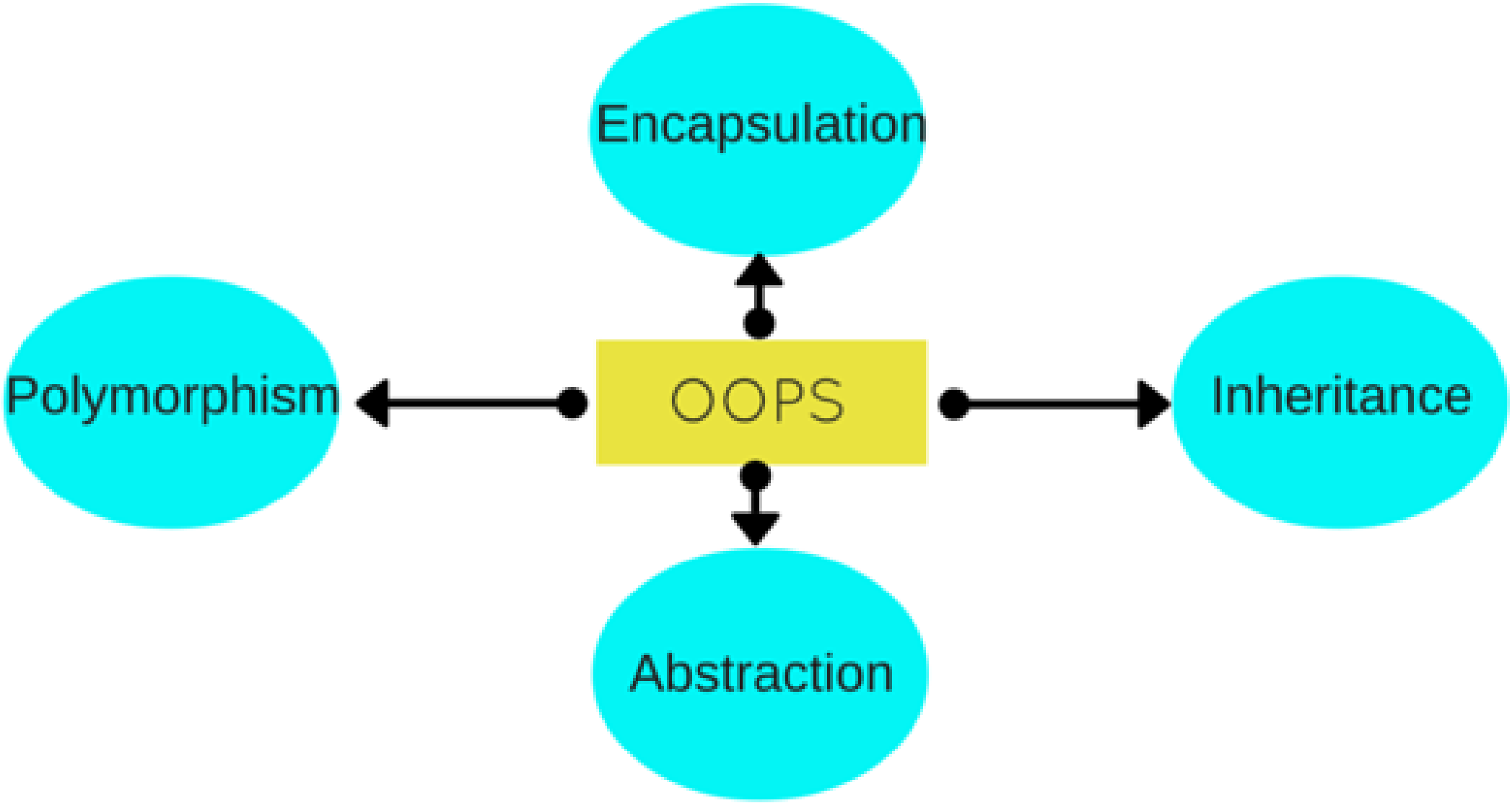
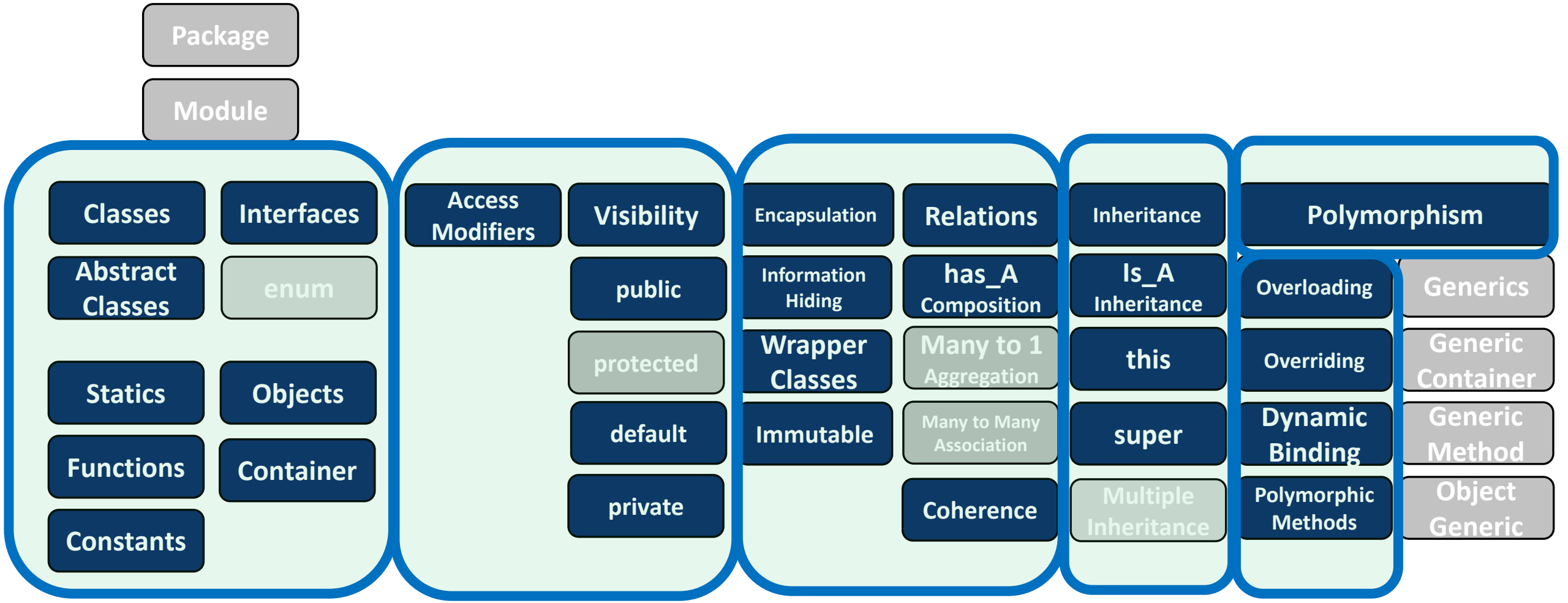
CS 24 AP Computer Science A Review

