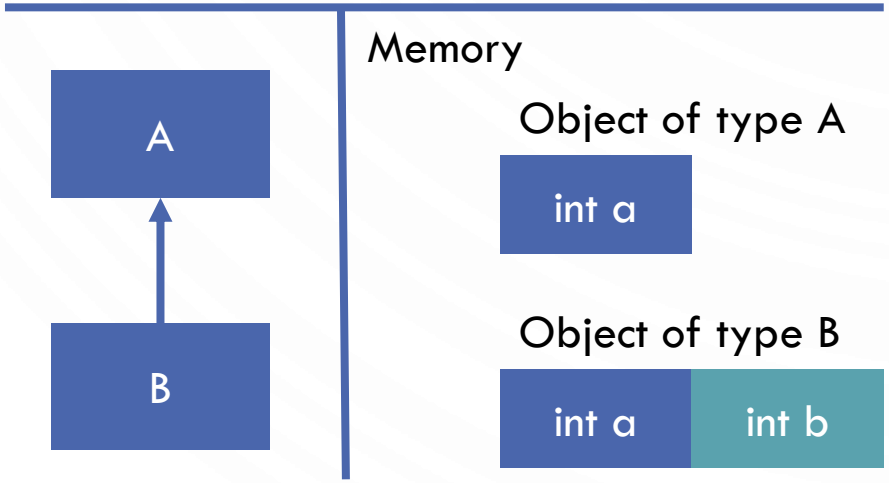
INHERITANCE

- **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



INHERITANCE IN JAVA

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



- 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
}
```

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Constructor chaining – invoking a constructor of a class will invoke all of the superclass's constructors

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Start by invoking Faculty constructor

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Invoke Employee constructor explicitly

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Invoke Employee constructor with `this`. If you use `this`, it must be the very first thing in the constructor. It supersedes even the call to `super`.

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Invoke Person constructor implicitly, before line 16

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Print (1) and return to Employee constructor

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Print (2) and return to Employee constructor

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Print (3) and return to Faculty constructor. Note that **super** is not called twice!

TRACE

```
1.  public class Faculty extends Employee {
2.    public static void main(String[] args) {
3.      new Faculty();
4.    }
5.    public Faculty() {
6.      super();
7.      System.out.println("(4) Faculty's no-arg constructor is invoked");
8.    }
9.  }
10. class Employee extends Person {
11.   public Employee() {
12.     this("(2) Invoke Employee's overloaded constructor");
13.     System.out.println("(3) Employee's no-arg constructor is invoked");
14.   }
15.   public Employee(String s) {
16.     System.out.println(s);
17.   }
18. }
19. class Person {
20.   public Person() {
21.     System.out.println("(1) Person's no-arg constructor is invoked");
22.   }
23. }
```

Print (4) and return to main.

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. }
```

FIND THE ERROR

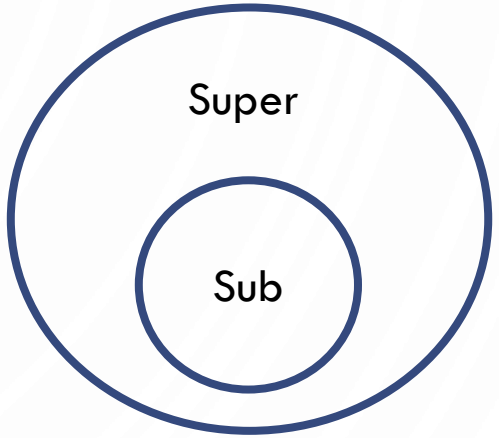
```
1.   public class Apple
2.       extends Fruit {
3.   }
4.
5.   class Fruit {
6.     public Fruit(String name) {
7.       System.out.println(
8.         "Fruit's constructor
9.           is invoked");
10.    }
11.  }
```

- If a superclass does not have a no-arg constructor, you must explicitly call an appropriate one in the subclass constructor with super.
- To fix, change Apple to the following:

```
1.   public class Apple
2.       extends Fruit {
3.     public Apple() {
4.       super("apple");
5.     }
6.   }
```

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

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

OVERRIDING VS. OVERLOADING

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

OVERRIDING

- Note – An instance method can be overridden only if it is accessible. Thus a `private` method **cannot** be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
- Note – Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is **hidden**.

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.

