eC Academy

Realize Your Dreams

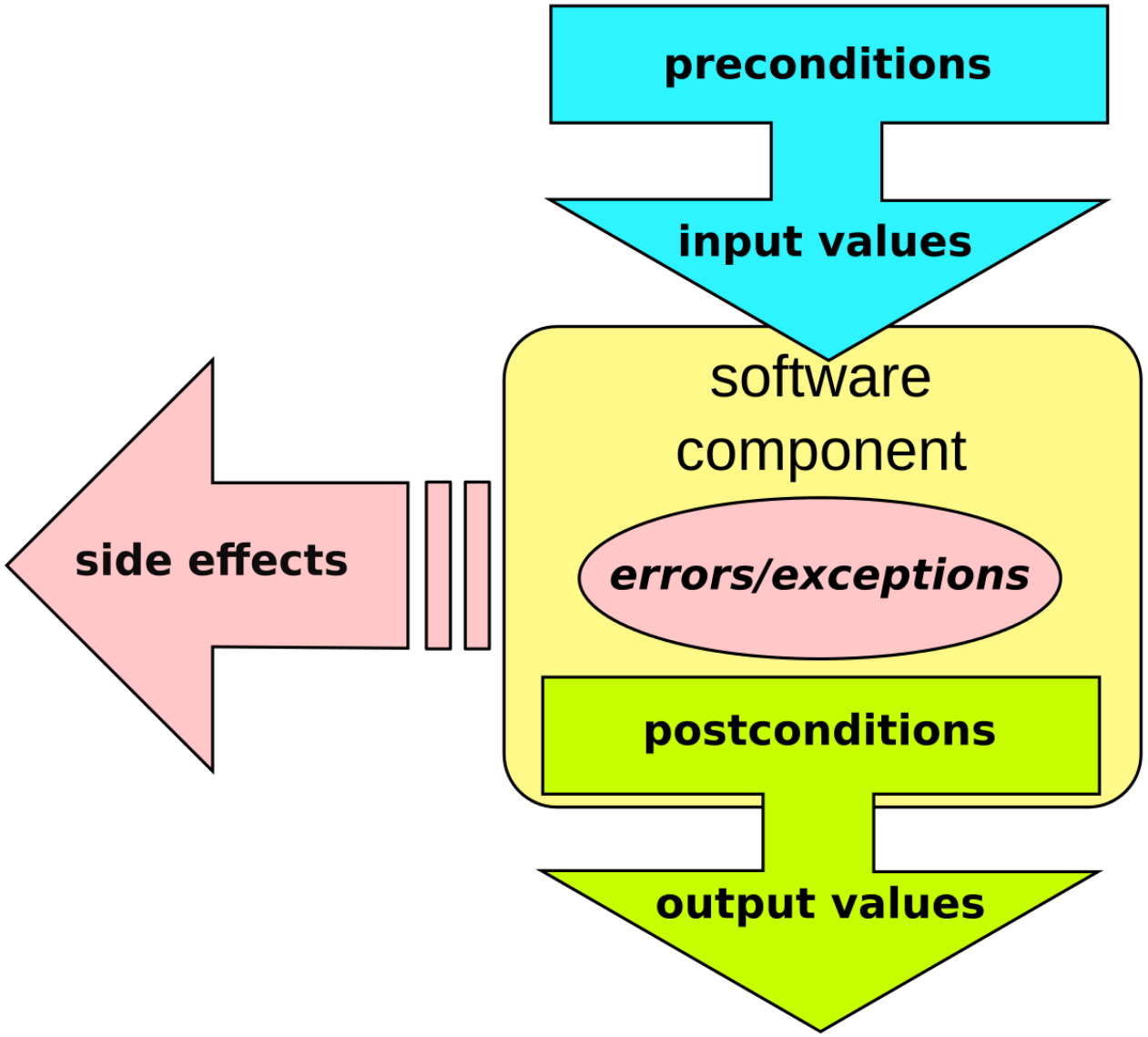
AP Computer Science A Review

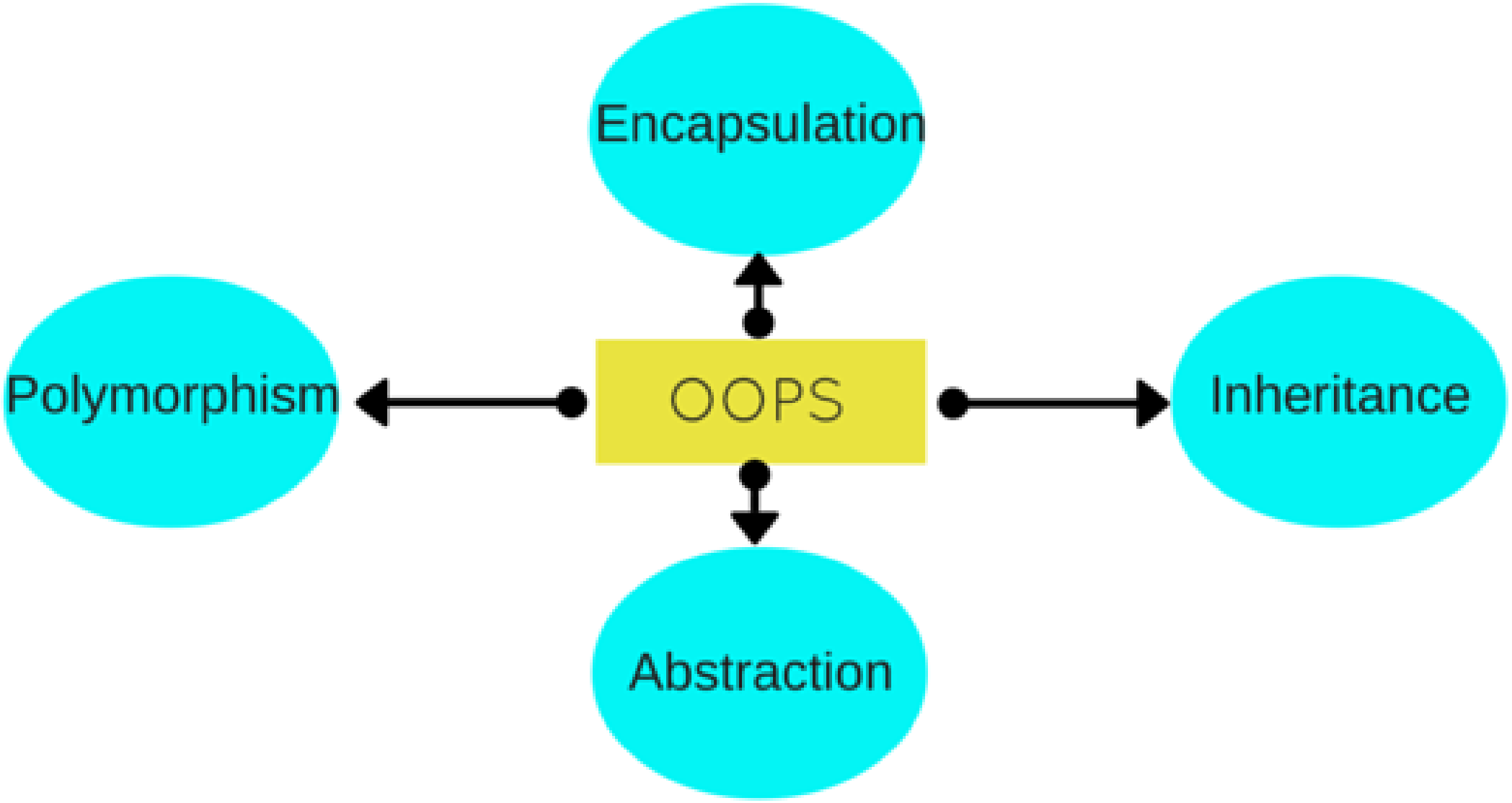
Week 4: Object-Oriented Programming

DR. ERIC CHOU
IEEE SENIOR MEMBER

SECTION 1

Overview of Object-Oriented Programming


preconditions
input values
software
component
errors/exceptions
postconditions
output values
side effects


Encapsulation
Polymorphism
OOPS
Inheritance
Abstraction

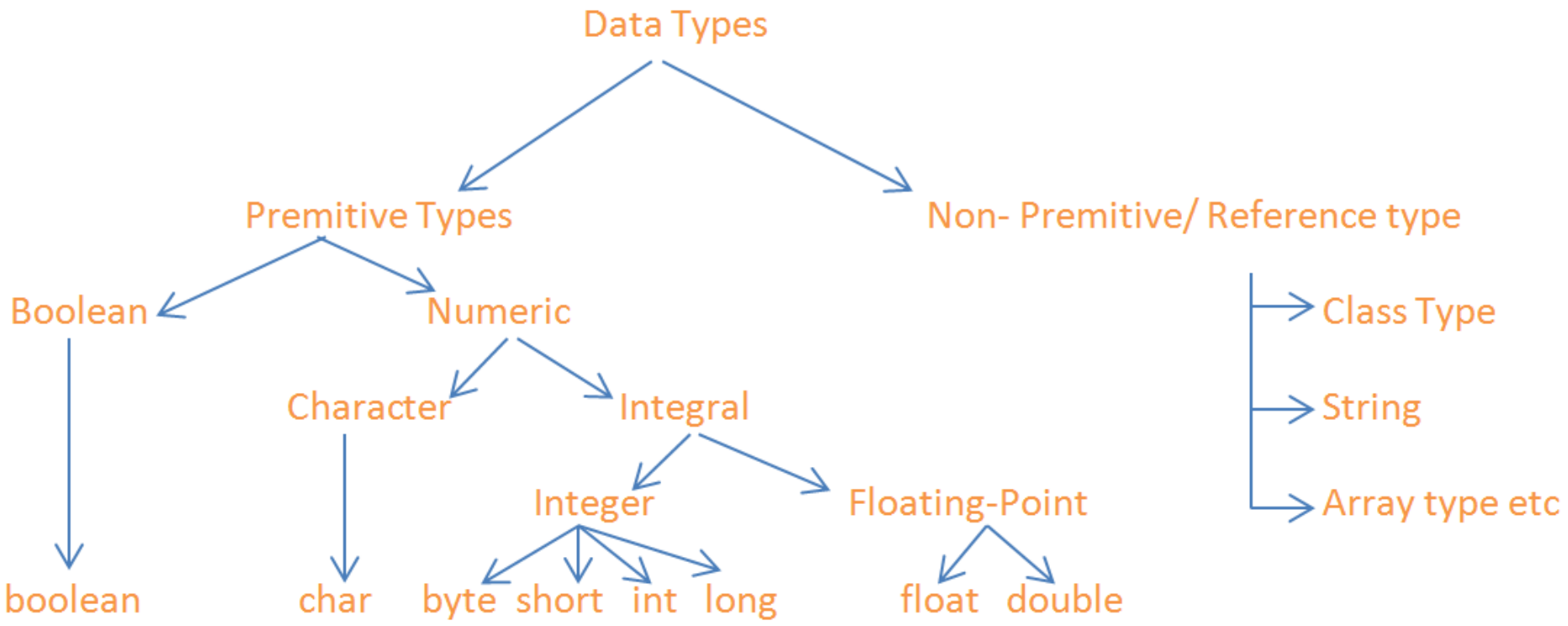
Object-Oriented Programming

- Package
- Module
- Classes
- Interfaces
- Abstract Classes
- enum
- Statics
- Objects
- Functions
- Container
- Constants
- Access Modifiers
- Visibility
  - public
  - protected
  - default
  - private
- Encapsulation
- Information Hiding
- Wrapper Classes
- Immutable
- Relations
- has_A Composition
- Many to 1 Aggregation
- Many to Many Association
- Coherence
- Inheritance
- Is_A Inheritance
- this
- super
- Multiple Inheritance
- Polymorphism
- Overloading
- Overriding
- Dynamic Binding
- Polymorphic Methods
- Generics
- Generic Container
- Generic Method
- Object Generic

