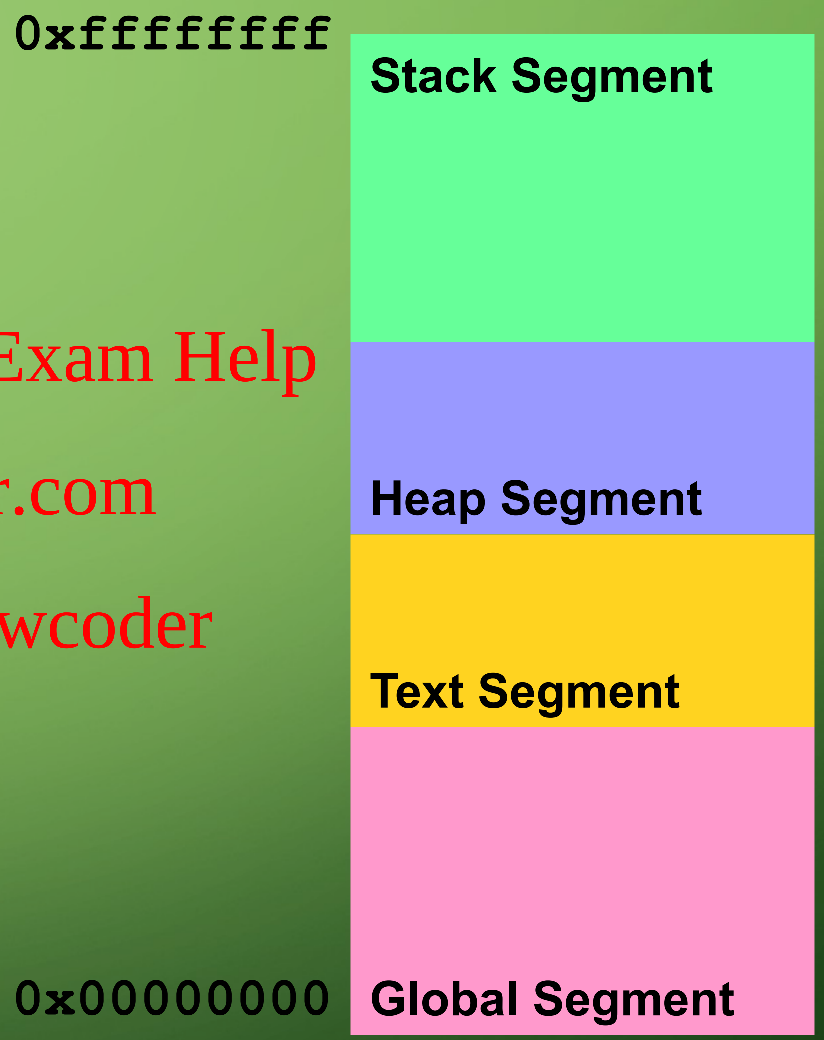
CSE 219
COMPUTER SCIENCE III

OOP++

What is memory?

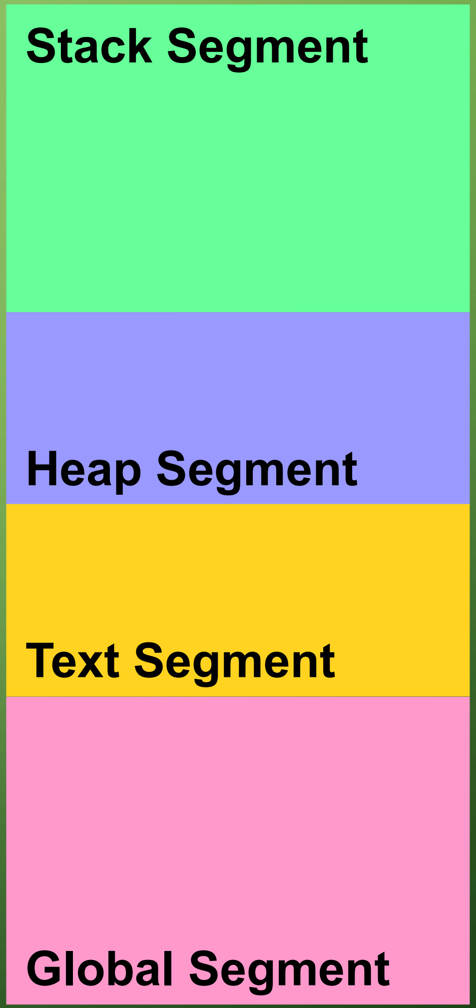
- A giant array of bytes
- How do we assign data to/get data from memory?
  - in Java we don't
  - the JVM does
  - using memory addresses
- We use object ids



What goes in each memory segment?