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```

OBJECT'S TOSTRING() METHOD

- The `toString()` method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
1. Loan loan = new Loan();
2. System.out.println(
3.   loan.toString());
```

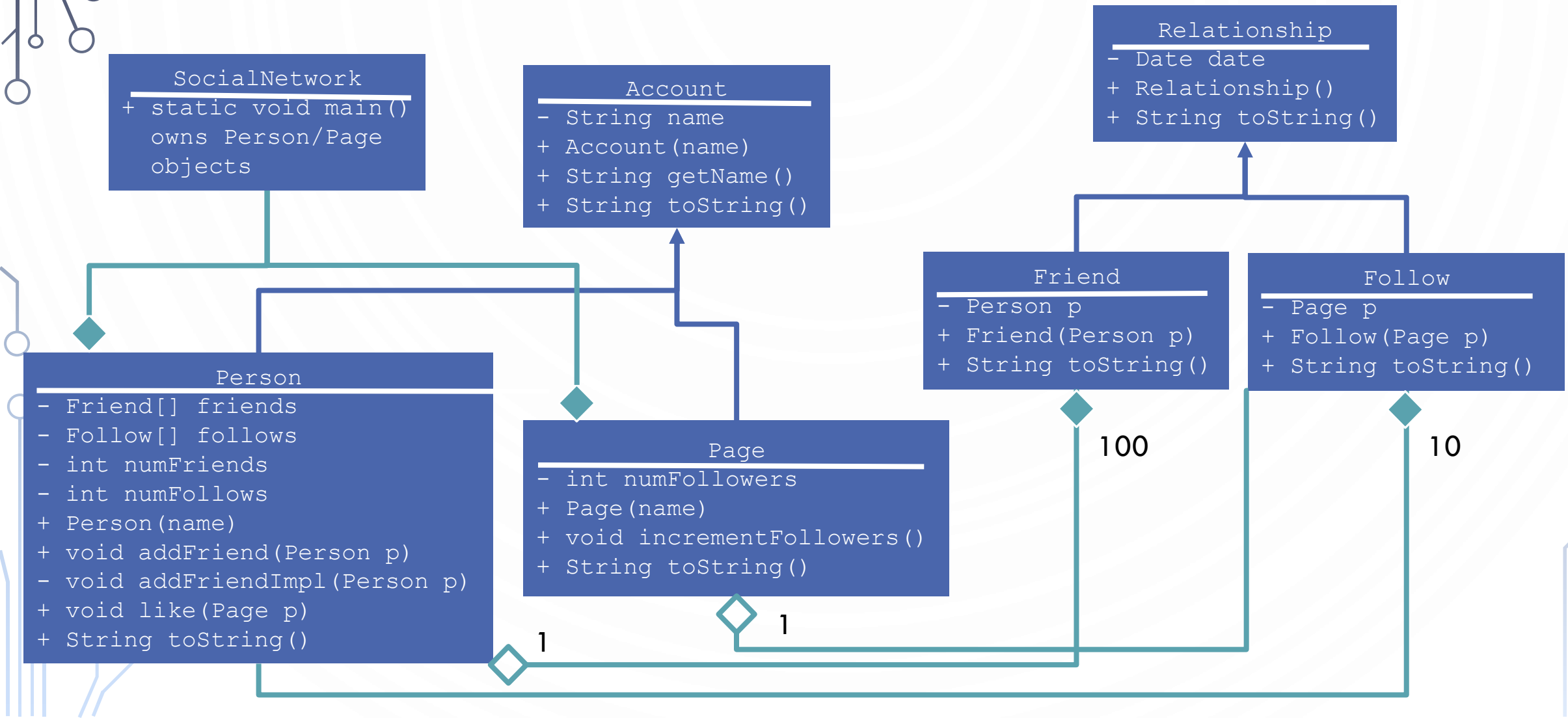
- The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a digestible string representation of the object.

EXTENDED EXAMPLE – A SOCIAL NETWORK

SOCIAL NETWORK

- We can make a simple social network to support multiple types of accounts and different types of relationships.
  - Accounts
    - Page
    - Person
  - Relationship
    - Friend
    - Follow
  - Social Network

UML DIAGRAM FOR ACCOUNTS



SOCIAL NETWORK

```
1.  public class SocialNetwork {
2.    public static void main(String args[]) {
3.      Person p1 = new Person("Jory Denny");
4.      Person p2 = new Person("Zachary Pollack");
5.      Page p3 = new Page("Zen-Do-Kai Karate");
6.      p2.addFriend(p1);
7.      p1.like(p3);
8.      p2.like(p3);
9.      System.out.println(p1 + "\n");
10.     System.out.println(p2 + "\n");
11.     System.out.println(p3);
12.   }
13. }
```

RELATIONSHIP

```
1.  import java.util.Date;
2.  public class Relationship {
3.     private Date date = new Date();
4.     public Relationship() {}
5.     public String toString() {
6.        return "Relationship created " + date;
7.        //Auto converts date to string through
8.        //Object's toString()
9.     }
10. }
```

FRIEND

