OOP++

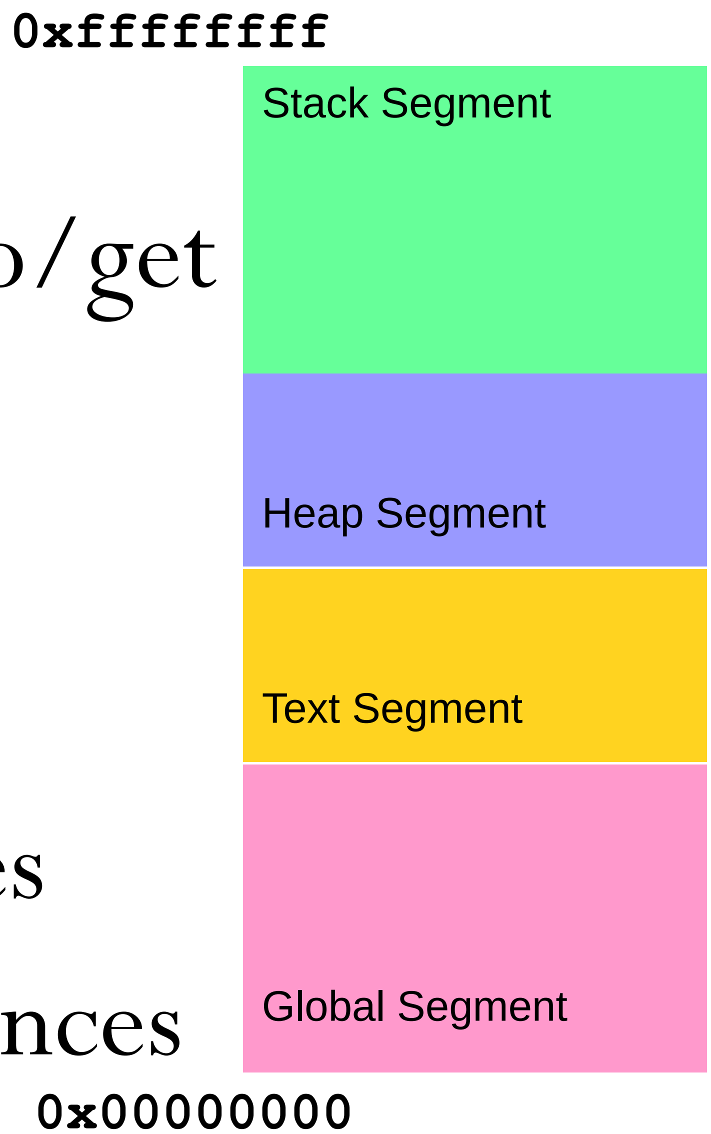
CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219

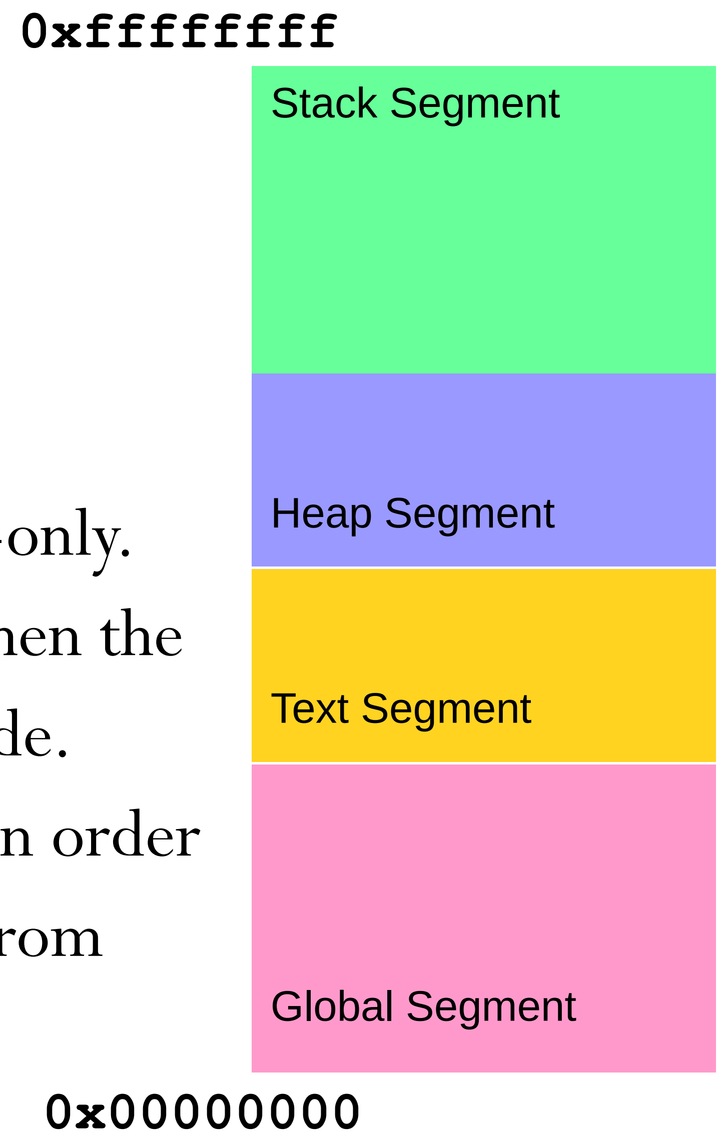
What is memory?

