How’s PSA4?
A. I’m done!
B. I’m working on it
C. I’ve read it, but haven’t started coding
D. I haven’t even read it

Get it done ASAP! It’s a very good practice for next week’s test

Are all students people? Pondering life’s deep questions

Any issue with your participation grades?
First days missing cause you registered late?

Follow up on Piazza:
“Gradesource/iClicker issues”
1. In Java, when we define a subclass, it includes both what is defined in this new subclass and what was defined in the superclass. How do we call this concept?

A. Abstraction

B. Inheritance

C. Encapsulation

D. Composition

E. None of the above
2. What keyword do you use to define a subclass?

A. inherits  
B. extends  
C. super  
D. throws
3. What does 'overriding' mean?

A. To define new classes from an existing class.

B. To provide methods for accessing private variables of a class.

C. To define multiple methods with the same name but different signatures.

D. To provide a new implementation for a method in a subclass.
4. The class LeopardShark is a subclass of Shark, and the class Shark is a subclass of Animal.

True or false (and why): *Every Shark object is also a LeopardShark object.*

A. True, because the subclass LeopardShark might have properties and methods that the Shark object does not.

B. False, because the subclass LeopardShark might have properties and methods that the Shark object does not.
Next week test

Emphasis on PSA3 and PSA4 topics
Scope, Objects, references and memory models

Main scope

Nim object scope

Nim constructor scope
Static vs Non-static

We can have

• static methods and variables
  (“belong” to the Class, so we **don’t need** an object to access them)

• Instance (NON-static) methods and variables
  (“belong” to an Instance, so **we need** an object to access them)
Static vs Non-static

We can have

• static methods and variables
  (“belong” to the Class, so we **don’t need** an object to access them. They are usually actions/properties that “concern” to the whole class)

• Instance (NON-static) methods and variables
  (“belong” to an Instance, so **we need** an object to access them. They are usually actions/properties that “concern” to ONLY ONE object)
public class Alien{
    // FIELDS
    private static int numberOfAliens; //one per class!
    private int numLegs; //one per object!
    private int numArms; //one per object!

    // METHODS
    public Alien(int n){
        this.numArms = 2; this.numLegs = n;
        numberOfAliens ++ ;
    }
    public static boolean invasionFinished(){
        return numberOfAliens>5;
    }
    public int getNumLegs(){
        return this.numberOfLegs;
    }
    public static void main(String[] args){
        Random generator = new Random();
        while ( ! Alien.invasionFinished() ){
            Alien a = new Alien(generator.nextInt(10));
            System.out.println("Alien"+ Alien.numberOfAliens + ", Num legs:"+ a.getNumLegs());
        }
    }
public class Alien{
    // FIELDS
    private static int numberOfAliens;  // one per class!
    private int numLegs;              // one per object!
    private int numArms;             // one per object!

    // METHODS
    public Alien(int n){
        this.numArms = 2;
        this.numLegs = n;
        numberOfAliens ++ ;
    }

    public static boolean invasionFinished(){ return numberOfAliens>5; }
    public int getNumLegs(){ return this.numberOfLegs; }

    public static void main(String[] args){
        Random generator = new Random();
        while ( ! Alien.invasionFinished() ){
            Alien a = new Alien(generator.nextInt(10));
            System.out.println("Alien"+Alien.numberOfAliens + "
                , Num legs:"+a.getNumLegs());
        }
    }
}
Static methods can be called directly on the class name (can also be called on an object of the right class, but it’s strongly recommended NOT TO DO IT)

Non-static methods must be called on an object

<table>
<thead>
<tr>
<th>Instance method</th>
<th>STATIC method</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NON STATIC)</td>
<td>(there is no “instance” to use... there is no “calling object”)</td>
</tr>
<tr>
<td>✓ Access instance variable</td>
<td>X Access instance variable</td>
</tr>
<tr>
<td>✓ Access static variable</td>
<td>✓ Access static variable</td>
</tr>
<tr>
<td>✓ Call instance method</td>
<td>✓ Call static method</td>
</tr>
<tr>
<td>✓ Call static method</td>
<td>X Call instance method</td>
</tr>
</tbody>
</table>
Static vs Non-static

Static methods can be called directly on the class name (can also be called on an object of the right class, but it’s strongly recommended NOT TO DO IT)

Static methods can be called directly on the class name (can also be called on an object of the right class, but it’s strongly recommended NOT TO DO IT)

WE WILL NOT BE CREATING A LOT OF “STATIC” ELEMENTS, so don’t worry much about them. THE MOST IMPORTANT THING FOR YOU TO REMEMBER FOR NOW IS THAT THEY DON’T NEED AN OBJECT TO CALL THEM TO ACTUALLY RUN, so we will call them from the Class Name.