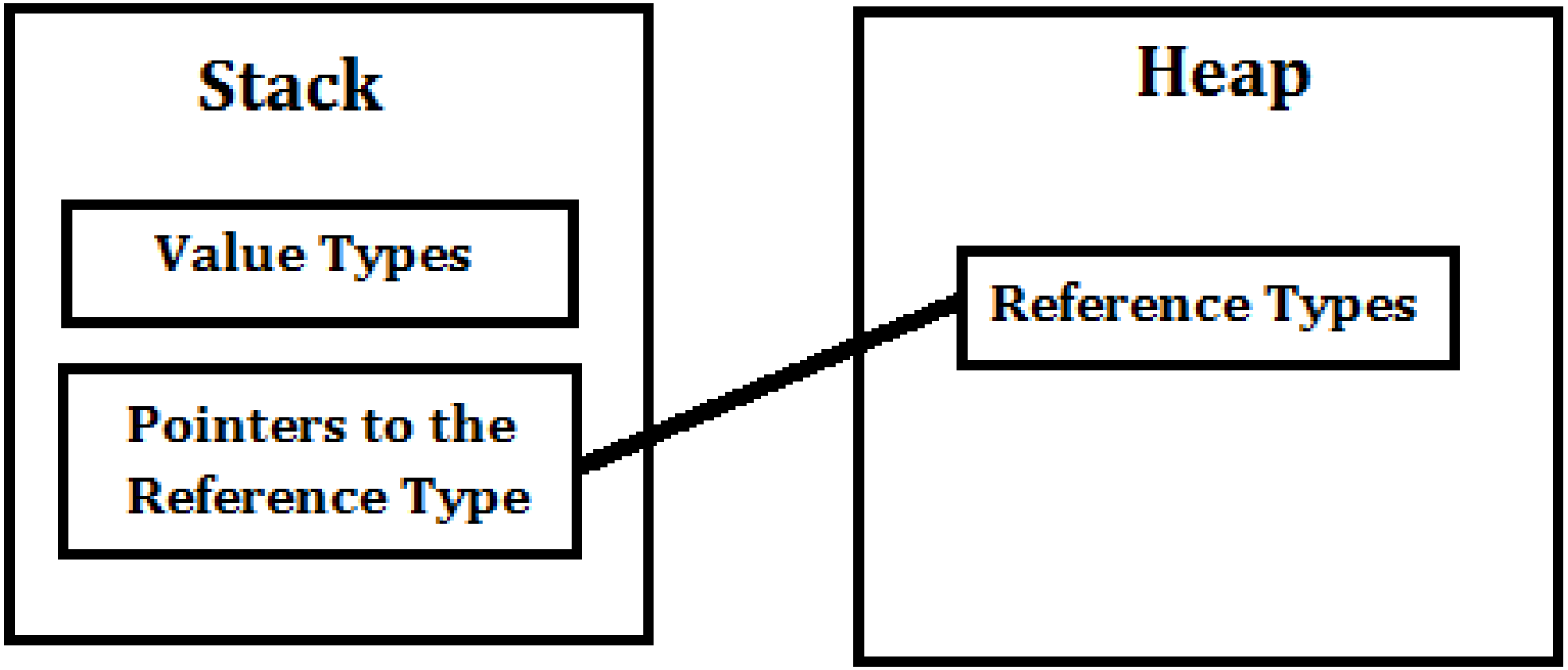
SECTION 1

Java Reference Data Type

Primitive versus Non-Primitive Data Type



Reference Variables



Declaration of A Reference Data Type

variable name

Person p1 = new Person();

datatype of p1
reference datatype

create a new person
object

BASIS FOR COMPARISON	POINTER	REFERENCE
Basic	The pointer is the memory address of a variable.	The reference is an alias for a variable.
Returns	The pointer variable returns the value located at the address stored in pointer variable which is preceded by the pointer sign '*'.	The reference variable returns the address of the variable preceded by the reference sign '&'.
Operators	*, ->	&
Null Reference	The pointer variable can refer to NULL.	The reference variable can never refer to NULL.
Initialization	An uninitialized pointer can be created.	An uninitialized reference can never be created.
Time of Initialization	The pointer variable can be initialized at any point of time in the program.	The reference variable can only be initialized at the time of its creation.
Reinitialization	The pointer variable can be reinitialized as many times as required.	The reference variable can never be reinitialized again in the program.

Pointers

Pointer is indirect addressing, while reference is alias.

C/C++ Pointers vs References

Consider the following code:

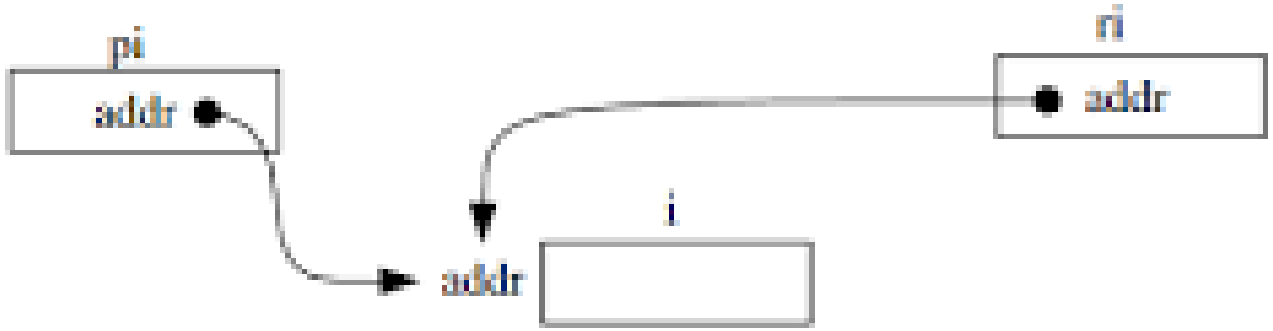
Pointers

```
int i;
int *pi = &i;
```

References

```
int i;
int &ri = i;
```

In both cases the situation is as follows:



Reference Variable

- Hashable
- Polymorphic
- Casting OK
- Inheritance Hierarchy OK.

Works as parameter, return data type, object reference. (First-class Data Type)

Reference is a pointer with restrictions.

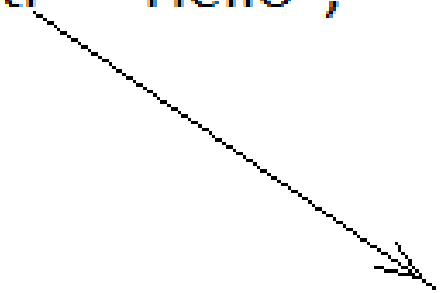
- A reference is a pointer with restrictions.
- A reference is a way of accessing an object, but is not the object itself. If it makes sense for your code to use these restrictions, then using a reference instead of a pointer lets the compiler to warn you about accidentally violating them.