- **Text Segment**
  - stores program instructions

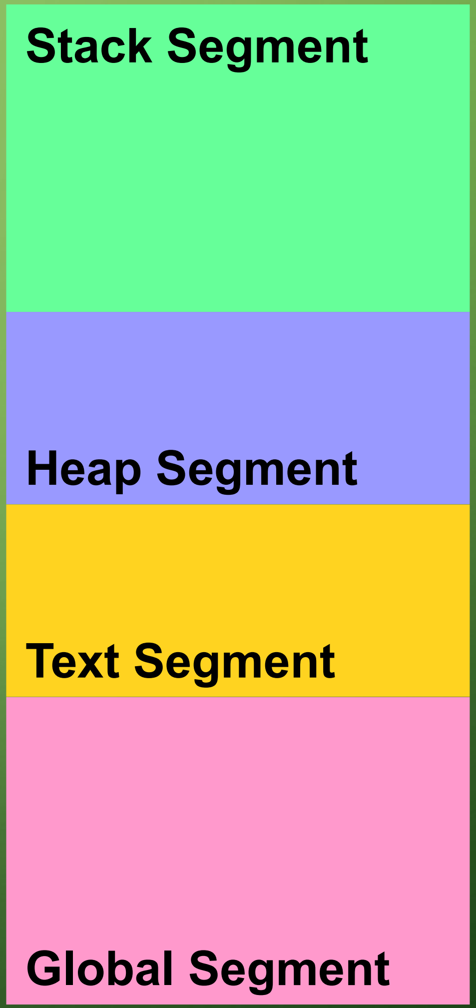
- **Global Segment**
  - data that can be reserved at compile time
  - global data (like static data)



What goes in each memory segment?

- **Stack Segment**
  - temporary variables declared inside methods
  - method arguments
  - removed from memory when a method returns

- **Heap Segment**
  - for dynamic data (whenever you use new)
  - data for constructed objects
  - persistent as long as an existing object variable references this region of memory
    - for Java, C#, Python, etc.



Why do we care?

- Java has Automatic Memory Management

BUT

- We want to be better programmers, right?

- It is related to things we need to know:

  – Type Abstraction & Generics
  – Actual vs. Apparent types
  – Java & Call by Value
  – Static vs. Non-static

How would one design a game framework?

- Not a simple application
- Mixes static and non-static
- Uses lots of inheritance
- It is ***extensible.*** How do achieve this?
  - ***abstraction***

What is abstraction?

- Ignoring certain low-level details of a problem to get a simpler solution
  - Logical first step in any design
  - What parts of the problem can be abstracted out to a higher-level solution?

- Abstraction Techniques:
  - Type Abstraction
  - Iteration Abstraction (Iterator design pattern)
  - Data Abstraction (State design pattern)
  - etc.

Type Abstraction

- Abstract from a data types to *families* of related types
  - a many to one map
  - ex: `public void equals(Object obj)`

- How can we do this?
  - Inheritance & Polymorphism via:
    - Polymorphic variables
    - Polymorphic methods (arguments & return type)

- To understand *type abstraction*, it helps to first know how objects are managed by Java

Types

- A type specifies a well-defined set of values
  - example: int, String
- Java is a strongly typed language
  - compiled code is guaranteed to be type safe
  - one exception: class casting
- Remember the rules of class casting?


```
Student s = new Student();
Person p = (Person) s;
```

Let's think Student extends Person

```
public class Person
{
    public String firstName;
    public String lastName;
    public String toString()
    { return firstName + " " + lastName;      }
}

public class Student extends Person
{
    public double GPA;
    public String toString()
    { return "" + GPA;                 }
}
```

Class Casting

- An object can be cast to an ancestor type
  - It happens automatically in 3 cases. When?

- Ex: **Student extends Person**


```
Person p = new Person();
Student s = new Student();
p = new Student();
s = new Person();
p = (Person)new Student();
p = (Student)new Student();
s = (Person)new Person();
s = (Student)new Person();
```

Which lines would produce compiler errors?

Which lines would produce run-time errors?

Objects as Boxes of Data

- When you call **`new`**, you get an id of a box
  - you can give the box to variables
  - variables can share the same box
  - after **`new`**, we can't add variables to the box

- These rules explain why this is legal:

```
Person p = new Student();
```

```
firstName:   null
lastName:    null
GPA:         0.0
```

- But this is not:

```
Student s = new Person();
```

```
firstName:   null
lastName:    null
```

<Generics>

- The compiler looks out for you

- It's better to get a compiler error than a run-time error
  - motivation behind <generics>

- What is <generics>?

  - specifies families of types for use. Ex:

```
ArrayList<Shape> shapes = new ArrayList();
```

Old Way Versus New Way

- Which is better? Why?