✓ Access instance variable
✓ Access static variable
✓ Call instance method
✓ Call static method

✗ Access instance variable
✓ Access static variable
✗ Call instance method
✓ Call static method
Beyond copy-and-paste

Java is an *object-oriented* programming language:

- **Classes**: user-defined datatypes
- **Objects**: variables of those types

The primary goal of OOP is to **create a good abstraction**

- one that models relationships accurately
- without forcing the user to keep track of more than necessary

There are *two* relationship types that Java can model . . .
Object-oriented programming

There are two relationship types that Java can model . . .

Two approaches to re-use code in Java:

Object reuse (what we have been doing ...)

```java
public class Make8BFrame {
    public static void main(String[] args) {
        JFrame myFrame = new JFrame("This is my window");
        myFrame.setSize(300, 400);
        myFrame.add(new JLabel("CSE 8B"));
        myFrame.setVisible(true);
    }
}
```

Object specialization/extension (what we are learning now ...)

```java
public class CSE8BFrame extends JFrame {
    public CSE8BFrame() {
        super("This is my window");
        this.setSize(300, 400);
        this.add(new JLabel("CSE 8B"));
    }
    public static void main(String[] args) {
        CSE8BFrame myFrame = new CSE8BFrame();
        myFrame.setVisible(true);
    }
}
```
Two approaches to re-use code in Java:

Object reuse (what we have been doing ...)

```java
public class Make8BFrame {
    public static void main(String[] args) {
        JFrame myFrame = new JFrame("This is my window");
        myFrame.setSize(300, 400);
        myFrame.add( new JLabel( "CSE 8B" ));
        myFrame.setVisible(true);
    }
}
```

Why don’t we need to create a new JFrame object here?
A. We do. This code will cause an error.
B. We are not creating a window at all, so we don’t need a JFrame.
C. The CSE8BFrame constructor does create a new JFrame object.

SUPER INVOKES JFrame CONSTRUCTOR!!!
Inheritance Hierarchy

Base Class
Person

Derived Class
Student

Very Derived Class
CSEMajor

class Person {

class Student extends Person {

A Student is “Person” and something else …

A CSEMajor is “Student” and something else …

class CSEMajor extends Student {

Any further?
Inheritance Hierarchy

Person

Student

CSEMajor

Data

String name;

Methods

boolean isAsleep(int hr)
{
    return hr > 22 || hr < 7;
}
Inheritance Hierarchy

- **Person** (Base Class)
  - **Student** (Derived Class)
    - **CSEMajor** (Very Derived Class)

**Data**
- `String name;`
- `int units;`

**Methods**
- `boolean isAsleep(int hr)`
  
  ```java
  boolean isAsleep(int hr)
  {
      return hr > 22 || hr < 7;
  }
  ```
  
  **overriding the previous method**

- `boolean isAsleep(int hr)`
  
  ```java
  boolean isAsleep(int hr)
  {
      return hr > 2 && hr < 8;
  }
  ```
Inheritance Hierarchy

```
public class Person {
    public String name;
    public int units;
    public boolean isTutor;
}

public class Student extends Person {
    public boolean isAsleep(int hr) {
        return hr > 2 && hr < 8;
    }
}

public class CSEMajor extends Student {
    public boolean isAsleep(int hr) {
        return hr > 22 || hr < 7;
    }
}
```

??
Person p = new Person( "Sally" );

Think of a variable with a reference like a leash. Java thinks “On the other end of this leash, there has to be a Person”
Think of a variable with a reference like a leash. Java thinks “On the other end of this leash, there has to be a Person”
Will the line above cause an error?
A. Yes
B. No
Variable types vs. object types

Student \( s = \text{new Person("Sally")}; \)

Will the line above cause an error?