```
1. public class Friend extends Relationship {
2.    private Person friend;
3.    public Friend(Person p) {friend = p}
4.    public String toString() {
5.       return "Friend. " + super.toString() +
6.          ". With " + friend.getName() + ".";
7.    }
8. }
```

FOLLOW

```
1.  public class Follow extends Relationship {
2.    private Page page;
3.    public Follow(Page p) {
4.      page = p; page.incrementFollows();
5.    }
6.    public String toString() {
7.      return "Follow. " + super.toString() +
8.        ". Following " + page.getName() + ".";
9.    }
10. }
```

ACCOUNT

```
1. public class Account {
2.    private String name;
3.    public Account(String n) {name = n;}
4.    public String getName() {return name;}
5.    public String toString() {
6.       return "Account name: " + name;
7.    }
8. }
```

PAGE

```
1.  public class Page extends Account {
2.    private int numFollowers = 0;
3.    public Page(String name) {
4.      super(name);
5.    }
6.    public void incrementFollows() {++numFollowes;}
7.    public String toString() {
8.      return "Page\n\t" + super.toString() +
9.        "\n\tNumber of followers: " + numFollowers;
10.   }
11. }
```

PERSON – 1

```
1.  public class Person extends Account {
2.    private Friend[] friends = new Friend[100];
3.    private Follow[] follows = new Follow[10];
4.    private int numFriends = 0;
5.    private int numFollows = 0;

6.    public Person(String name) {
7.      super(name);
8.    }

9.    public void addFriend(Person p) {
10.     if(numFriends < 100 &&
11.         p.numFriends < 100) {
12.       addFriendImpl(p);
13.       p.addFriendImpl(this);
14.     }
15.     else
16.       System.out.println("Cannot add friend: "
17.         + p.getName() + " to " + getName();
18.   }
19.   private void addFriendImpl(Person p) {
20.     for(int i = 0; i < 100; ++i) {
21.       if(friends[i] == null) {
22.         friends[i] = new Friend(p);
23.         ++numFriends;
24.         break;
25.       }
26.     }
27.   }
28.
29.   public void like(Page p) {
30.     if(numFollows < 10) {
31.       for(int i = 0; i < 10; ++i) {
32.         if(follows[i] == null) {
33.           follows[i] = new Follow(p);
34.           ++numFollows;
35.           break;
36.         }
37.       }
38.     }
39.   }
```

PERSON – 2

```
40.     public String toString() {
41.       String s = "Person\n\t" + super.toString() +
42.         "\n\tNumber of friends: " + numFriends +
43.         "\n\tNumber of follows: " + numFollows;
44.       s += "\n\tFriends";
45.       for(Friend f : friends)
46.         if(f != null)
47.           s += "\n\t\t" + f; //toString is automatically called
48.       s += "Follows";
49.       for(Follow f : follows)
50.         if(f != null)
51.           s += "\n\t\t" + f; //toString is automatically called
52.       return s;
53.     }
54.   }
```

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**.

METHOD MATCHING VS. BINDING

- Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

TYPE CONVERSION REVISITED

- From dealing with data we know types can be converted to each other:

```
int a = 5;
double x = a;
int b = (int)(x*x);
```

- We can do the same with our own objects!

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.}
```

