**Static Variables and Methods:**
When are static variables and methods used?

Example of a static variable:

Example of a static (class) method:

Inheritance – a relationship defined between objects that share characteristics.

A subclass can itself be a *superclass* for another subclass, leading to an *inheritance hierarchy* of classes.

**Example:** The arrow designates the *is-a* relationship.

![Inheritance Diagram](image)

The inheritance relationship between a subclass and a superclass is specified in the declaration of the subclass, using the keyword *extends*. 
Example:

```java
public class Person {
    public String myName;
    public String myID;
    ...
}

public class Student extends Person {
    public final static int NUM_TESTS = 3;
    private int[] myTests;
    private String myGrade;
    //constructors
    public Student () {
        super();
        myTests = new int[NUM_TESTS];
        myGrade = "";
    }
    public Student (String name, int[]tests, String grade) {
        super (name);
        myTests = tests;
        myGrade = grade;
    }
    public void setGrade (String newGrade ) {
        myGrade = newGrade;
    }
    public computeGrade () {
        if myName.equals("")
            myGrade = "No grade";
        else if (getTestAverage() >=65)
            myGrade = "Pass";
        else myGrade = "fail";
    }
    public double getTestAverage() {
        //Calculates and returns the average of all tests
    }
}

public class UnderGrad extends Student {
    public Undergrad() {
        super();
    }
    public Undergrad (String name, int[]tests, String grade) {
        super (name, tests, grade);
    }
    public void computeGrade() {
        if (getTestAverage() >=70)
            setGrade ("Pass");
        else
            setGrade ("Fail");
    }
}

public class GradStudent extends Student {
    private int myGradID;
    //default constructor
    public GradStudent() {
        super();
        myGradID=0;
    }
    //constructor
    public GradStudent (String name, int[]tests, String Grade, int gradID) {
        super (name, tests, grade);
        myGradID = gradID;
    }
    public int getID() {
        return myGradID;
    }
    public void computeGrade () {
        super.computeGrade();
        if (getTestAverage() >90)
            setGrade ("Pass with Honors");
    }
}
```
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 must define its own constructors.
- A subclass cannot access the private members of its superclass.