A. Yes
B. No
Will the line above cause an error?
A. Yes  
B. No

* Students are ALWAYS a person. (They come from Person class)
* Person objects are NOT always students.
  So they can’t “answer” all the questions a Student will (run methods that a Student will)

The check to verify that variable type and object type (appear to?) match is done at COMPILIE TIME
Variable types vs. object types

Declared type

Object type

Must (appear to) match at compile time

Is there an error in the code above?
A. No, it’s fine
B. Yes, the first line will cause an error
C. Yes, the second line will cause an error
D. Yes, both lines would cause an error

```java
String s = new String( "Hello" );
Pixel p = new String( "This is a Pixel" );
```
class Person
{
    protected String name; // data member - protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

What will print in the main code here?

A. "Now offline: Sally"
B. "Now online: Sally"
C. "Now offline: Person@15c61fb5"
D. "Now online: Person@15c61fb5"
class Person
{
    protected String name; // data member - protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student extends Person
{
    protected int units; // additional data member

    public Student( String name, int units ) {
        super(name);
        this.units = units;
    }

    public boolean isAsleep( int hr ) { // override
        return 2 < hr && 8 > hr;
    }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}

Student s;
s = new Student("Sally", 16);
s.status(1);

What will this code print?
A. Now online: Sally units: 16
B. Now offline: Sally units: 16
C. Now online: units: 16
D. Now online: units: 16
E. Other (or error)
class Person
{
    protected String name;  // data member – protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student extends Person
{
    protected int units;  // additional data member

    public Student( String name, int units ) {
        super(name);
        this.units = units;
    }

    public boolean isAsleep( int hr ) // override
    { return 2 < hr && 8 > hr; }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}
class Person
{
  protected String name;  // data member – protected

  public Person( String name ) { this.name = name; }
  public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
  public String toString() { return name; }

  public void status( int hr )
  {
    if ( this.isAsleep( hr ) )
      System.out.println( "Now offline: " + this );
    else
      System.out.println( "Now online: " + this );
  }
}

class Student extends Person
{
  protected int units;  // additional data member
  public Student( String name, int units ) { super(name); this.units = units; }

  public boolean isAsleep( int hr ) // override
  { return 2 < hr && 8 > hr; }

  public String toString()
  {
    String result = super.toString();
    return result + " units: " + units;
  }
}

Polymorphism

All Students are People. So it is OK to store a Student in a variable that refers to a Person, because you will never try to do something that a Student cannot do. This makes the compiler happy.

AT RUNTIME (Dynamic Binding), Java determines that the OBJECT is actually a Student, so it invokes the STUDENT version of the isAsleep method (using the Person’s status method, because there is no specific method with that name for Student).

in main:

Person p;
p = new Student( "Sally", 16 );
p.status( 1 );
### Variable types vs. object types

<table>
<thead>
<tr>
<th>Declared (variable, reference) type</th>
<th>Actual type of the Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person ( p ) ( = ) new Student( “Sally”, 16 );</td>
<td>new Student( “Sally”, 16 );</td>
</tr>
<tr>
<td>Student ( s ) ( = ) new Student( “Sally”, 16 );</td>
<td></td>
</tr>
<tr>
<td>( p ).status(1); // These two lines do</td>
<td></td>
</tr>
<tr>
<td>( s ).status(1); // EXACTLY the same thing</td>
<td></td>
</tr>
</tbody>
</table>

- **Declared type** and object type must (appear to) match at compile time.

- At compile time Java makes sure that the methods and fields exist by using the variable (reference) type.

- At runtime, Java uses the **ACTUAL type** of the object to determine which method to call (the variable type no longer matters).
class Person
{
    protected String name;  // data member – protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student extends Person
{
    protected int units;  // additional data member

    public Student( String name, int units ) {
        super(name);
        this.units = units;
    }

    public boolean isAsleep( int hr ) // override
    { return 2 < hr && 8 > hr; }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}

Student s;
s = new Person( "Sally" );
s.status( 1 );

What will this code print?
A. Now online: Sally units: 0  
B. Now offline: Sally  
C. This code has a compile error
D. This code has a run-time error
E. I don’t know

A Person object may not be able to do what we may request to a Student object.
class Person
{
    protected String name;  // data member - protected

    public Person( String name )  { this.name = name; }
    public boolean isAsleep( int hr )  { return 22 < hr || 7 > hr; }
    public String toString()  { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student
extends Person
{
    protected int units; // additional data member

    public Student( String name, int units )  {
        super(name);
        this.units = units;
    }

    public boolean isAsleep( int hr )  // override
    { return 2 < hr && 8 > hr; }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}