- 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/references



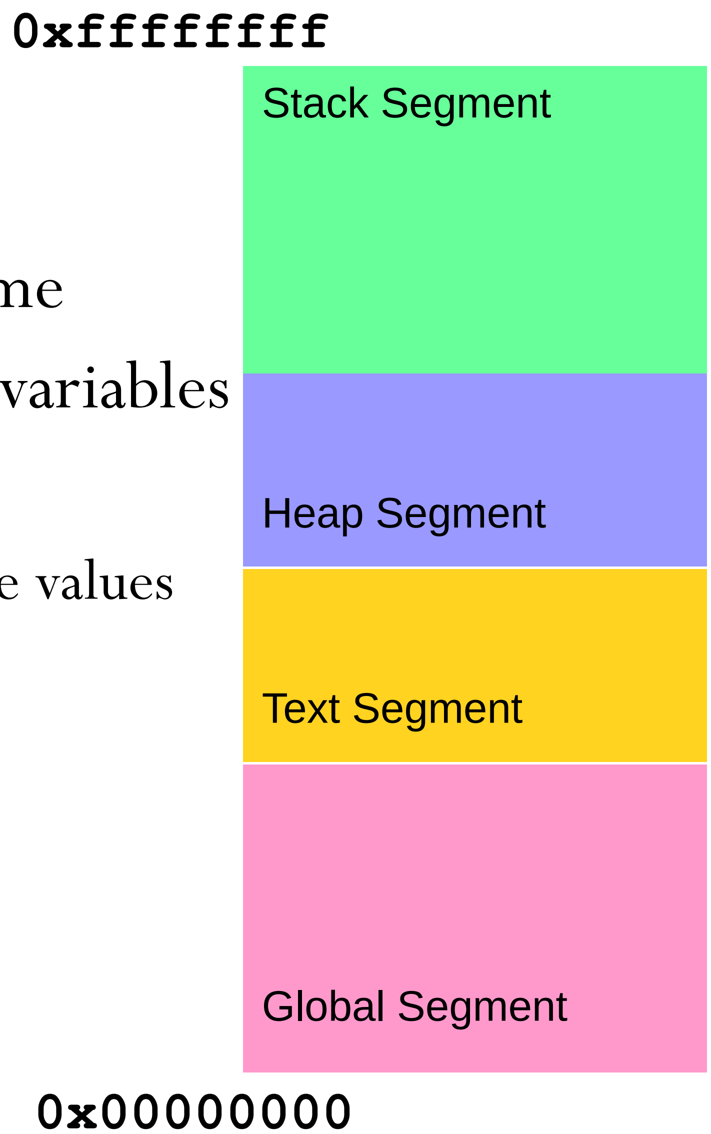
What goes in each memory segment?