Declaring Subclass Objects:
When a variable of a superclass 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 UnderGrad();
```

Since Student is not necessarily a GradStudent nor an UnderGrad, the following declarations are *not* valid:

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

Valid Declarations:
```
Student s = new Student("Brian Sea", new int[]{90, 94, 99}, "none");
Student g = new GradStudent("Kevin Carter", new int[]{87, 89, 90}, "none");
```

What happens when the following method calls are implemented?
```
s.computeGrade();
g.computeGrade();
u.computeGrade();
```

Polymorphic:
A method that has been overridden in at least one subclass.

**Polymorphism** – the mechanism of selecting the appropriate method for a particular object in a class hierarchy.

** With polymorphism, the programmer can deal in generalities and let the execution-time environment concern itself with the specifics.

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.
EXAMPLE:
public class StudentTest
{
   public static void computeAllGrades (Student [] studentList)
   {
      for (int j = 0; j<studentList.length; j++)
         if (studentList[j] !=null)
            studentList[j].computeGrade();
   }

   public static void main (String[] args)
   {
      Student[] stu =new Student [5];
      stu[0]=new Student (“Brian Sea”, new int[] {90, 94,99}, “none”);
      stu[2] = new GradStudent (“Vicki Tramel”, new int[] {85, 90, 92}, “none”, 1234”);
      computeAllGrades(stu);
   }
}

Type Rules for Polymorphic Method Calls
\textit{a.method(b)}

- For a declaration like \textit{Superclass a = new Subclass();}
The type of \textit{a} at compile time is \textit{Superclass}; at runtime it is \textit{Subclass}.
- At compile time, method must be found in the class of \textit{a}, that is, in \textit{Superclass}. (This is true whether the method is polymorphic or not.) If method cannot be found in the class of \textit{a}, you need to do an explicit cast on \textit{a} to its actual type.
- For a polymorphic method, at run time the actual type of \textit{a} is determined – \textit{Subclass} in this example – and method is selected from \textit{Subclass}. This could be an inherited method if there is no overriding method.
- The type of parameter \textit{b} is checked at compile time. You may need to do an explicit cast to the subclass type to make this correct.
Abstract Class:

- A class in which the programmer never intends to instantiate any objects.
- The purpose is to provide an appropriate superclass from which other classes may inherit interface.
- A class is made abstract by declaring it with keyword `abstract`.
- A class that contains at least one abstract method is an abstract class.
- Although objects cannot be instantiated from abstract superclass, we can declare references to abstract superclasses.

Example:
public abstract class Shape
{
    public double area() {return 0.0;}
    public double volume() {return 0.0;}
    public abstract String getName();
}

// Definition of class Point
public class Point extends Shape {
    protected int x, y; // coordinates of the Point

    // no-argument constructor
    public Point() { setPoint( 0, 0 ); }

    // constructor
    public Point( int a, int b ) { setPoint( a, b ); }

    // Set x and y coordinates of Point
    public void setPoint( int a, int b )
    { x = a;
      y = b;
    }

    // get x coordinate
    public int getX() { return x; }

    // get y coordinate
    public int getY() { return y; }

    // convert the point into a String representation
    public String toString()
    { return "[" + x + ", " + y + "]"; }

    // return the class name
    public String getName() { return "Point"; }
}

public class Circle extends Point
{
    protected double radius;

    public Circle()
    {
    }

    public Circle (double r, int x, int y)
    {
    }

    public void setRadius (double r)
    {
    }

    public double area()
    {
    }

    public String toString()
    {
    }

    public String to

**What is an Interface and why are they important?**
- A contract that says, “Every class that signs this contract will provide the methods defined in this interface.
- An interface does not provide implementation for the methods it defines.
- To declare an interface use the keyword `interface`.
- Multiple interfaces can be implemented by only one base class can be extended.
- An interface contains only methods, never variables.
- The methods in an interface are usually public.
- If more than one class implements an interface, its methods are polymorphic.
- A class can implement more than one interface.
- Interfaces can be organized in an inheritance hierarchy.

**INTERFACES VS ABSTRACT CLASSES**
- Use an abstract base class when you are creating an *is a* relationship but you do not ever want to instantiate the base class.
- Use an interface when you are not creating an *is a* relationship, but rather are stating *has these capabilities*.
- The abstract base class creates a set of types.
- An interface creates a contract, a set of promises.

**EXAMPLE INTERFACE:**
```java
public interface Colorable {
    public Color getColor();
    public void setColor(Color newColor);
}
```

**Important Interface that is a part of the AP Java subset:**
- `Comparable` - contains only one method, `compareTo`, which takes an object as a parameter and returns an integer.
  - Example: `if(obj1.compareTo(obj2) < 0)...`

**Action Listener Interface: (contains only one method!)**
- When the ActionListener interface, you agree to define the `ActionPerformed()` method, which must return `void` and takes a single parameter of type `ActionEvent`.

**MOUSE EVENTS and MOUSE MOTION EVENTS:**
**Methods of the `MouseListener` Interface**
- `mouseEntered(MouseEvent e)` - //Raised each time mouse enters a component
- `mouseExited(MouseEvent e)` - //Raised each time mouse exits a component
- `mousePressed(MouseEvent e)` - //Raised each time mouse button is pressed
- `mouseReleased(MouseEvent e)` - //Raised each time mouse button is released
- ` mouseClicked(MouseEvent e)` - //Raised each time mouse is clicked

****All five methods must be implemented even if you only intend to use one!****

**Methods of the `MouseMotion` Interface**
- `mouseDragged(MouseEvent e)` - //Invoked when a mouse button is pressed on a component and then dragged.
- `mouseMoved(MouseEvent e)` - //Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

**Example Heading:**
```java
public class MouseClick extends JApplet implements MouseListener
    //Promise is made to implement all the methods of the MouseListener interface
```
Throwing and Catching Exceptions:

Once an exception is thrown, processing stops and the JVM looks for a `catch` block to handle the exception. The first place the JVM looks is within the method where the exception was thrown. If none is found, the JVM unwinds the stack and looks for a catch statement in the calling method.

Try/Catch blocks:

```java
try {
    quotient = doDivide (first, second);
    System.out.println ("Quotient = "+quotient);
} catch (ArithmeticException e) {
    JOptionPane.showMessageDialog (null, "Arithmetic exception caught!", "Arithmetic Error", JOptionPane.ERROR_MESSAGE);
}
```

NOTE: Division by 0 exceptions are thrown only for integer division. If doubles are used for the values, no exception will be thrown for division by 0.0. So you might choose to throw your own exception.

```java
if (divisor == 0)
    throw new ArithmeticException();
```

```java
or
```
```java
    if (divisor == 0)
        throw new ArithmeticException ("Tried to divide "+dividend+" by zero!");
```

EXCEPTION CLASSIFICATIONS

- All exceptions ultimately derive from the Throwable class. The intermediate subclasses of Throwable are Error and Exception.
- The Java framework calls Errors and RunTimeExceptions “unchecked exceptions,” and it calls IOExceptions “checked exceptions.”
- Checked exceptions are checked by the compiler to ensure that the exception is caught or specified.

Exceptions to know:

- NullPointerException:
- ArrayIndexOutOfBoundsException:
- ArithmeticException:

The ClassCastException:
The ClassCastException is a run-time exception thrown to signal an attempt to cast an object to a class of which it is not an instance.

```java
Student u = new UnderGrad();
System.out.println ((String) u);
```

Example: Write a try-catch block that checks that valid data has been entered before converting a string to an integer.