Student s;
s = new Person( "Sally" );
s.status( 1 );

in main:

What will this code print?
A. Now online: Sally units: 0
B. Now offline: Sally
C. This code has a compile error
D. This code has a run-time error
E. I don't know

Not every Person object is a Student. If you store a Person in a variable that is supposed to reference a Student, the compiler gets nervous (and gives you an error) because you might try to make that object do something that it cannot do. This is true EVEN IF you never actually ask it to do something it cannot do (as in this example).
class Person
{
    protected String name;  // data member – protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student extends Person
{
    protected int units;   // additional data member

    public Student( String name, int units ) { this.name = name; this.units = units; }

    public boolean isAsleep( int hr ) // override
    { return 2 < hr && 8 > hr; }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}

This code has an error. What is it?
A. The Person class has no empty constructor, which will cause an error in the Student constructor
B. The member variable name cannot be accessed in the constructor for Student
C. super.toString() in the Student’s toString method is undefined
D. The student class has no status method defined in it

If we don’t call to the superclass constructor, the default EMPTY (no parameters) is called automatically

in main:
Student s;
s = new Student( "Sally", 16 );
s.status( 1 );
class Person
{
    protected String name;  // data member – protected

    public Person( String name ) { this.name = name; }
    public boolean isAsleep( int hr ) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

class Student
{  
    protected int units;  // additional data member
    public Student( String name, int units )  {
        super(name);
        this.units = units;
    }

    public boolean isAsleep( int hr )  // override
    { return 2 < hr && 8 > hr; }

    public String toString()
    {
        String result = super.toString();
        return result + " units: " + units;
    }
}

in main:
Student s;
s = new Student( "Sally", 16 );
s.status( 1 );

The constructor of the superclass will automatically be called by the subclass’s constructor. If the superclass does not have a default constructor, you must explicitly invoke the super-class’s constructor with the right parameters.
Polymorphism

Sometimes an exact type is not known until **run-time:**
- The compiler will assume the object is of the **declared** type.

The **constructor** still determines the **actual** type of the Object.
- At run-time, Java will use the **actual type's** latest (**most-derived**) methods.

A: `Person p = new Student( "Sally", 16 );`
B: `System.out.println("p's name is " + p.name);`
C: `System.out.println( p.isAsleep( 24 ) );`
D: `p.status( 24 );`
E: `System.out.println("P is taking " + p.units);`

One of these lines of code will cause an error. Which one? (And how do you fix it)
Casting!

Sometimes an exact type is not known until **run-time:**
- The compiler will assume the object is of the **declared** type.

The **constructor** still determines the **actual** type of the Object.
- At run-time, Java will use the **actual type's** latest (most-derived) methods.

A: `Person p = new Student( "Sally", 16 );`
B: `System.out.println("p's name is " + p.name);`
C: `System.out.println( p.isAsleep( 24 ) );`
D: `p.status( 24 );`
E: `System.out.println("P is taking " + ((Student)p).units);`

Casting changes the type of the reference to the object to **make the compiler believe us** that the object can really do what we ask it.
Casting can be dangerous, so be sure you know what you’re doing!
Inheritance and Polymorphism summary

• At compile time (references matter):
  
  – The object must ALWAYS be the type of the reference that refers to it. If the reference type is Foo, then the object must ALWAYS be a Foo (including any subclass of Foo). *For example, a Student is ALWAYS a Person, but a Person is not always a Student.*
  
  – When an object is referenced by a variable, the **reference type** determines **what the compiler thinks** the object type is. E.g., you cannot reference a student’s units field through a Person reference, even if the underlying object is a Student. Similarly, the following will cause a compile error:

    ```java
    Person p = new Student( "Sally", 16 );  // OK!
    Student s = p;  // Error here: the compiler uses p to determine type of
    // RHS object
    ```

  – Casting allows you to change the type of a reference *at compile time*. It will cause a runtime error if the actual object is not the type you are trying to cast to

• At run time (objects matter):
  
  – Java uses the **actual type of the object** to determine which methods/variables to use. The type of the reference no longer matters at runtime.
  
  – If you cast an object to a type that it is not, you will get a ClassCastException
Next week . . .

Inheritance!

Person

Student

CSEMajor