Week 3: Inheritance and Polymorphism

DR. ERIC CHOU
IEEE SENIOR MEMBER


Encapsulation
Polymorphism
OOPS
Inheritance
Abstraction

Object-Oriented Programming

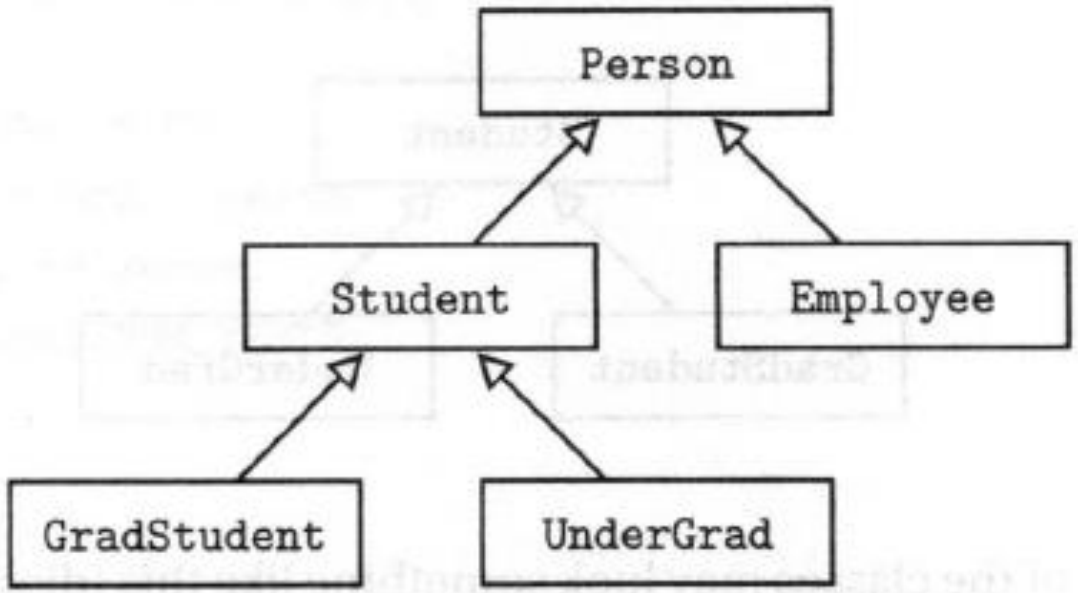


SECTION 1

Inheritance

Super Class and Sub-class

- Inheritance defines a relationship between objects that share characteristics. Specifically, it is the mechanism whereby a new class, called a **subclass**, is created from an existing class, called a **superclass**, by absorbing its state and behavior and augmenting these with features unique to the new class. We say that the subclass inherits characteristics of its superclass.
- Don't get confused by the names: a subclass is bigger than a superclass-it contains more data and more methods!
- **Inheritance provides an effective mechanism for code reuse.** Suppose the code for a super class has been tested and debugged. Since a subclass object shares features of a superclass object, the only new code required is for the additional characteristics of the subclass.



Inheritance Hierarchy

- A subclass can itself be a superclass for another subclass, leading to an inheritance hierarchy of classes.
- For example, consider the relationship between these classes: **Person**, **Employee**, **Student**, **GradStudent**, and **UnderGrad**.

Is_A Relationship

- For any of these classes, an arrow points to its superclass. The arrow designates an inheritance relationship between classes, or, informally, an is-a relationship. Thus, an **Employee is-a Person**; a Student **is-a** Person; a GradStudent **is-a** Student; an UnderGrad **is-a** Student. Notice that the opposite is not necessarily true: A Person may not be a Student, nor is a Student necessarily an UnderGrad.
- Note that the is-a relationship is **transitive**: If a GradStudent is-a Student and a Student is-a Person, then a GradStudent is-a Person.

Subclasses

- Every subclass inherits the public or protected variables and methods of its superclass.
- Subclasses may have additional methods and instance variables that are not in the superclass.
- A subclass may redefine a method it inherits. For example, GradStudent and UnderGrad may use different algorithms for computing the course grade, and need to change a computeGrade method inherited from Student.
- This is called method **overriding**. If part of the original method implementation from the superclass is retained, we refer to the rewrite as **partial overriding**.

Inheritance

Inheritance in Java begins with the relationship between two classes defined like this:

class SubClass extends SuperClass

Inheritance expresses the is a relationship in that SubClass is a (**specialization** of) SuperClass. The extends relation has many of the same characteristics of the implements relationship used for interfaces

Interface MyImplementationClass implements MyInterface

As with inheritance, we say that **MyImplementationClass** is a **MyInterface**.

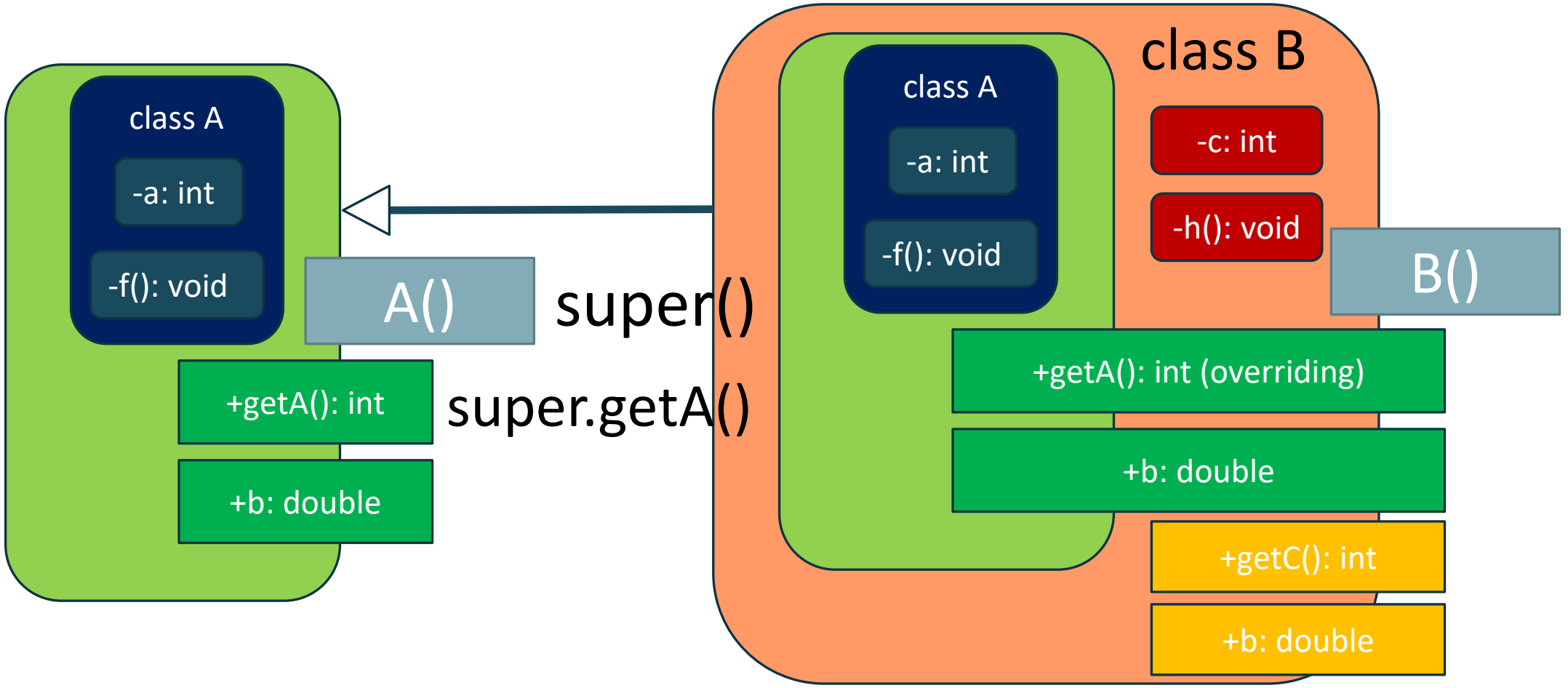
The Keyword super

When you make an object from a child class, the child class constructor automatically calls the no-arg constructor of the parent class using the super() call (some integrated development environments, or IDEs, display this instruction).

However, if you want to call a parent constructor with arguments from the child class constructor, you need to put that in your code explicitly as the first line of code in the child class constructor using the super(arguments) instruction.