Java Variable

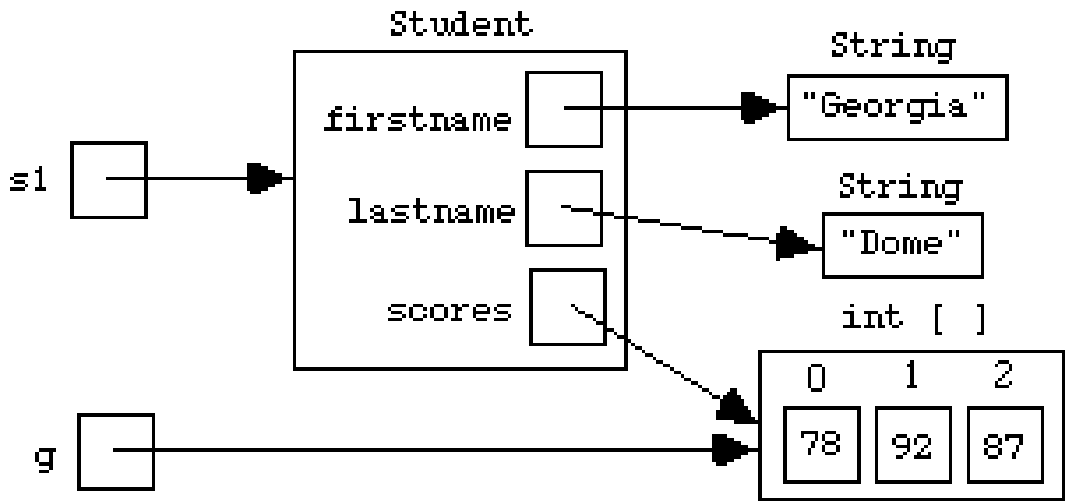
```
int count = 100;
```

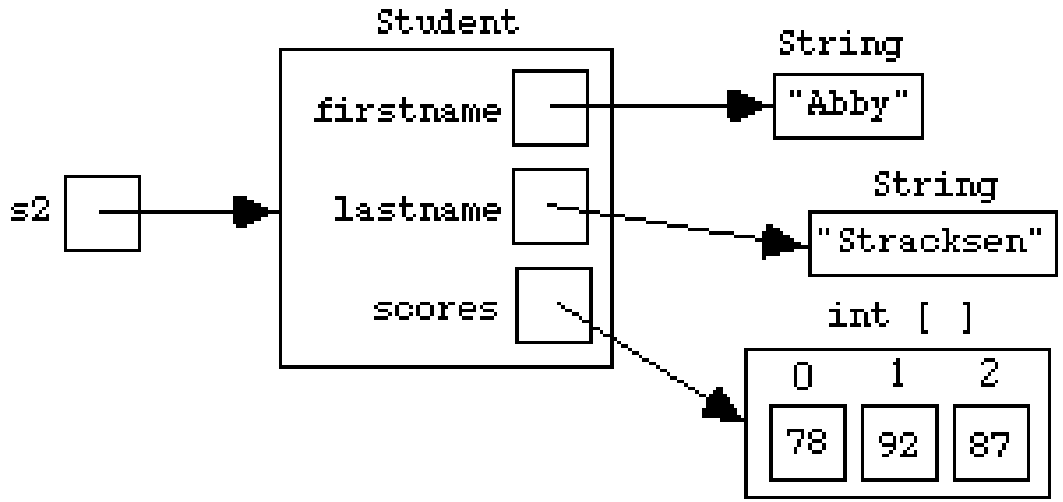
Memory Data	Memory Address
100	240056

```
String str = "Hello";
```



240713	
Hello	240713





this is object reference

super is class reference.

Reference can be of static or non-static.

SECTION 1

Java Reference Data Examples

Reference(Object, Array, String)

- Reference as name for Object.
- Reference as parameter.
- Reference as return value.

Objects Passed As Reference

- Any object to the reference itself will re-direct the reference.

```
public static f(Circle c1, Circle c2){
  c2 = c1;  // c2 is re-directed
  c2.setRadius(5); // set new radius to the c1's object
}
```

PassByReference.java

```java
public class PassByReference
{
    public static void goAhead(Circle c, int n){
        c.setRadius(50);
        n=50;
    }

    public static void main(String[] args){
        Circle myCircle = new Circle(100);
        int number = 100;
        goAhead(myCircle, number);
        System.out.println(myCircle.getRadius());
        System.out.println(number);
    }
}
```

BlueJ: Terminal Window - ...

Options

```
50.0
100
```

Can only enter input while

Array Passed As Reference

- If the array's data type is also reference type. Any update to the array elements will be done to the element permanently.
- An array itself is also a reference type.
- Update to the array itself (not the elements) is also a re-direction.

PassByReferenceArray

```java
public class PassByReferenceArray
{
    public static void main(String[] args){
      String[] myFavoriteThings = {"Memes", "Vines", "Snapchat"};

      whoops(myFavoriteThings);
      for (String s: myFavoriteThings){
          System.out.println(s);
        }
    }

    public static void whoops(String[] arr){
      for (int i=0; i<arr.length; i++){
          arr[i] = "Fearless Pandas";
        }
    }
}
```

BlueJ: Terminal Window - ...

Options

```
Fearless Pandas
Fearless Pandas
Fearless Pandas
```

Can only enter input while

Non-destructive Array Operation

- Avoid mutator operations to the original array element.
- If updates needed, copy the array to a new array first. And, in this case, if the update need to be returned, return the new array.
- The original array will stay unchanged unless you make the following call:

```
arr = mutating(arr);
```

```java
public class NonDestructive{
    public static void main(String[] args){
      String[] myFavoriteThings = {"Memes", "Vines", "Snapchat"};

      for (String s: myFavoriteThings){
          System.out.println(s);
      }

        String[] watchThis = yeah(myFavoriteThings);
        for (String s: watchThis){
          System.out.println(s);
        }
    }

    public static String[] yeah(String[] arr){
        String[] temp = new String[arr.length];

        for (int i=0; i<arr.length; i++){
            temp[i] = arr[i];
        }
        for (int i=0; i<temp.length; i++){
            temp[i] = "Fearless Pandas";
        }
        return temp;
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
Memes
Vines
Snapchat
Fearless Pandas
Fearless Pandas
Fearless Pandas
```

Can only enter input while your

NonDestructive.java

Comparing Against a null Reference

- null is not an object reference. It can only be compared by == and !=.

```java
public class Null{
   static Circle circle2;
   public static void main(String[] args){
       Circle circle1 = new Circle(10);
       // book is buggy here.
       // Circle circle2;

       System.out.println(doANullCheck(circle1));
       System.out.println(doANullCheck(circle2));
    }

   public static double doANullCheck(Circle c){
        if (c != null){
            return c.getArea();
         }
        else {
            return -1;
         }
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
314.1592653589793
-1.0
```

Can only enter input while your

SECTION 1

Overloaded Constructors

Overloaded Constructors

- If no constructor written in a program, default constructor is A(){}
- If a constructor is written, A(){} is overridden. The no-arg constructor needs to be written as well.
- Write the constructor with the longest parameter list first. Other constructor should just call the longest constructor.

```java
public class Student{
    private String firstName;
    private String lastName;
    private int age;
    public Student(){}
    public Student(String first){
        firstName = first;
    }
    public Student(String first, String last){
        firstName = first;
        lastName  = last;
    }
    public Student(String first, String last, int yearsold){
        firstName = first;
        lastName  = last;
        age       = yearsold;
    }
}
```

Equivalent Constructors (Student2.java)

```java
public class Student2{
    private String firstName;
    private String lastName;
    private int age;
    public Student2(){}
    public Student2(String first){
        this(first, null, 0);
    }
    public Student2(String first, String last){
        this(first, last, 0);
    }
    public Student2(String first, String last, int yearsold){
        firstName = first;
        lastName  = last;
        age       = yearsold;
    }
}
```

Overloading and Overriding

- **Overloading:** same class, same method name, different method signature.
- **Overriding:** child class, same method name, same method signature.

```java
public class Methods{
    public static void doSomething(int param){
        System.out.println("INT: "+param);
    }
    public static void doSomething(double param){
        System.out.println("DOUBLE: "+param);
    }
    public static void doSomething(int a, double b){
        System.out.println("INT: "+a+"  Double: "+b);
    }
    public static void doSomething(double a, int b){
        System.out.println("DOUBLE: "+a+"  INT: "+a);
    }
    public static void main(String[] args){
        int a=3; double b=5;
        doSomething(a);
        doSomething(b);
        doSomething(a, b);
        doSomething(b, a);
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
INT: 3
DOUBLE: 5.0
INT: 3  Double: 5.0
DOUBLE: 5.0  INT: 5.0
```