- Text Segment
  - Also called **code segment**
  - stores program instructions
    - contains executable instructions
  - It has a fixed size and is usually read-only.
  - If the text section is not read-only, then the architecture allows self-modifying code.
  - It is placed below the heap or stack in order to prevent heap and stack overflows from overwriting it.



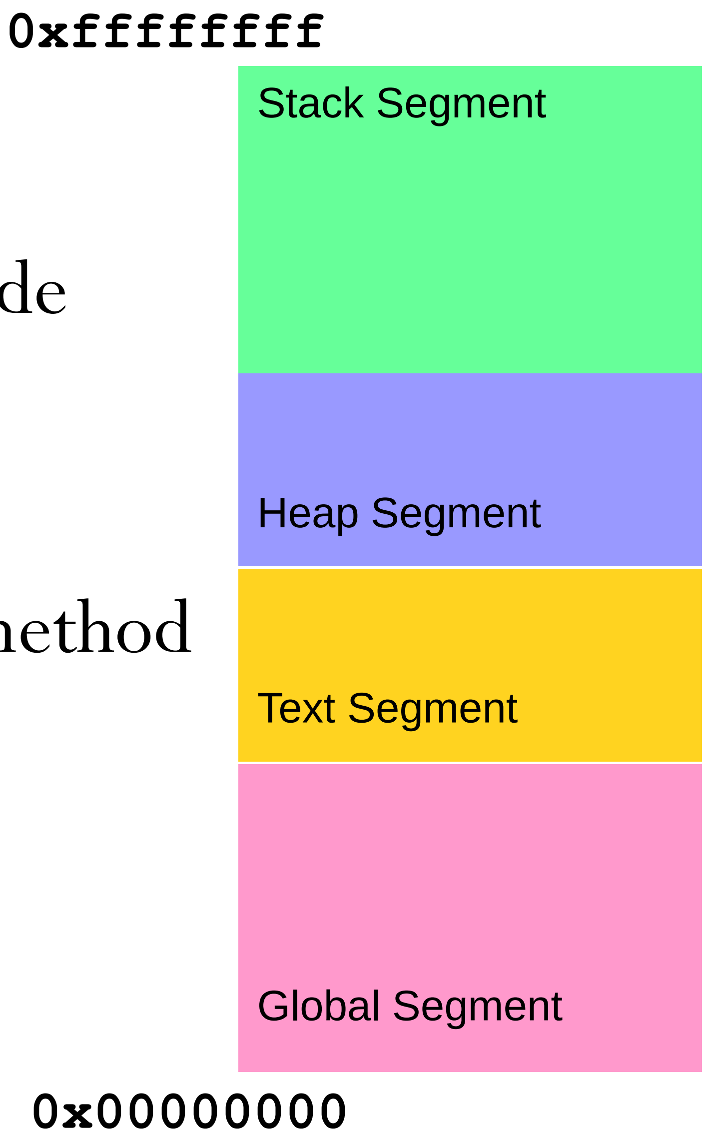
What goes in each memory segment?

- Global Segment
  - data that can be reserved at compile time
  - contains the global variables and static variables that are initialized by the programmer
    - The **data segment** is read-write, since the values of the variables can be altered at run-time.