The child is then making a call to a parameterized constructor of the parent class.

super keyword

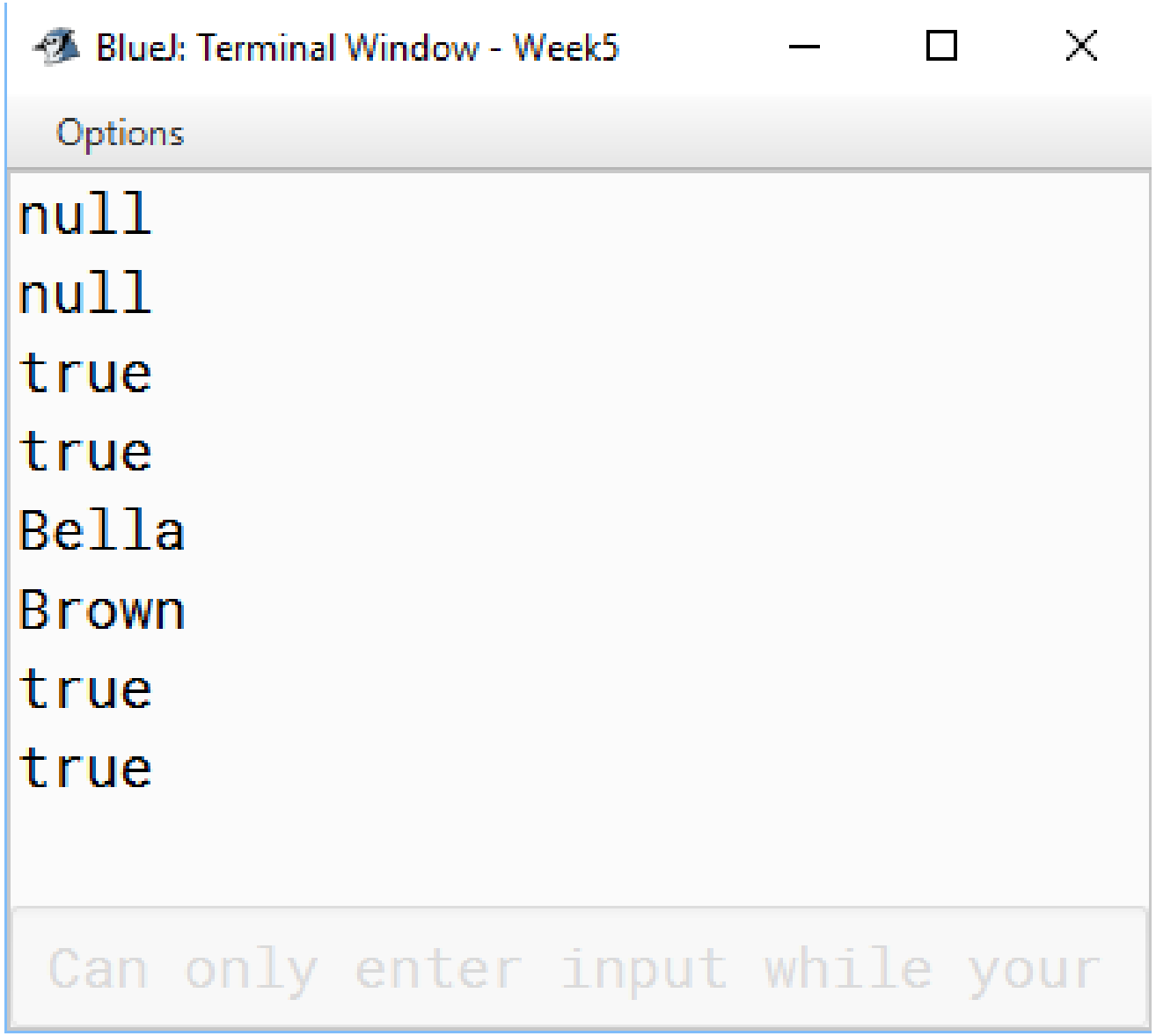


super() Keyword

```java
public class Mammal{
   boolean vertebrate;
   boolean milkProducer;
   String hairColor;

   public Mammal(){
       vertebrate = true;
       milkProducer = true;
     }
   public Mammal(String color){
       vertebrate = true;
       milkProducer = true;
       hairColor = color;
     }
   public boolean isVertebrate()  { return vertebrate; }
   public boolean isMilkProducer(){ return milkProducer; }
   public String  getHairColor()  { return hairColor; }
}
```

```java
public class Dog extends Mammal{
    String name;
    public Dog(){
        super();
     }
    public Dog(String hairColor, String nameOfDog){
        super(hairColor);
        name = nameOfDog;
     }
    public String getName(){ return name; }
    public static void main(String[] args){
        Dog myDog1 = new Dog();
        System.out.println(myDog1.getName());
        System.out.println(myDog1.getHairColor());
        System.out.println(myDog1.isVertebrate());
        System.out.println(myDog1.isMilkProducer());
        Dog myDog2 = new Dog("Brown", "Bella");
        System.out.println(myDog2.getName());
        System.out.println(myDog2.getHairColor());
        System.out.println(myDog2.isVertebrate());
        System.out.println(myDog2.isMilkProducer());
     }
}
```



super() Keyword

Casting

- Casting a reference upward is OK.

  Object obj = new Friend();

  obj will lose data field/methods. And, that is fine.

- Casting a reference downward is not OK.

  Friend fr = new Object();

  fr don’t get the data field/methods that it requires. So, it is not OK.

Inheriting Instance Methods and Variables

- A **subclass** inherits all the public and protected methods of its superclass.
- It does not, however, inherit the private instance variables or private methods of its parent class, and therefore does not have direct access to them. To access private instance variables, a subclass must use the accessor or mutator methods that it has inherited.
- In the Student example, the **UnderGrad** and **GradStudent** subclasses inherit all of the methods of the Student superclass. Notice, however, that the Student instance variables name, tests, and grade are private and are therefore not inherited or directly accessible to the methods in the **UnderGrad** and **GradStudent** subclasses. A subclass can, however, directly invoke the public accessor and mutator methods of the superclass. Thus, both **UnderGrad** and **GradStudent** use getTestAverage. Additionally, both **UnderGrad** and **GradStudent** use setGrade to access indirectly-and modify-grade.

Inheriting Instance Methods and Variables

- If, instead of pri vate, the access specifier for the instance variables in Student were public or protected, then the subclasses could directly access these variables. The keyword protected is not part of the AP Java subset.
- Classes on the same level in a hierarchy diagram do not inherit anything from each other, for example, UnderGrad and GradStudent). All they have in common is the identical code they inherit from their superclass.

Method Overriding and the super Keyword

- Any public method in a superclass can be overridden in a subclass by defining a method with the same return type and signature (name and parameter types). For example, the compute Grade method in the UnderGrad subclass overrides the computeGrade method in the Student superclass.
- Sometimes the code for overriding a method includes a call to the superclass method. This is called partial overriding. Typically, this occurs when the subclass method wants to do what the superclass does, plus something extra. This is achieved by using the keyword super in the implementation. The computeGrade method in the GradStudent subclass partially overrides the matching method in the Student class.

Method Overriding and the super Keyword

- The statement

```
super.computeGrade();
```

signals that the computeGrade method in the superclass should be invoked here. The additional test

```
if (getTestAverage() >= 90)
    …
```

allows a GradStudent to have a grade Pass with distinction. Note that this option is open to GradStudents only.

Note: Private methods cannot be overridden.

Constructors and super

- Constructors are never inherited! If no constructor is written for a subclass, a default constructor with no parameters is generated, which calls the no-argument constructor from the superclass. If the superclass does not have a no-argument constructor, but only a constructor with parameters, a compiler error will occur. If there is a no-argument constructor in the superclass, inherited data members will be initialized as for the superclass. Additional instance variables in the subclass will get a default initialization-o for primitive types and null for reference types.
- A default constructor is a no-argument constructor generated by the compiler when a subclass doesn't have an explicit constructor.

Constructors and super

- A subclass constructor can be implemented with a call to the super method, which invokes the superclass constructor. For example, the no-argument constructor in the UnderGrad class is identical to that of the Student class. This is implemented with the statement

```
super();
```

- The second constructor in the UnderGrad class is called with parameters that match those in the constructor of the Student superclass.

```
public UnderGrad(String studName, int[] studTests,
String studGrade)
{ super (studName, studTests, studGrade); }
```

Constructors and super

- For each constructor, the call to super has the effect of initializing the instance variable name, tests, and grade exactly as they are initialized in the Student class.
- Contrast this with the constructors in GradStudent. In each case, the instance variables name, tests, and grade are initialized as for the Student class. Then the new instance variable, gradID, must be explicitly initialized.

Constructors and super

```
public GradStudent(){
   super 0 ;
   gradID = 0;
}

public GradStudent(String studName, int[] studTests,
String studGrade, int gradStudID){
 super (studName, studTests, studGrade);
 gradID = gradStudID;
}
```

Constructors and super

Note

1. If super is used in the implementation of a subclass constructor, it must be used in the first line of the constructor body.