Can only enter input while your

```java
public class Methods2{
    public static void doSomething(Object param){
        System.out.println("Object: "+param);
    }
    public static void doSomething(Object a, Object b){
        System.out.println("Object: "+a+"  Object: "+b);
    }

    public static void main(String[] args){
        Integer a=3; Double b=5.0;
        doSomething(a);
        doSomething(b);
        doSomething(a, b);
        doSomething(b, a);
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
Object: 3
Object: 5.0
Object: 3  Object: 5.0
Object: 5.0  Object: 3
```

Can only enter input while your

```java
public class Methods3{
    public static <T> void doSomething(T param){
        System.out.println("T: "+param);
    }
    public static <T, E> void doSomething(T a, E b){
        System.out.println("T: "+a+"  E: "+b);
    }

    public static void main(String[] args){
        Integer a=3; Double b=5.0;
        doSomething(a);
        doSomething(b);
        doSomething(a, b);
        doSomething(b, a);
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
T: 3
T: 5.0
T: 3  E: 5.0
T: 5.0  E: 3
```

Can only enter input while your

Difference between Methods2 and Methods3

- Methods2 is by inclusion polymorphism. Methods3 is by parametric polymorphism. (Generic Method)
- Object loses subclass methods. T, E (Generic Class Type) won't
- Methods take too many functions, especially when supported data types are too many.
- Number abstract class can be used to replace Object in Methods2.

SECTION 1

Statics and Finals

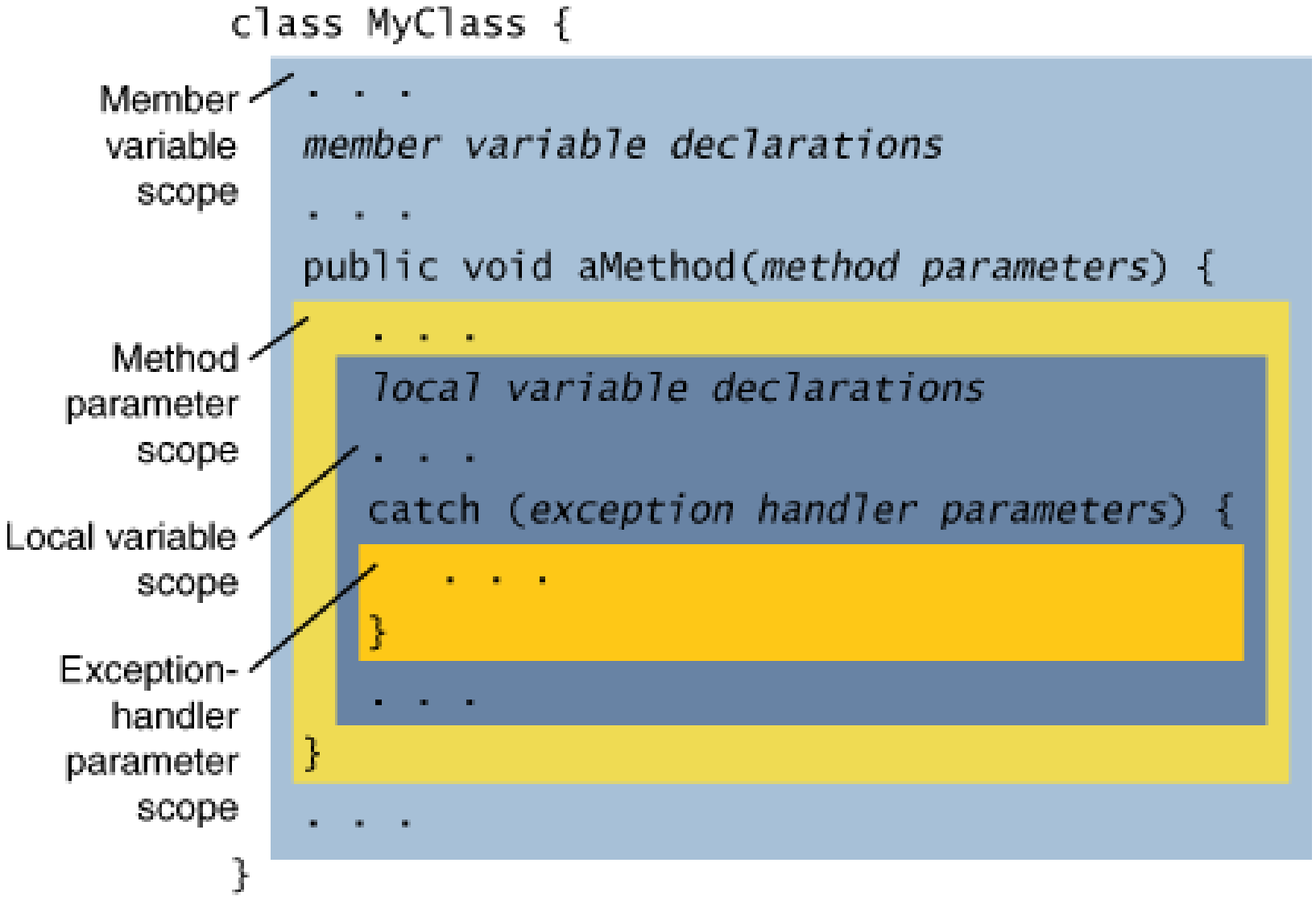
Finals

- final variable -> no change
- final method -> no override
- final class -> no extension