- Old Way:

```
ArrayList people = new ArrayList();
...
Person person = (Person)people.get(0);
```


- New Way:

```
ArrayList<Person> people = new ArrayList();
...
Person person = people.get(0);
```

The Collections Framework

- Learn to use it and you'll love it. Why?
  - because it uses type abstraction
- `ArrayList` implements `List`
  - can be passed to any method that takes a `List` object
- `Collections` methods process `List`s:
  - `Collections.binarySearch`
    - uses `Comparator` for comparisons
  - `Collections.reverseOrder`
  - `Collections.shuffle`
  - `Collections.sort`
    - uses `Comparable` for comparisons

Let's Make our `Students` sortable

- This is ***practical*** type abstraction
- We'll sort them via **`Collections.sort`**

- We'll learn the answer to a common interview question:
  - What's the difference between using **`Comparable`** and **`Comparator`**?

First using Comparable

```
public class Student<T>
      extends Person implements Comparable<Student<T>>
{
    public double GPA;
    public String toString()
    {
        return "" + GPA;
    }

    public int compareTo(Student<T> s)
    {
        if (GPA > s.GPA)          return 1;
        else if (GPA < s.GPA)     return -1;
        else                      return 0;
    }
}
```

What's the output?

```
public class ComparableExample {
    public static void main(String[] args){
        ArrayList<Student> students = new ArrayList();
        Student bob = new Student();
        bob.GPA = 3.9;
        students.add(bob);
        Student joe = new Student();
        joe.GPA = 2.5;
        students.add(joe);
        Student jane = new Student();
        jane.GPA = 3.6;
        students.add(jane);
        Collections.sort(students);
        System.out.println(students);
    }
}
```


Output: [2.5, 3.6, 3.9]

Then using Comparator

```
public class StudentComparator
                 implements Comparator<Student>
{
    @Override
    public int compare(Student s1, Student s2)
    {
        if (s1.GPA > s2.GPA)            return -1;
        else if (s1.GPA < s2.GPA)       return 1;
        else                            return 0;
    }
}
```

What's the output?

```
public class ComparatorExample {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList();
        Student bob = new Student();
        bob.GPA = 3.9;
        students.add(bob);
        Student joe = new Student();
        joe.GPA = 2.5;
        students.add(joe);
        Student jane = new Student();
        jane.GPA = 3.6;
        students.add(jane);
        StudentComparator sc = new StudentComparator();
        Collections.sort(students, sc);
        System.out.println(students);
    }
}
```


Output: [3.9, 3.6, 2.5]

Where's the type abstraction?

- The **`Comparable`** interface provides a standard means for communication with yet unknown types of objects

- What does that mean?
  - It means **`Student`** guarantees an abstract, standard mode of behavior (**`compareTo`**)
  - So, **`Collections.sort`** can sort **`Student`** objects
    - by calling the **`Student`** class' **`compareTo`** method

- Why is this important to us?
  - **Design patterns use lots of type abstraction**

Apparent vs. Actual

- In Java, objects have 2 types
- **Apparent** type
  - the type an object variable was **declared** as
  - the **compiler** only cares about this type
- **Actual** type
  - the type an object variable was **constructed** as
  - the **JVM** only cares about this type
- Important for method arguments and returned objects

Remember Student extends Person

```
public class Person
{
     public String firstName;
     public String lastName;
     public String toString()
     { return firstName + " " + lastName;      }
}

public class Student extends Person
{
     public double GPA;
     public String toString()
     { return "" + GPA;                   }
}
```

What's the output?

```java
public class ActualVsApparentExample
{
    public static void main(String[] args){
        Person p = new Person();
        p.firstName = "Joe";
        p.lastName = "Shmo";
        print(p);
        p = new Student();
        p.firstName = "Jane";
        p.lastName = "Doe";
        print(p);
        Student s = (Student)p;
        print(s);
    }

    public static void print(Person p){
        System.out.println(p);
    }
}
```

Actual vs. Apparent rule of thumb

- Apparent data type of an object determines what methods may be called

- Actual data type determines where the implementation of a called method is defined
  - – JVM look first in actual type class & works its way up

Call-by-Value

- Java methods always use call-by-value

- What does that mean?
  - method arguments are *copied* when sent
  - this includes object **ids**

- Let's see some examples

What's the output?

```
public class CBVTester
{
    public static void main(String[] args)
    {
        Person p = new Person();
        p.firstName = "Joe";
        foo(p);
        System.out.prinln(p.firstName);
    }

    public static void foo(Person fooPerson)
    {
        fooPerson = new Person();
        fooPerson.firstName = "Bob";
    }
}
```