2. If no constructor is provided in a subclass, the compiler provides the following default constructor:

```
public SubClass() {
    super();  //calls no-argument constructor of superclass
  }
```

3. If the superclass has at least one constructor with parameters, the code in

Note 2 above will cause a compile-time error if the superclass does not also contain a no-argument constructor.

Rules for Subclasses

- A subclass can add new private instance variables.
- A subclass can add new public, private, or static methods.
- A subclass can override inherited methods.
- A subclass may not redefine a public method as private.
- A subclass may not override static methods of the superclass.
- A subclass should define its own constructors.
- A subclass cannot directly access the private members of its superclass. It must use accessor or mutator methods.

SECTION 2

Subclasses

Declaring Subclass Objects

- When a superclass object is declared in a client program, that reference can refer not only to an object of the superclass, but also to objects of any of its subclasses. Thus, each of the following is legal:

```
Student s = new Student();
Student g = new GradStudent();
Student u = new UnderGrade);
```

- This works because a GradStudent is-a Student, and an UnderGrad is-a Student.
- Note that since a Student is not necessarily a GradStudent nor an UnderGrad, the following declarations are not valid:

```
GradStudent g = new Student();
UnderGrad u = new Student();
```

Declaring Subclass Objects

Consider these valid declarations:

```
Student s = new Student("Brian Lorenzen", new int[]
{90,94,99}, “none");
Student u = new UnderGrad("Tim Broder", new int[]
{90,90,100}, “none") ;
Student g = new GradStudent("Kevin Cristella", new
int[] {85,70,90}, "none", 1234);
```

suppose you make the method call

```
s.setGrade("Pass");
```

The appropriate method in Student is found and the new grade assigned.

Declaring Subclass Objects

- The method calls

```
g.setGrade("Pass");
```

and

```
u.setGrade("Pass");
```

achieve the same effect on g and u since GradStudent and UnderGrad both inherit the setGrade method from Student. The following method calls, however, won't work:

```
int studentNum = s.getID();
int underGradNum = u.getID();
```

- These don't compile because Student does not contain getID.

Declaring Subclass Objects

- Now consider the following valid method calls:

```
s.computeGrade();
g.computeGrade();
u.computeGrade();
```

- Since s, g, and u have all been declared to be of type Student, will the appropriate method be executed in each case? That is the topic of the next section, polymorphism.

Note

The initializer list syntax used in constructing the array parameters-for example, new int [] {90, 90, 100}- will not be tested on the AP exam.

SECTION 3

Polymorphism

Polymorphism

- A method that has been overridden in at least one subclass is said to be polymorphic. An example is **computeGrade**, which is redefined for both **GradStudent** and **UnderGrad**.
- Polymorphism is the mechanism of selecting the appropriate method for a particular object in a class hierarchy. The correct method is chosen because, in Java, method calls are always determined by the type of the actual object, not the type of the object reference.
- For example, even though s, g, and u are all declared as type Student, s.computeGrade(), g.computeGrade(), and u. computeGrade() will all perform the correct operations for their particular instances. In Java, the selection of the correct method occurs during the run of the program.

Here is a simple example.

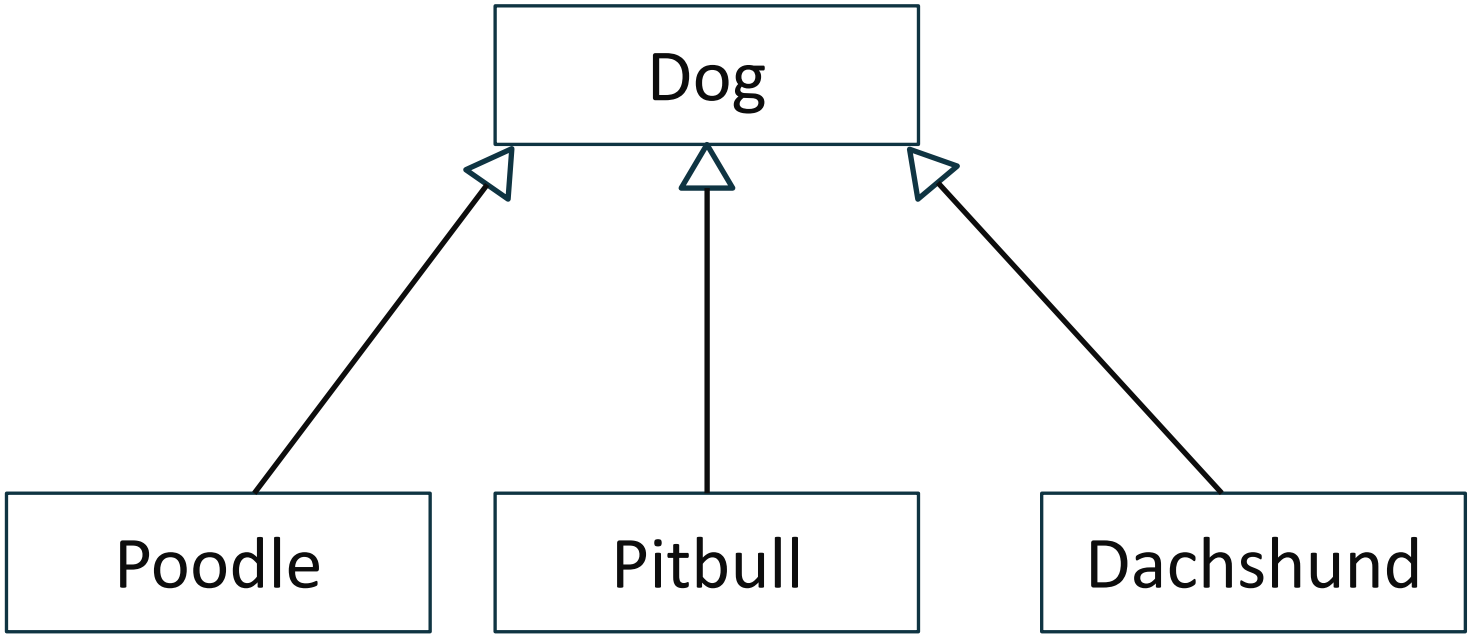
```
public class Dog {
   private String name;
   private String breed;

   public Dog (String aName, String aBreed) {
       name = aName;
       breed = aBreed;
   }
   …
}
```

```
public class Poodle extends Dog {
   private boolean needsGrooming;

   public Poodle (String aName, String aBreed,
                  boolean grooming) {
      super (aName, aBreed);
      needsGrooming = grooming;
   }
   …
}
```

Now consider this hierarchy of classes, and the declarations that follow it:



Polymorphism

- Suppose the Dog class has this method:

```
public void eat()
{ /* implementation not shown */ }
```

- And each of the subclasses, Poodle, PitBull, Dachshund, etc., has a different, overridden eat method. Now suppose that allDogs is an ArrayList<Dog> where each Dog declared above has been added to the list. Each Dog in the list will be processed to eat by the following lines of code:

```
for (Dog d: allDogs)
        d.eat();
```

- Polymorphism is the process of selecting the correct eat method, during run time, for each of the different dogs.

Polymorphism

- Recall that constructors are not inherited, and if you use the keyword super in writing a constructor for your subclass, the line containing it should precede any other code in the constructor.
- Recall also that if the superclass does not have a no-argument constructor, then you must write an explicit constructor for the subclass. If you don't do this, later code that tries to create a subclass object will get a compile-time error. This is because the compiler will generate a default constructor for the subclass that tries to invoke the no-argument constructor of the superclass.

Polymorphism

Overriding a Method of the Parent Class

In Java, a child class is allowed to override a method of a parent class. This means that even though the child inherited a certain way to do something from its parent, it can do it in a different way if it wants to. This is an example of polymorphism.

Static Binding: Binding at compile time by method signature matching.

Dynamic Binding: Binding at run-time by dynamic binding chain.

Static Binding

- Making a run - time decision about which instance method to call is known as dynamic binding or late binding. Contrast this with selecting the correct method when methods are overloaded (see p. 106) rather than overridden. The compiler selects the correct overloaded method at compile time by comparing the methods' signatures. This is known as static binding, or early binding.