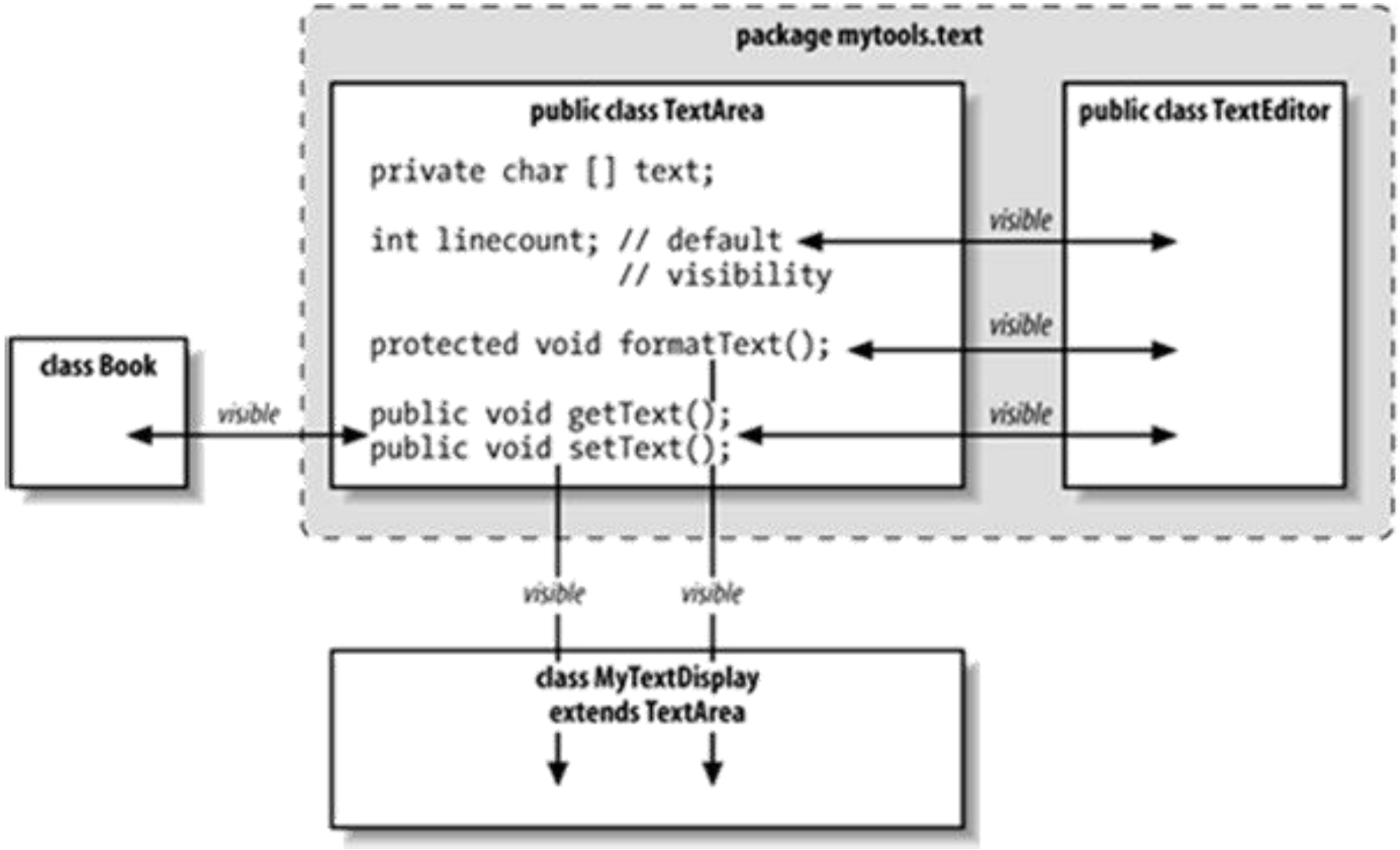
statics

- **Static variables:** Class variables (Shared access to all objects)
- **Static methods:** Class methods (Shared function for utility service purpose)
- **Static class:** Static classes are basically a way of grouping classes together in Java. (No top level static class) (Non-AP)
- **Static import:**

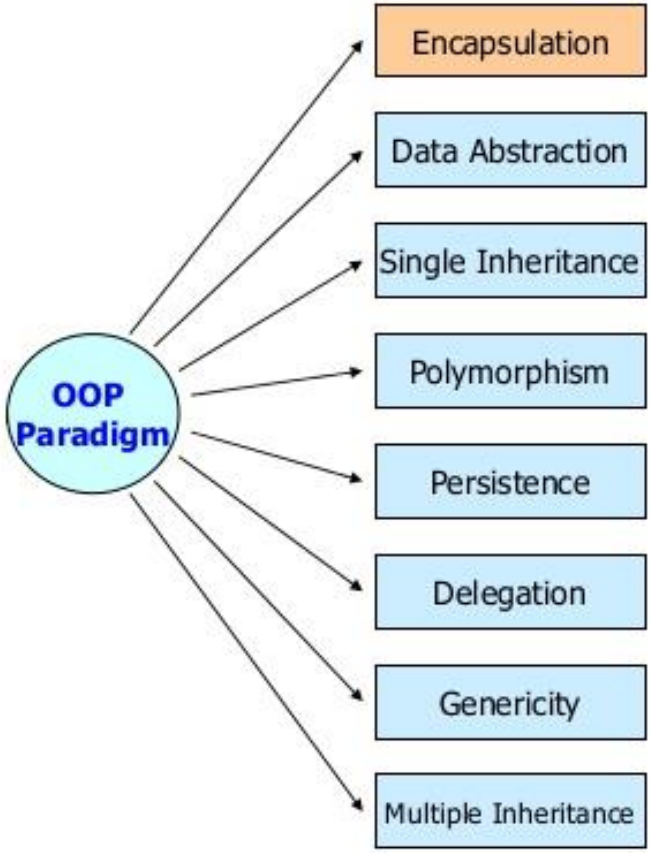
Static import is a feature introduced in the Java programming language that allows members (fields and methods) defined in a class as public static to be used in Java code; without specifying the class in which the field is defined. (Non-AP)


class MyClass {
. . .
member variable declarations
. . .
public void aMethod(method parameters) {
. . .
local variable declarations
. . .
catch (exception handler parameters) {
. . .
}
. . .
}
. . .
}
Member variable scope
Method parameter scope
Local variable scope
Exception-handler parameter scope

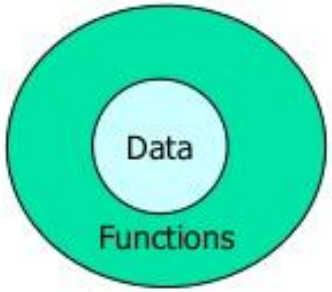
Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No


package mytools.text
public class TextArea
private char [] text;
int linecount; // default
// visibility
protected void formatText();
public void getText();
public void setText();
public class TextEditor
class Book
visible
visible
visible
visible
visible
visible
class MyTextDisplay
extends TextArea

Encapsulation



- It associates the code and the data it manipulates into a single unit; and keeps them safe from external interference and misuse.



Private Access Modifier

```
class A{
   private int data=40;
   private void msg() {
   System.out.println("Hello java");}
   }
public class Test{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);  //Compile Time Error
  obj.msg();  //Compile Time Error
  }
}
```

This is second class where we try to access private variable and method

Private Access Modifier

```
class A{
  private int data;
  public void setA(int data) {
    this.data=data;  }
  public int getA() {
    return this.data; }
}
public class Test{
 public static void main(String args[]){
  A obj=new A();
  obj.setA(12);
  System.out.println("Data is: "+obj.getA());
  }
}
```

Public setter method

SECTION 1

Keyword this

Keyword this

- The keyword this is a reference to the current object, or rather, the object whose method or constructor is being called. It is also referred to as the implicit parameter.
- Demo Program: Batman.java

```java
public class Batman
{
    public static void main(String[] args){
        Batman m1 = new Batman("Batman");
        m1.doSomeThing();
    }

    private String name;
    public Batman(String name){
        this.name = name;
    }

    public String getName(){ return name; }
    public void doSomeThing(){ nowDoSomethingWith(this); }
    public void nowDoSomethingWith(Batman myObject){
        System.out.println("I am "+myObject.getName());
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
I am Batman
```

Can only enter input while your

IllegalArgumentException

- If you pass an argument to a method and the value of the argument does not meet certain criteria required by the method, an **IllegalArgumentException** error may by thrown during run-time.
- Programmers may also choose to write a method that terminates with an **IllegalArgumentException** if it does not receive the expected input.