What's the output?

```
public class CBVTester2
{
    public static void main(String[] args)
    {
        Person p = new Person();
        p.firstName = "Joe";
        foo(p);
        System.out.println(p.firstName);
    }

    public static void foo(Person fooPerson)
    {
        fooPerson.firstName = "Bob";
        fooPerson = new Person();
        fooPerson.firstName = "Chris";
    }
}
```

What's the output?

```
public class CBVTester3
{
    public static void main(String[] args)
    {
        Person p = new Person();
        p.firstName = "Joe";
        p = foo(p);
        System.out.println(p.firstName);
    }

    public static Person foo(Person fooPerson)
    {
        fooPerson.firstName = "Bob";
        fooPerson = new Person();
        fooPerson.firstName = "Chris";
        return fooPerson;
    }
}
```

Interfaces

- Specify abstract methods
  - method headers with no bodies

```
public interface ActionListener
{
  public int actionPerformed(ActionEvent ae);
}
```


- A class that implements **`ActionListener`** must define **`actionPerformed`**
  - else a syntax error

- So Swing call your event handler's **`actionPerformed`**

Abstract Classes

- Can specify abstract and concrete methods
- Any class that **extends** an **abstract** class:
  - guarantees it will define all abstract methods, ex:

```
public abstract class AbstractDie {
  protected int upValue = 1;
  protected int numSides = 6;
  public abstract void roll();
  public int getUpValue() { return upValue; }
}

public class Die extends AbstractDie {
  public void roll() {
        upValue = (int)(Math.random()*6)+ 1;
  }
}
```

Interfaces/Abstract classes & Polymorphism

- Similar rules of polymorphism apply

- Objects can have an apparent type of:
  - A concrete class
  - An interface
  - An abstract class

- Objects can never have the actual type of an interface or abstract class. Why?

What about *default* methods?

- Java 8 addition for interfaces. Now you can say:

```
public interface TestInterface {
 public boolean performTest(int expectedValue, int actualValue);
 public static String getDescription(
                 boolean equivalent, int expectedValue, int actualValue) {
   if (equivalent) return "SUCCESS: " + expectedValue + " == " + actualValue;
   else return "FAILURE: " + expectedValue + " != " + actualValue;
 }
 public default String getDescription(int expectedValue, int actualValue) {
  if (performTest(expectedValue, actualValue))
   return "SUCCESS: " + expectedValue + " == " + actualValue;
  else
   return "FAILURE: " + expectedValue + " != " + actualValue;
 }
}
```

What about instance & static variables?

- In interfaces they are automatically *final*

```
public interface TestInterface {
  public static int x = 5;
  public int y = 6;
  public int z;
}
```


Which line has an error?

Interfaces vs. Abstract Classes vs. Concrete Classes

- Which of these:
  - can have instance variables?
  - can have static variables?
  - can have static methods?
  - can have static final constants?
  - can have constructors?
  - can have abstract methods?
  - can have concrete methods?
  - can have default methods?
  - can be constructed?

`static` vs. non-`static`

- `static` methods and variables are important to many design patterns

- What's the difference?

  - `static` (class) methods & variables are scoped to a class
    - one `static` variable for all objects to share

  - non-`static` (object) methods & variables are scoped to a single object
    - each object owns its non-`static` methods & variables

What's the output?

```
public class StaticExample {
    public int nonStaticCounter = 0;
    public static int staticCounter = 0;

    public StaticExample(){
        nonStaticCounter++;
        staticCounter++;
    }

    public static void main(String[] args){
        StaticExample ex;
        ex = new StaticExample();
        ex = new StaticExample();
        ex = new StaticExample();
        System.out.println(ex.nonStaticCounter);
        System.out.println(staticCounter);
    }
```

`static` usage

- Can a **`static`** method:
  - directly call (without using a ".") a non-**`static`** method in the same class?
  - directly call a **`static`** method in the same class?
  - directly reference a non-**`static`** variable in the same class?
  - directly reference a **`static`** variable in the same class?

- Can a non-**`static`** method:
  - directly call (without using a ".") a non-**`static`** method in the same class?
  - directly call a **`static`** method in the same class?
  - directly reference a non-**`static`** variable in the same class?
  - directly reference a **`static`** variable in the same class?

```
public class Nothing {
 private int nada;
 private static int nothing;

 public void doNada()            { System.out.println(nada);       }
 public static void doNothing()
  { System.out.println("NOTHING"); }

 public static void myStaticMethod()   {
          doNada();
          doNothing();
          nada = 2;
          nothing = 2;
          Nothing n = new Nothing();
          n.doNada();
          n.nada = 2;
  n.nothing = 6;
}

public void myNonStaticMethod() {
          doNada();
          doNothing();
          nada = 2;
          nothing = 2;
          Nothing n = new Nothing();
          n.doNada();
          n.nada = 2;
}
```