Dynamic Binding

- In polymorphism, the actual method that will be called is not determined by the compiler. Think of it this way: The compiler determines if a method can be called (i.e., does the method exist in the class of the reference?), while the run-time environment determines how it will be called (i.e., which overridden form should be used?).

Example 1

```
Student s = null;
Student u = new UnderGrad("Tim Broder", new int[] {90,90,100}, “none");
Student g = new GradStudent("Kevin Cristella", new int [] {85, 70, 90},
"none", 1234);
System.out.print("Enter student status: “);
System.out.println("Grad (G), Undergrad (U), Neither (N)");

String str = ... ; //read user input
if (str. equals ("G"))
    s = g;
else if (str.equals("U"))
    s = u;
else
    s = new Student();
s.computeGrade();
```

Example 2

```
public class StudentTest{
    public static void computeAllGrades(Student[] studentList){
        for (Student s : studentList)
            if (s ! = null)
                s.computeGrade();
        }

    public static void main(String[] args){
        Student[] stu = new Student[5];
        stu[0] = new Student("Brian Lorenzen", new int[] {90,94,99}, "none");
        stu[1] = new UnderGrad("Tim Broder", new int[] {90,90,100}, "none");
        stu[2] = new GradStudent("Kevin Cristella", new int[] {85,70,90},
                                 "none",1234);
        computeAllGrades(stu);
    }
}
```

null Reference

- Here an array of five Student references is created, all initially null. Three of these references, stu[0], stu[1], and stu[2], are then assigned to actual objects. The computeAllGrades method steps through the array invoking for each of the objects the appropriate computeGrade method, using dynamic binding in each case. The null test in computeAllGrades is necessary because some of the array references could be **null**.

Polymorphism

Dynamic Binding

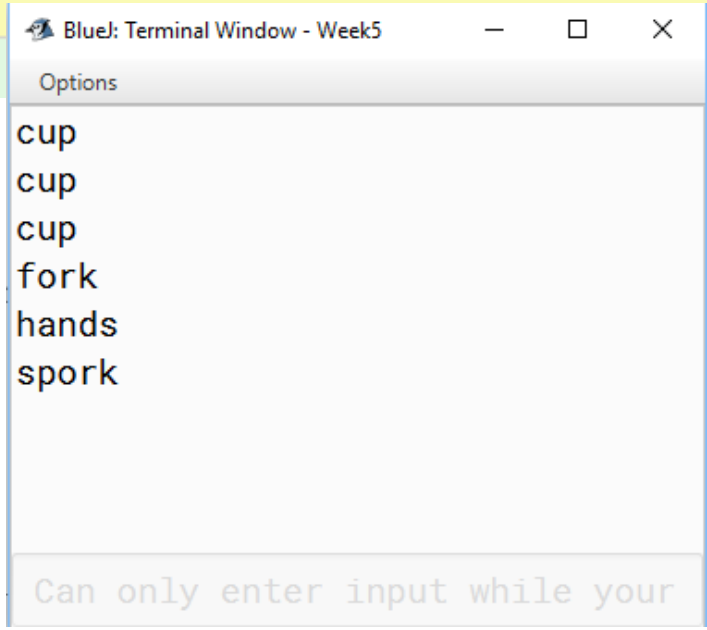
- In Java, a child class is allowed to override a method of a parent class. This means that even though the child inherited a certain way to do something from its parent, it can do it in a different way if it wants to.
- This is an example of **polymorphism**.

Static Binding Example

```java
public class FamilyPerson{
    public String drink(){ return "cup"; }
    public String eat()  { return "fork";}
    public static void main(String[] args){
      FamilyPerson mom = new FamilyPerson();
      Baby junior = new Baby();
      SporkUser auntSue = new SporkUser();
      System.out.print("\f");
      System.out.println(mom.drink());
      System.out.println(junior.drink());
      System.out.println(auntSue.drink());
      System.out.println(mom.eat());
      System.out.println(junior.eat());
      System.out.println(auntSue.eat());
    }
}
```

```java
public class Baby extends FamilyPerson{
    public String eat() { return "hands"; }
}
```

```java
public class SporkUser extends FamilyPerson{
    public String eat(){ return "spork"; }
    public static void main(String[] args){
     }
}
```



Dynamic Binding Example

```java
import java.util.List;
import java.util.ArrayList;
public class DynamicFamilyPerson{
    public static void main(String[] args){
        FamilyPerson mom = new FamilyPerson();
        FamilyPerson junior = new Baby();
        FamilyPerson auntSue = new SporkUser();
        List<FamilyPerson> family = new ArrayList<FamilyPerson>();
        family.add(mom);
        family.add(junior);
        family.add(auntSue);
        for (FamilyPerson member: family){
            System.out.println(member.drink());
            System.out.println(member.eat());
            System.out.println();
        }
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
cup
fork

cup
hands

cup
spork
```

Can only enter input while your

Using super in a Subclass

- A subclass can call a method in its superclass by using super. Suppose that the superclass method then calls another method that has been overridden in the subclass. By polymorphism, the method that is executed is the one in the subclass. The computer keeps track and executes any pending statements in either method.

Example 3

```
public class Dancer{
    public void act(){
        System.out.print (" spin ");
        doTrick();
    }

    public void doTrick(){
        System.out.print (" float ");
    }
}

public voipublic class Acrobat extends Dancer{
    Public void act(){
        super.act();
        System.out.print (" flip “);
    }
    public void doTrick(){
        System.out.print (" somersault “);
    }
}
```

super() is an Implicit Class Reference

- Reference variable for super class.
- Used as super class constructor super(), super(a, b);
- Used for calling super class methods.

Example of super.eat() using super keyword for super class method

```
1  public class Baby2 extends FamilyPerson {
2     private int age;
3     public Baby2(int myAge){ age = myAge; }
4     public String eat(){
5        if (age > 3) return "hands or a " + super.eat();
6        else return "hands";
7     }
8     public static void main(String[] args){
9        Baby2 youngBaby = new Baby2(2);
10       Baby2 oldBaby   = new Baby2(4);
11       System.out.println(youngBaby.eat());
12       System.out.println(oldBaby.eat());
13    }
14 }
```