SECTION 1

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);
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
Cirle[r=5.0]
```

Can only enter input while your

Modified Circle.java

Inclusion Polymorphism

```java
public class FamilyMember{
    public void eat(){
        System.out.println("Class: "+getClass().getName());
    }
    public static void timeToEat(Object hungryMember){
        ((FamilyMember) hungryMember).eat();
    }
    public static void main(String[] args){
        FamilyMember uncleDon = new FamilyMember();
        timeToEat(uncleDon);
    }
}
```

BlueJ: Terminal Window - Week5

Options

```
Class: FamilyMember
```

Can only enter input while your

SECTION 1

Inheritance

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

class MyImplementationClass implements MyInterface

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

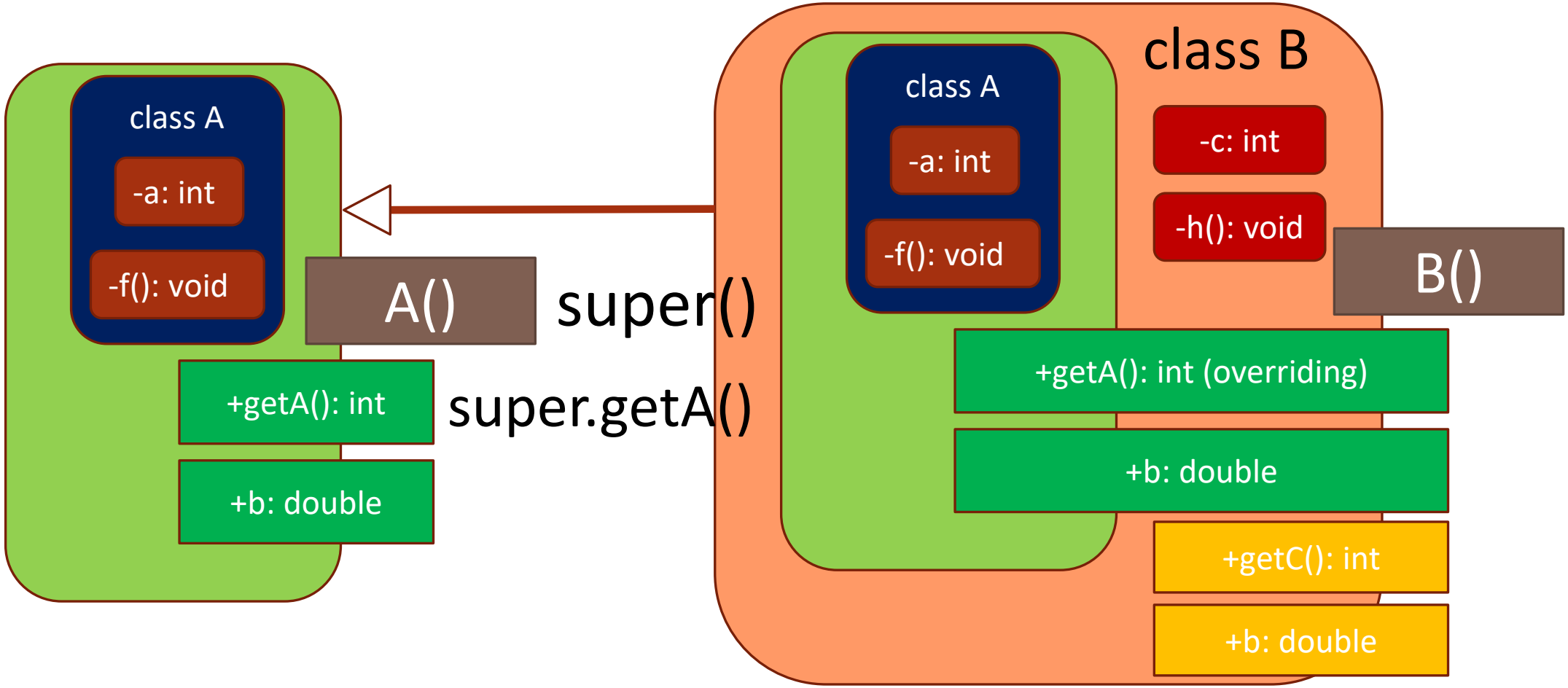
The Keyword super

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());
     }
}
```

BlueJ: Terminal Window - Week5

Options

```
null
null
true
true
Bella
Brown
true
true
```

Can only enter input while your

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.

SECTION 1

Polymorphism

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.

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){
     }
}
```

BlueJ: Terminal Window - Week5

Options

```
cup
cup
cup
fork
hands
spork
```

Can only enter input while your

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

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

```java
public class Baby2 extends FamilyPerson {
   private int age;
   public Baby2(int myAge){ age = myAge; }
   public String eat(){
       if (age > 3) return "hands or a " + super.eat();
       else return "hands";
    }
   public static void main(String[] args){
        Baby2 youngBaby = new Baby2(2);
        Baby2 oldBaby   = new Baby2(4);
        System.out.println(youngBaby.eat());
        System.out.println(oldBaby.eat());
    }
}
```

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.

Baby3 - Week5

Class Edit Tools Options

Baby3 Baby2

Compile Undo Cut Copy Paste Find... Close Source Code