CASTING ANALOGY

- To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

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. }
```

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.

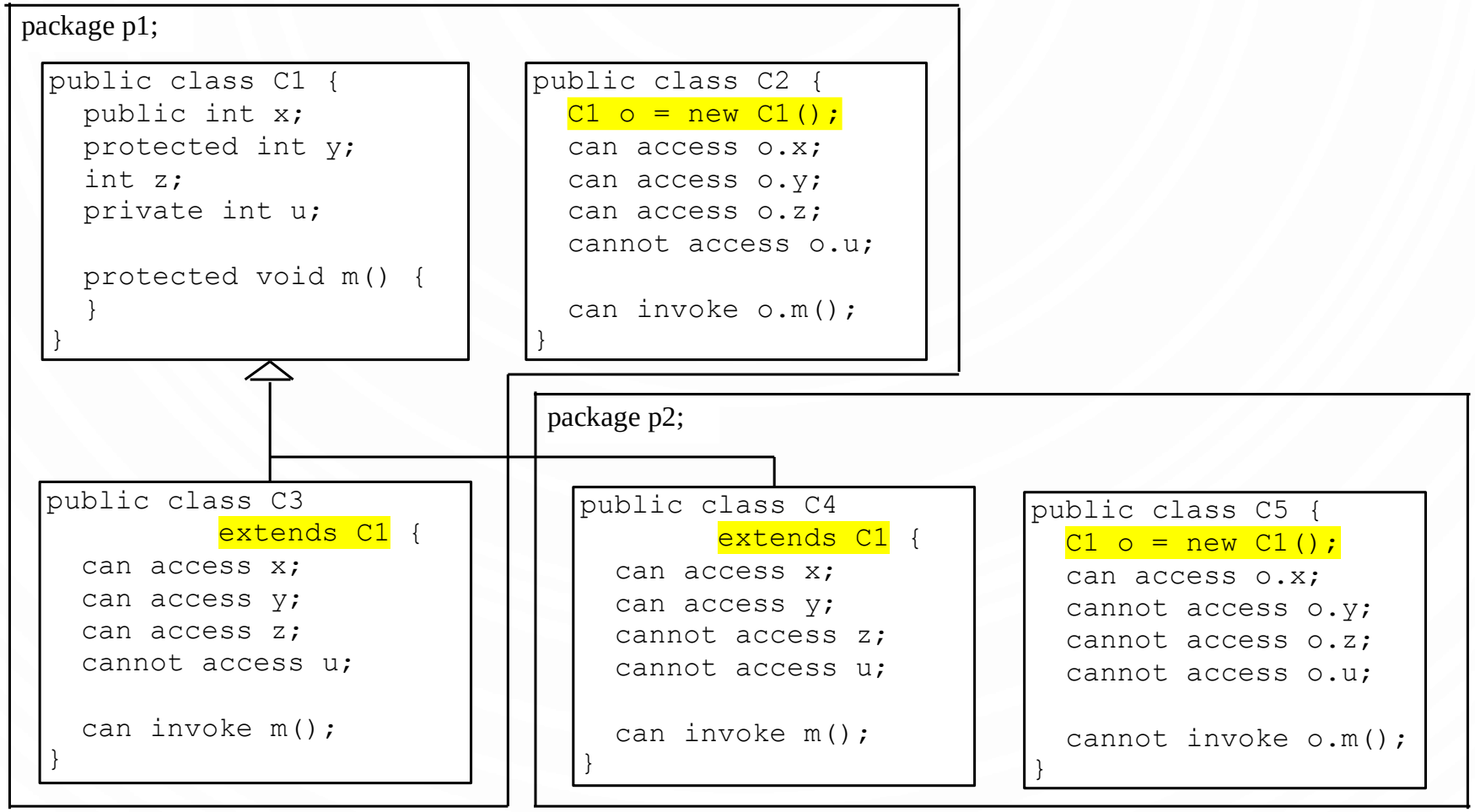
Visibility Increases

`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



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;
}
```

CH. 13
ABSTRACT CLASSES AND INTERFACES

A 4 SLIDE OVERVIEW. ONLY HIGH LEVEL CONCEPTS AND CONSTRAINTS WOULD BE TESTED.

ABSTRACT CLASSES

- In modeling, sometimes we don't want to allow types to be defined:
  `Shape s = new Shape(Color.red); //Makes no sense. What is s really?`
- We can use abstract classes to facilitate this to provide better protection to other software developers on our team. Also specified interface (API) requirements of subtypes.

```
1. public abstract class Shape {   //Abstract here disbars the code above.
2.                                 //No "new" is allowed on this type.
3.     protected Shape(Color c) {…} //Constructor is protected because
4.                                 //nothing but subtypes will access it
5.     …
6.     public abstract double area(); //If a function is abstract no
7.                                    //definition needs to be provided
8.     public abstract double perimeter(); //Also subtypes are now required
9.                                         //to define them!
10.}
```

SOME INTERESTING POINTS ON ABSTRACT

- An abstract method cannot be contained in a non abstract class
- If a subclass of an abstract superclass does not implement all of the abstract methods, then it must also be declared as abstract
- Cannot use new on an abstract type, but constructors can be defined (for use with super). Also can still use the abstract type for polymorphism!
- An abstract class does not require abstract methods
- A subclass can be abstract even if the superclass is concrete (non abstract)

INTERFACES

- An interface is a way to define only the behavior of a class. They contain only constants and abstract methods (almost like a purely abstract class).

```
1. public interface AreaComputation {  //Note "interface"
2.                                     //not "class"
3.     public static final double PI = Math.PI;
4.     public abstract area();
5. }
```

INTERFACES

- Cannot have constructors
- All variables must be `public static final`
- All methods must be `public abstract`
- The last two points imply you don't need to specify any modifiers at all
- Useful for writing algorithms for searching or sorting (these need comparison), i.e., Comparable things (any object “implementing” the Comparable interface)
- Used to support multiple inheritance

INTERFACES

- To inherit an interface:

```
public class Shape implements
        AreaComputation, PerimeterComputation {
...}
```

- Implementing an interface requires implementation of all of the abstract methods, or declaring as an abstract class.
- Interfaces commonly used as a weaker is-a relationship, specifically is-kind-of referring to possessing certain properties only
- Oddly, interfaces can “extend” other interfaces