BlueJ: Terminal Window - Week5

Options

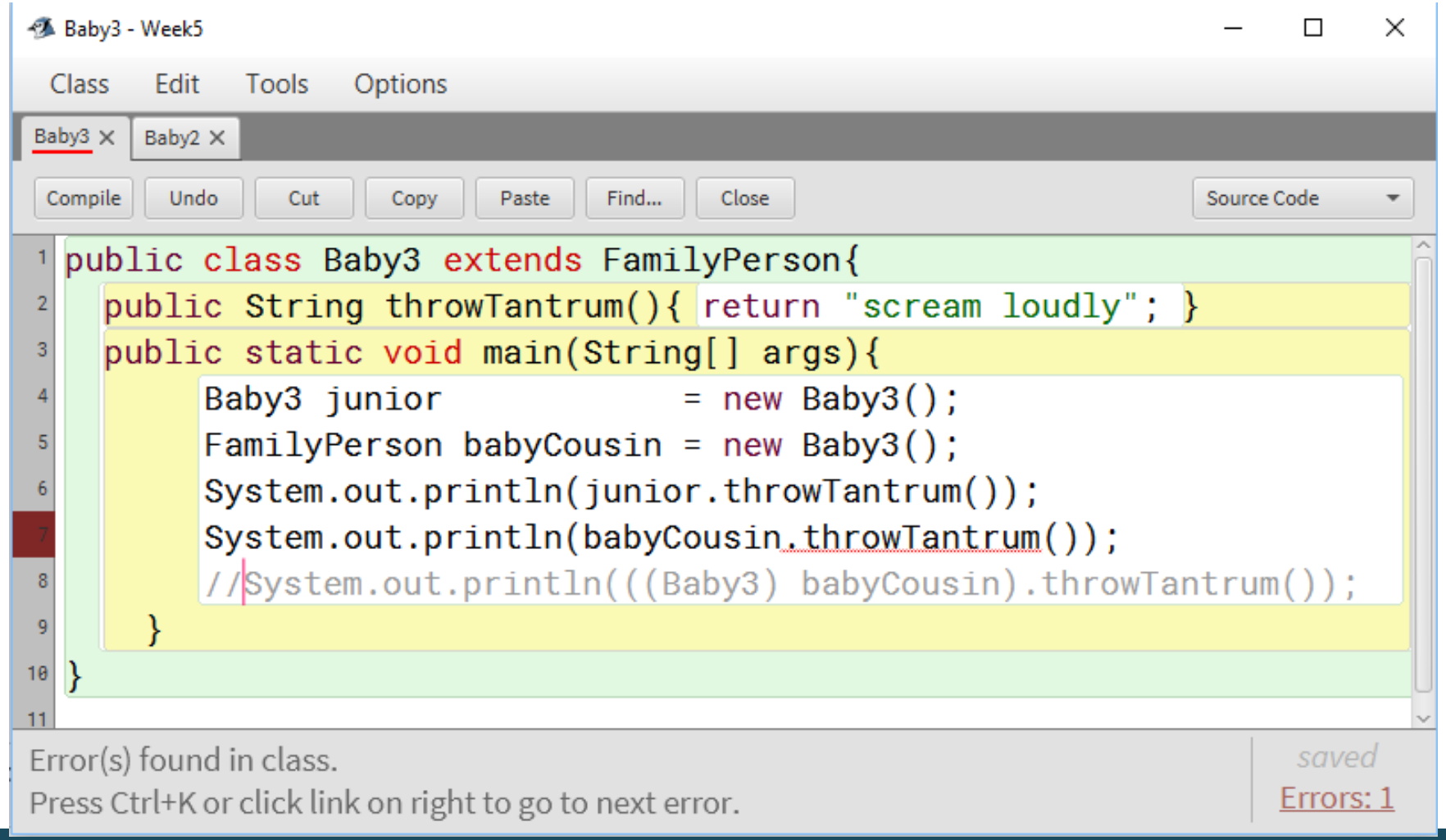
```
hands
hands or a fork
```

Can only enter input while your

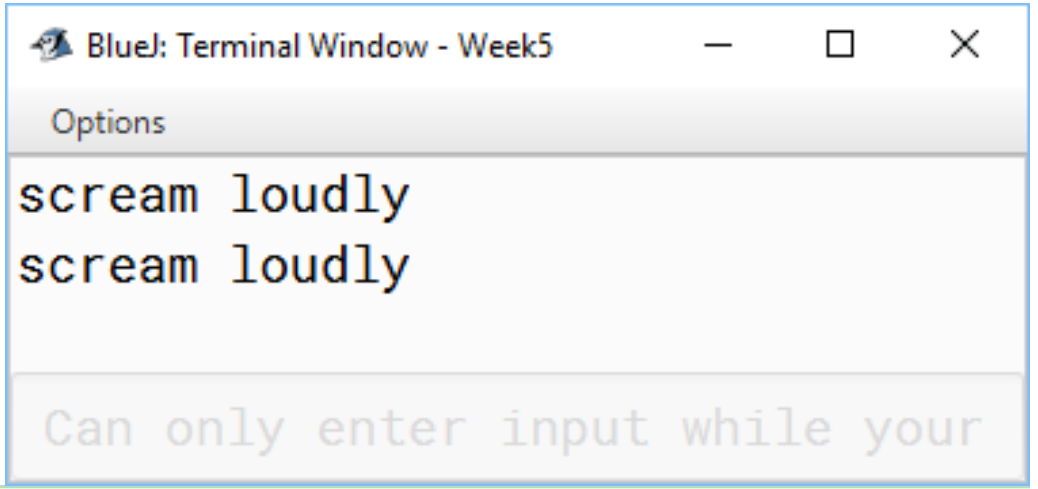
Using a Parent Class Reference for a Child Method

- Polymorphic Methods: OK
- Non-polymorphic methods: Not OK. Parent reference lost child features.

Super Reference cannot Hold Subclass methods.



Casting back to subclass reference to regain a method.




```
1  public class Baby3 extends FamilyPerson{
2     public String throwTantrum(){ return "scream loudly"; }
3     public static void main(String[] args){
4           Baby3 junior               = new Baby3();
5           FamilyPerson babyCousin = new Baby3();
6           System.out.println(junior.throwTantrum());
7           //System.out.println(babyCousin.throwTantrum());
8           System.out.println(((Baby3) babyCousin).throwTantrum());
9     }
10 }
```

SECTION 4

Type Compatibility

Downcasting

- Consider the statements

```
Student s = new GradStudent();
GradStudent g = new GradStudent();
int x = s.getID(); Ilcompile-time error
int y = g.getID(); Illegal
```

- Both sand g represent GradStudent objects, so why does s .getIDO cause an error? The reason is that s is of type Student, and the Student class doesn't have a getID method. At compile time, only nonprivate methods of the Student class can appear to the right of the dot operator when applied to s. Don't confuse this with polymorphism: getID is not a polymorphic method. It occurs in just the GradStudent class and can therefore be called only by a GradStudent object.

Downcasting

- The error shown above can be fixed by casting s to the correct type:

```
int x = ((GradStudent) s).getID();
```

- Since s (of type Student) is actually representing a **GradStudent** object, such a cast can be carried out. Casting a superclass to a subclass type is called a downcast.

Downcasting

Note

1. The outer parentheses are necessary, so will still cause an error, despite the cast. This is because the dot operator has higher precedence than casting, so s .getIDO is invoked before s is cast to GradStudent.

2. The statement

```
int y = g.getID();
```

compiles without problem because g is declared to be of type GradStudent, and this is the class that contains getID. No cast is required.

Downcasting

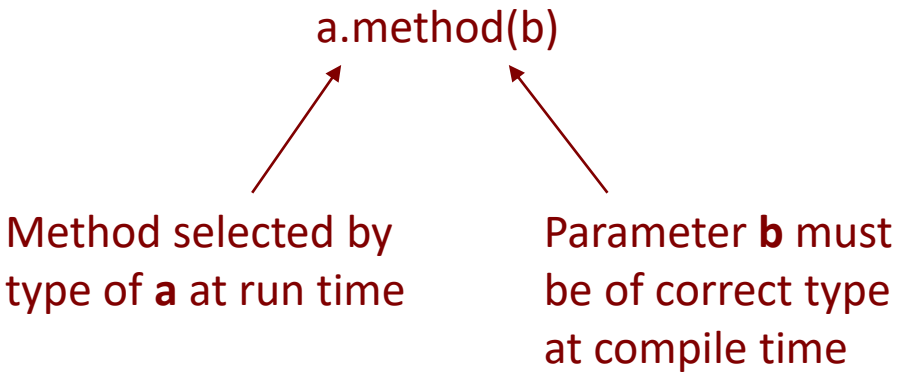
3. Class casts will not explicitly be tested on the AP exam. You should, however, understand why the following statement will cause a compile-time error:

```
int x = s.getID(); // No getID method in Student class
```

And the following statement will compile without error:

```
int y = g.getID(); // getID method is in GradStudent
class
```

Type Rules for Polymorphic Method Calls



- For a declaration like

```
Superclass a = new Subclass();
```

  the type of a at compile time is Superclass; at run time it is Subclass.

- At compile time, method must be found in the class of a, that is, in Superclass. (This is true whether the method is polymorphic or not.) If method cannot be found in the class of a, you need to do an explicit cast on a to its actual type.
- For a polymorphic method, at run time the actual type of a is determined-Subclass in this example-and method is selected from Subclass. This could be an inherited method if there is no overriding method.
- The type of parameter b is checked at compile time. It must pass the **is-a** test for the type in the method declaration. You may need to do an explicit cast to a subclass type to make this correct.