```
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 }
11
```

Error(s) found in class.
Press Ctrl+K or click link on right to go to next error.

saved
Errors: 1

Casting back to subclass reference to regain a method.

BlueJ: Terminal Window - Week5

Options

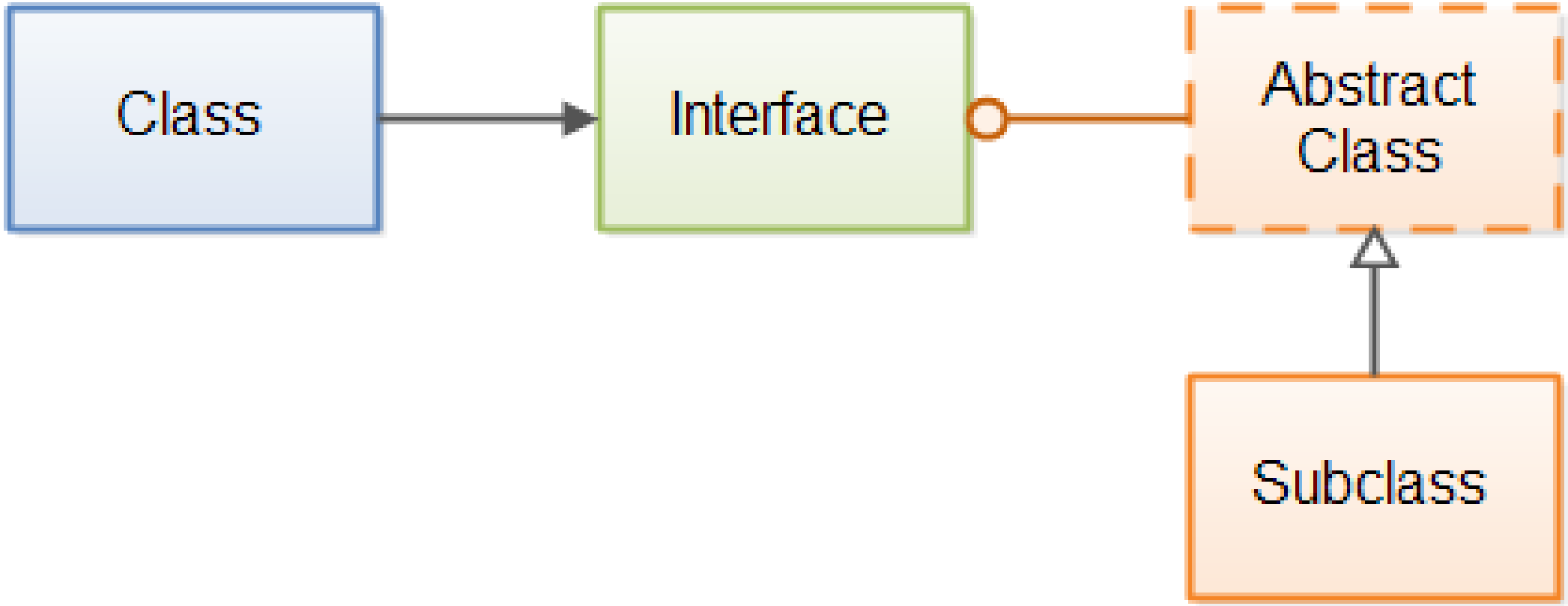
```
scream loudly
scream loudly
```

Can only enter input while your

```java
public class Baby3 extends FamilyPerson{
   public String throwTantrum(){ return "scream loudly"; }
   public static void main(String[] args){
         Baby3 junior            = new Baby3();
         FamilyPerson babyCousin = new Baby3();
         System.out.println(junior.throwTantrum());
         //System.out.println(babyCousin.throwTantrum());
         System.out.println(((Baby3) babyCousin).throwTantrum());
      }
}
```

SECTION 1

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());
         }
     }
}
```

BlueJ: Terminal Window - Week5

Options

```
I fly
I swim
```

Can only enter input while your

SECTION 1

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

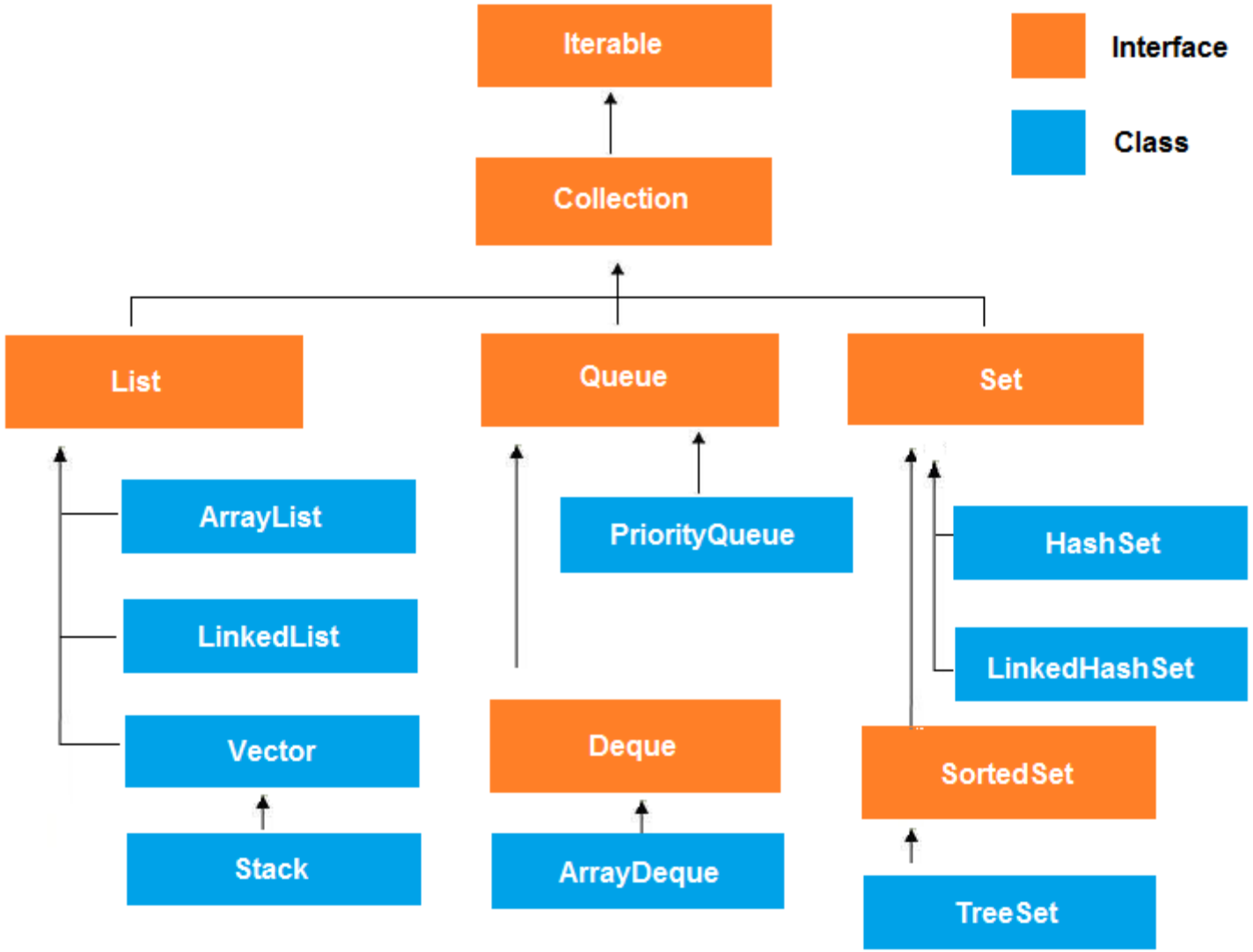
Leaving abstract methods only to be implemented in Concrete classes.

SECTION 1

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
PriorityQueue
HashSet
LinkedList
LinkedHashSet
Vector
Deque
SortedSet
Stack
ArrayDeque
TreeSet