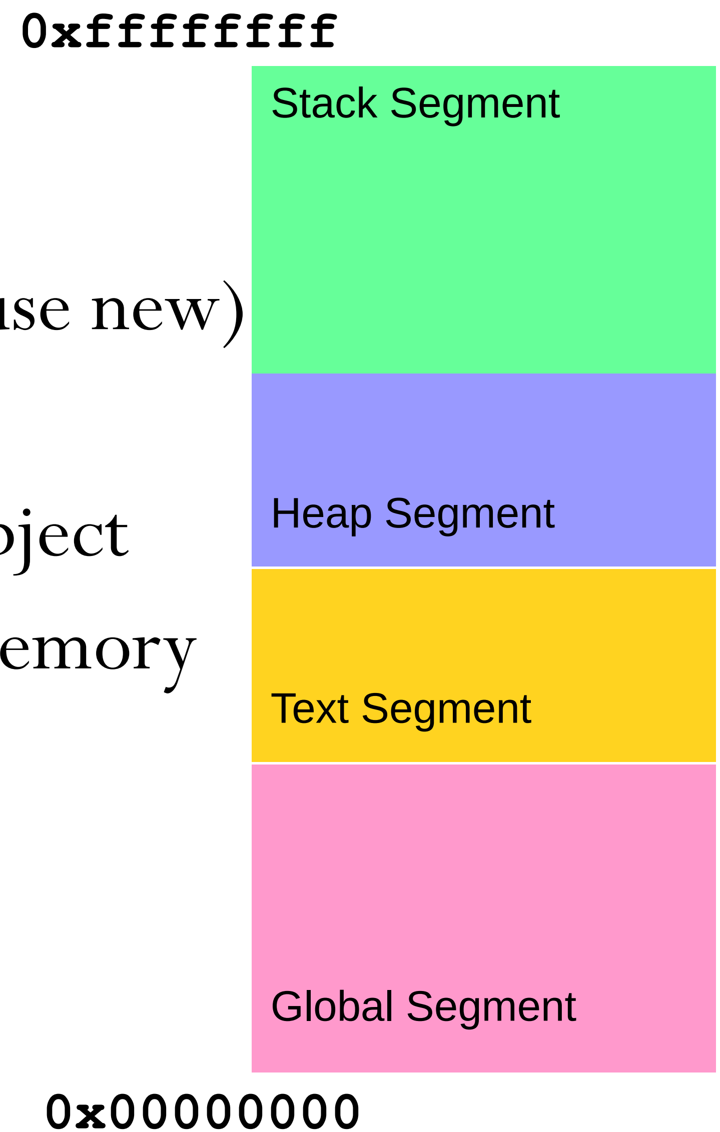
What goes in each memory segment?

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



What goes in each memory segment?

- 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
    - Java, C#, Python, etc.
    - Automatic Garbage Collection



Memory

- Java has Automatic Memory Management
  - Type Abstraction & Generics
  - Actual vs. Apparent types
  - Java & Call by Value
  - Static vs. Non-static
- As users we must know how to write our programs:
  - Call-by-value:
    - The value is copied from arguments (actual parameters into the real parameters)
    - Primitive variables contain the value
      - Once a method returns the local variables are lost
    - The reference variables (class instances) contain the address of the object on the heap (formal params. refer to the same objects, the object is not deleted when the method returns)

Object oriented programming
How would one design a framework?

- Make it *extensible*. How to achieve this?
  - *abstraction*
  - Uses lots of inheritance
- Generics
- Abstract Classes
- Interfaces
- Static vs. dynamic

What is abstraction?

- Ignoring certain low-level details of a problem to get a simpler reusable 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:
  - example:

```
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

```
Student s = new Student();
Person p = (Person) s; // Explicit casting
// OR
Person p = s; // implicit casting
```

Student extends Person

Person.java

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

Student.java

```
public class Student extends Person {
    public double GPA;
    public String toString(){
        return "Student: " + super.toString()
            + ", gpe: " + GPA;
    }
}
```

Class Casting

- An object can be cast to an ancestor type

```
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?

Class Casting

- An object can be cast to an ancestor type

```
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 (reference or address) of a box
  - you can give the address to variables
  - variables can share the same address
  - after **new**, we can't add variables/properties to the box
- These rules explain why implicit casting is legal:

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

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

- But no explicit casting is not:

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

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

<Generics>

- Generic datastructures
  - It's better to get a compiler error than a run-time casting error
- Specifies families of types for use

  Example: `ArrayList<Shape> shapes = new ArrayList();`
- 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

- It uses type abstraction
  - **`ArrayList implements List`**
    - can be passed to any method that takes a List object
- Collections methods process Lists:
  - **`Collections.binarySearch`**
    - uses **`Comparator`** for comparisons
  - **`Collections.sort`**
  - **`Collections.reverseOrder`**
  - **`Collections.shuffle`**
    - uses **`Comparable`** for comparisons

Let's Make our Students sortable

- Practical example of type abstraction
  - We'll sort them via **`Collections.sort`**

- **`Comparable`** and **`Comparator`**

Using Comparable

ComparableExample.java