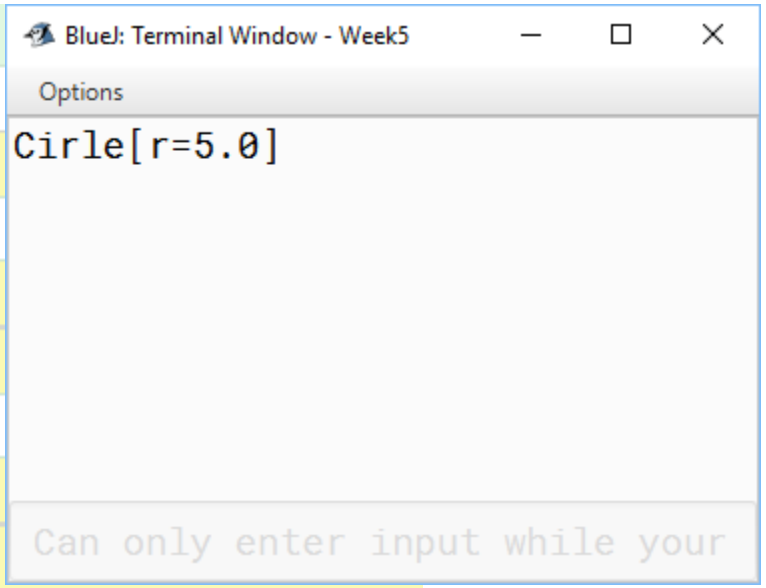
SECTION 5

Object Class

Use of Object Class

- Top level class. All classes are its sub-class.
- Use for inclusion polymorphism. (All objects can be in a Object[] array, or ArrayList<Object>). All objects can be retrieved from the containers and casted back the original data type. But, it is not very convenient if the code is written by other programmers.
- Polymorphic Methods (Often used):
  - toString()
  - equals()
  - getClass()

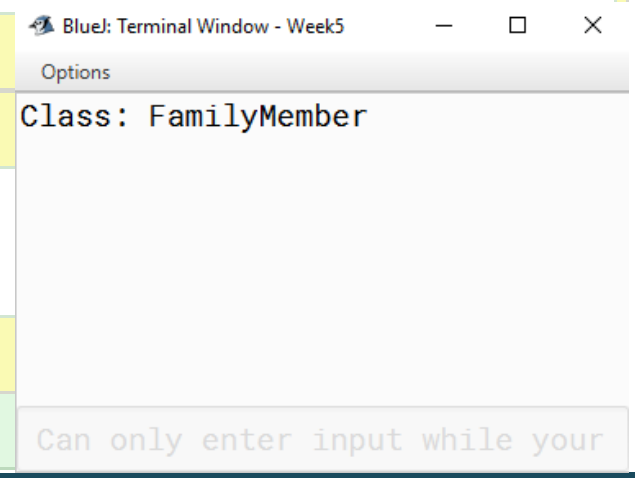
```java
public class Circle{
   double radius = 10.0;
   Circle(double r){  // constructor
       radius = r;
     }
   public double getArea(){
       return Math.PI * radius * radius;
     }
   public double getPerimeter(){
       return 2*Math.PI*radius;
     }
   public double getRadius(){ return radius; }
   public void   setRadius(double r){ radius = r; }
   public String toString(){ return "Cirle[r="+radius+"]";}
   public static void main(String[] args){
       Circle circle = new Circle(5.0);
       System.out.println(circle);
     }
}
```



Modified Circle.java

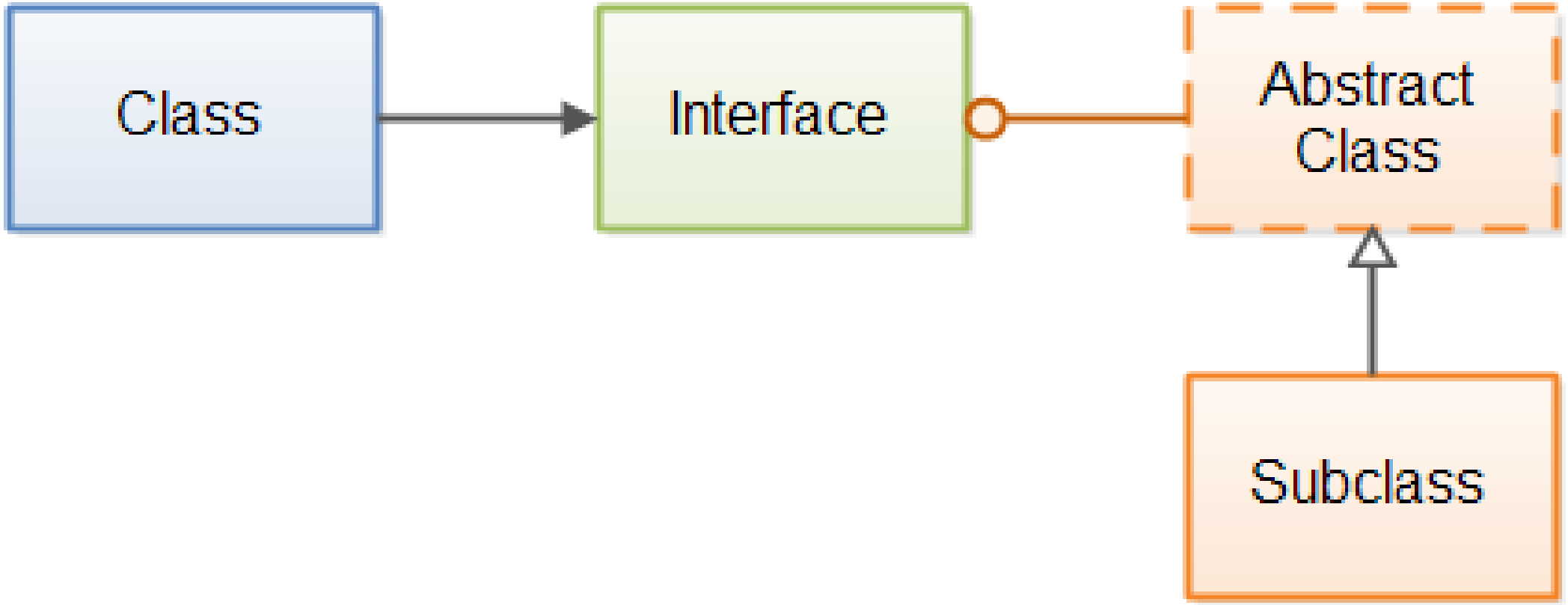
Inclusion Polymorphism

```
1  public class FamilyMember{
2      public void eat(){
3          System.out.println("Class: "+getClass().getName());
4      }
5      public static void timeToEat(Object hungryMember){
6          ((FamilyMember) hungryMember).eat();
7      }
8      public static void main(String[] args){
9          FamilyMember uncleDon = new FamilyMember();
10         timeToEat(uncleDon);
11     }
12 }
```



SECTION 6

Abstract Class


Class
Interface
Abstract Class
Subclass

	Abstract Class	Interface
Declaration	`abstract class Foo {…}`	`interface Foo {…}`
Methods	some or no abstract methods	all methods has no body
Instantiation	—	—
Inheritance	an abstract class can extend only one class (abstract or not)	an interface can extend many interfaces
Implementation	a class can extend only one abstract class (if the class doesn't implement all the abstract methods, it will be also abstract)	a class can implements any number of interfaces
Field Types	public, protected, private, static, final	only public, static, final
Method Types	public, protected, private	only public

Abstract Class

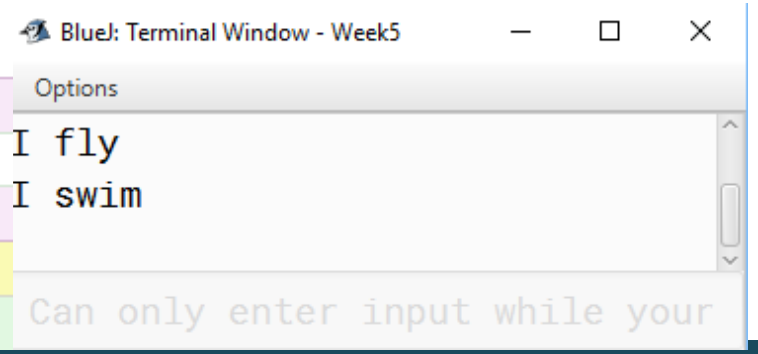
- An abstract class
  - Must include the keyword abstract in the class declaration
  - Can include no method declaration that has the keyword **abstract** in its method signature and does not contain implementation. But a class with abstract method must be an abstract class.
  - Cannot be instantiated.
  - Can include instance variables and implemented (**concrete**) methods
- A subclass that extends the abstract class must implement the abstract methods from the abstract class (unless the subclass is also an abstract class!).
- A subclass can only extend **one** abstract class.
- An **abstract** class works really well for designs that include polymorphism.

```java
public class Fish extends GameCharacter{
    public String move(){ return "I swim"; }
}
```

```java
public class Bird extends GameCharacter{
  public String move(){  return "I fly"; }
}
```

```java
public abstract class GameCharacter{
  public String sayHello(){ return "Hello"; }
  public abstract String move();
}
```

```java
import java.util.ArrayList;
import java.util.List;
public class GameCharacerTester{
    public static void main(String[] args){
        GameCharacter character1 = new Bird();
        GameCharacter character2 = new Fish();
        List<GameCharacter> team = new ArrayList<GameCharacter>();
        team.add(character1); team.add(character2);
        for (GameCharacter character: team){
           System.out.println(character.move());
         }
     }
}
```



SECTION 7

Interface

Interface

- An interface
  - Must include the keyword **interface** in its declaration.
  - Can only contain method declarations (and not the code)
  - Cannot include any instance variables or constructors
  - Cannot be instantiated (you cannot make an object from an interface)
  - Cannot include any implemented (concrete) methods
- Any subclass that implements an interface must contain the code for the methods of the interface.
- A class can implement more than one interface.
- An interface works well for designs that include polymorphism.

```java
public class SmartPhone implements NetflixPlayer{
   public String play() { return "Pressed play on a SmartPhone."; }
   public String pause() { return "Pressed pause on a SmartPhone."; }
   public String rewind() { return "Pressed rewind on a SmartPhone."; }
}
```

```java
public class Laptop implements NetflixPlayer{
   public String play() { return "Pressed play on a Laptop."; }
   public String pause() { return "Pressed pause on a Laptop."; }
   public String rewind() { return "Pressed rewind on a Laptop."; }
}
```

```java
public interface NetflixPlayer{
   String play();
   String pause();   // data field must be fianl static (constant)
   String rewind();  // public static by default
}
```

```java
import java.util.ArrayList;
import java.util.List;
public class NetflixPlayerTester{
 public static void main(String[] args){
     NetflixPlayer device1 = new SmartPhone();
     NetflixPlayer device2 = new Laptop();
     List<NetflixPlayer> devices = new ArrayList<NetflixPlayer>();
     devices.add(device1);
     devices.add(device2);
     for (NetflixPlayer device: devices){
           System.out.println(device.play());
           System.out.println(device.pause());
           System.out.println(device.rewind());
           System.out.println();
        }
  }
}
```

BlueJ: Terminal Window - Week5

Options

```
Pressed play on a SmartPhone.
Pressed pause on a SmartPhone.
Pressed rewind on a SmartPhone.

Pressed play on a Laptop.
Pressed pause on a Laptop.
Pressed rewind on a Laptop.
```

Can only enter input while your pr

Interface

- Membership (Inclusion Polymorphism)
- Capability (Polymorphic Method)

Abstract Class

- Unfinished Class
- Template of a class
- Adapter Pattern (Concretize abstract methods from Interfaces with pass block {})

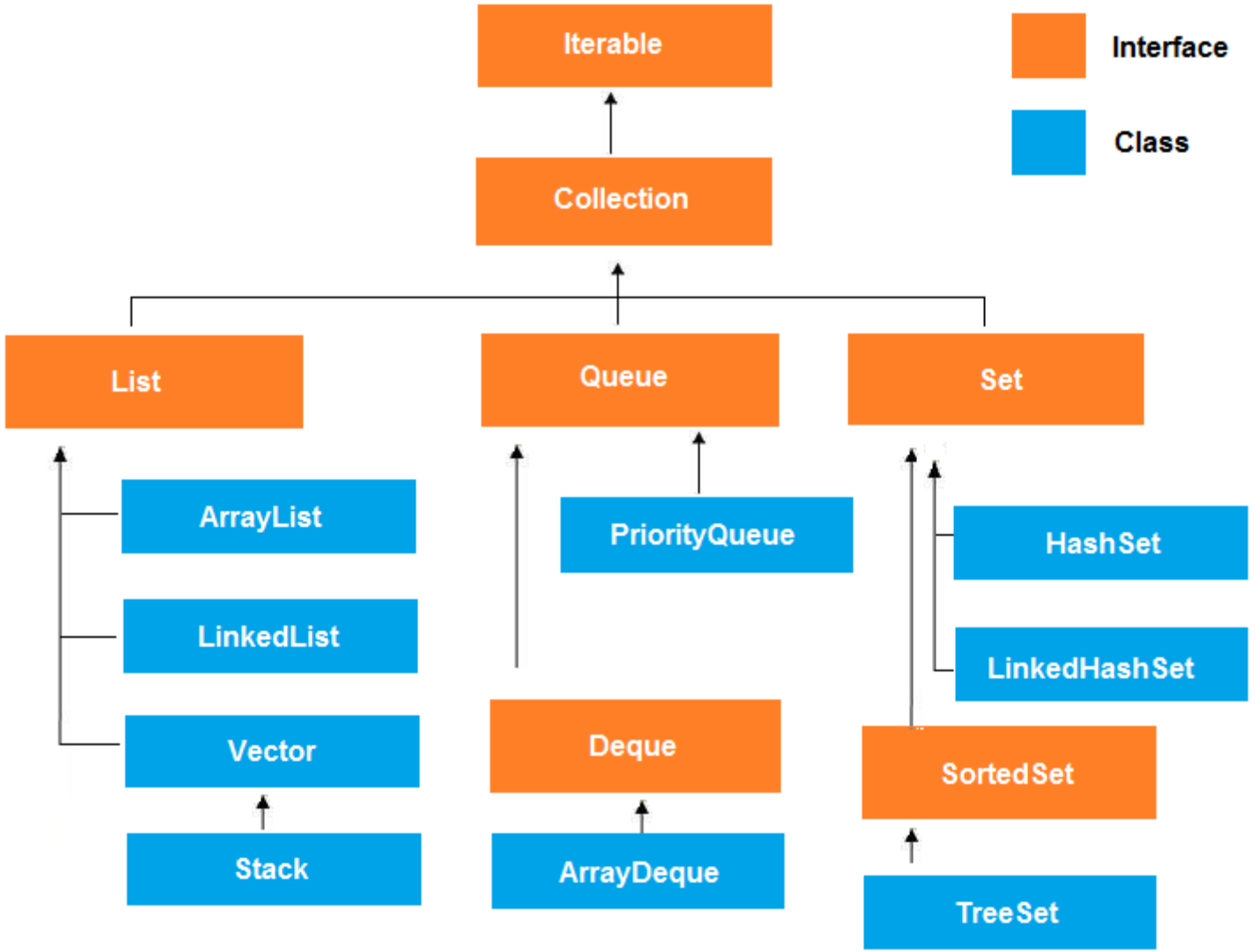
Leaving abstract methods only to be implemented in Concrete classes.

SECTION 8

API Interfaces

API Interface

- Iterable – Iterator()
- Iterator – hasNext(), next(), remove() → for-each loop
- Comparable – compareTo() → Arrays.sort()
- Clonable – none → copyTo(), arrayCopy()
- List (-> Collection -> Iterable (Iterator)) → add(), set(), get(), remove(), and … (Implements ListIterator Interface)


Iterable
Interface
Class
Collection
List
Queue
Set
ArrayList
LinkedList
Vector
Stack
PriorityQueue
Deque
ArrayDeque
HashSet
LinkedHashSet
SortedSet
TreeSet

SECTION 9

Use of Is_A relationship

Summary

- New Data Type conversion: int[] to MyArray
- Combination of Array and ArrayList

  ArrayList< ArrayList<Integer>> to ArrayList< Alist>
- Creating Data Type of different Flavors

  Student -> HighSchoolStudent, CollegeStudent, GradudateStudent
- Polymorphic Method Max, Min