```
import java.util.ArrayList;
import java.util.Collections;
class ComparableStudent
   implements Comparable<ComparableStudent>{
    public double GPA;
    public String toString() {
         return "" + GPA;
     }
     public int compareTo(ComparableStudent s){
         if (GPA > s.GPA)              return 1;
         else if (GPA < s.GPA)         return -1;
         else                          return 0;
     }
}
```

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

Output: [2.5, 3.6, 3.9]

Using Comparator

ComparatorExample.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
public class StudentComparator
            implements Comparator<Student>{
    @Override
    public int compare(Student s1, Student s2){
/*  Compares its two arguments for order. Returns a negative integer, zero, or a positive
integer as the first argument is less than, equal to, or greater than the second.  */
    if (s1.GPA > s2.GPA)                return -1;
        else if (s1.GPA < s2.GPA)        return 1;
        else                             return 0;
    }
```

ComparatorExample.java

```
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]

Type abstraction

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

Example: **Person** p = new **Student**(…);

- Very important for method arguments and returned objects

Apparent vs. Actual Example

Remember Person and Student classes

Person.java

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

Student.java

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

    }
}
```

Apparent vs. Actual

```
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);
    }
}
```

ActualVsApparentExample.java

Apparent vs. Actual

- 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
  - Dynamic binding

Call-by-Value

- Java methods always use call-by-value:
  - method arguments are ***copied*** when sent
  - this includes object **ids**

Call-by-Value

CallByValueTester1.java

```
public class CallByValueTester1 {
    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 = new Person();
        fooPerson.firstName = "Bob";
    }
}
```

Call-by-Value

```
public class CallByValueTester1 {
    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 = new Person();
        fooPerson.firstName = "Bob";
    }
}
```

Output: Joe

Call-by-Value

CallByValueTester2.java

```
public class CallByValueTester2 {
    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";
    }
}
```

Call-by-Value

```
public class CallByValueTester2 {
    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";
    }
}
```

Output: Bob

Call-by-Value

CallByValueTester3.java

```
public class CallByValueTester3 {
    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;
    }
}
```

Call-by-Value

```
public class CallByValueTester3 {
    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;
    }
}
```

Output: Chris

Interfaces

- Specify abstract methods
  - method headers with no bodies

```
public interface EventHandler<T extends Event> {
      public void handle(T event);
}
```

- A class that implements **`EventHandler`** must define **`handle`**
  - else a syntax error
- So JavaFX knows to call your event handler's **`handle`**

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.

Interfaces/Abstract classes & Polymorphism

- Which of these (Interfaces, Abstract and Concrete classes):
  - can have instance variables?
  - can have static variables?
  - can have static final constants?
  - can have constructors?
  - can have abstract methods?
  - can have concrete methods?
  - can be constructed?
    - These are common interview questions.

Interfaces/Abstract classes & Polymorphism

- Which of these (Interfaces[i], Abstract[a] and Concrete[c] classes):
  - can have instance variables? [ac]
  - can have static variables? [ac]
  - can have static final constants? [iac]
  - can have constructors? [ac]
  - can have abstract methods? [ia]
  - can have concrete methods? [ac]
  - can be constructed? [c]
    - These are common interview questions.

static vs. non-static

- Static 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

StaticExample.java

```
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);
    }
}
```

StaticExample.java

```
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);
    }
}
```

Output: 1
3

`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?

`static` usage

- Can a **`static`** method:
  - directly call (without using a ".") a non-**`static`** method in the same class? No
  - directly call a **`static`** method in the same class? Yes
  - directly reference a non-**`static`** variable in the same class? No
  - directly reference a **`static`** variable in the same class? Yes
- Can a non-**`static`** method:
  - directly call (without using a ".") a non-**`static`** method in the same class? Yes
  - directly call a **`static`** method in the same class? Yes
  - directly reference a non-**`static`** variable in the same class? Yes
  - directly reference a **`static`** variable in the same class? Yes

```
public class Nothing {
 private int nada;                              // Errors?
 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;
}}
```

```java
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;
 }}
```