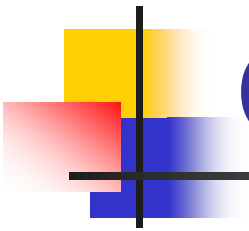
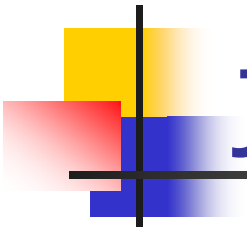


# CS 11 java track: lecture 3

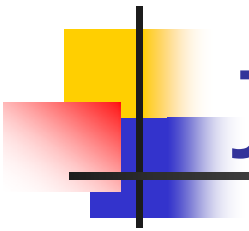
- This week:
  - documentation (javadoc)
  - exception handling
  - more on object-oriented programming (OOP)
    - inheritance and polymorphism
    - abstract classes and interfaces
  - graphical user interfaces (GUIs)



# javadoc (1)

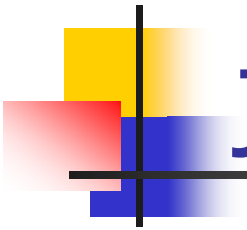
- three ways to write comments:

```
// this way (to end of line)
/* this way (to closing star-slash) */
/**
 * this way (javadoc style)
 */
```



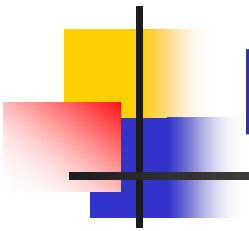
# javadoc (2)

```
/**
 * This is a javadoc comment for a method.  Note
 * the two asterisks after the starting "/".
 *
 * @param foo  This is a comment for the argument "foo".
 * @return     This describes the return value.
 */
public int myMethod(double foo) {
    // code goes here
}
```



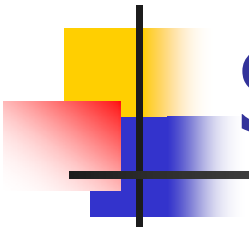
# javadoc (3)

- Javadoc comment tags:
  - @param – describes an argument
  - @return – describes the return value
  - @throws – describes an exception that can be thrown
- Can use the "javadoc" tool to generate HTML documentation from classes this way
  - but we won't (you can try if you want)



# Exception handling

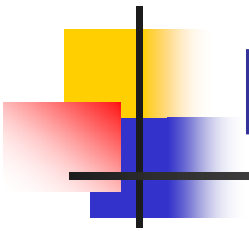
- Purpose of exceptions
  - deal with "exceptional situations"
    - errors (*e.g.* divide by zero)
    - rare occurrences (*e.g.* end of file when reading)
  - normal flow of control won't work
  - don't want to clutter the code with lots of tests



# Syntax

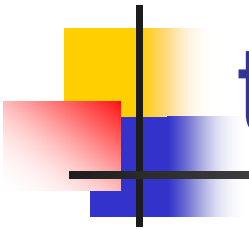
- syntax of exceptions

```
try {
    // some code that might throw an exception
} catch (IOException e) {
    // code to handle IO exceptions
} catch (Exception e) {
    // code to handle all other exceptions
} finally {
    // Do this whether or not an exception
    // was thrown.
}
```



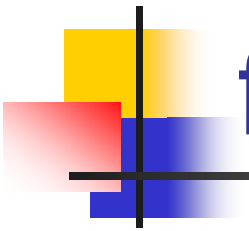
# More about exception syntax

- can have
  - try and one catch
  - try and several catches
  - try and finally
  - try, catch, finally
  - try, several catches, finally
- most of the code will be in the try block



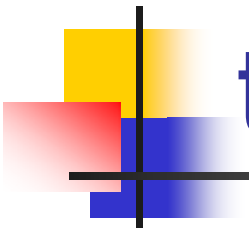
# try/catch/finally

- meaning of try/catch/finally:
  - try to execute the code in this block
  - if an exception is thrown, look at the catch blocks
    - if one of them identifies the type of the exception, execute that block; else go to the next block
    - type of exception == same class or a subclass
  - if there is a finally block, execute it after the try block and any catch blocks
    - even if a new exception is thrown



# finally

- finally blocks are rare
- usually used to "clean up" before exiting a method
  - *e.g.* close a file that was opened in a try block
- even if a new exception gets thrown in the try block or in a catch block, finally block still gets executed

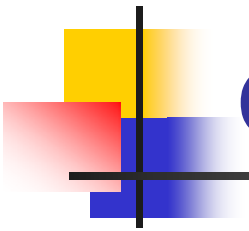


# throw

- if you want to *generate* an exception you must throw it

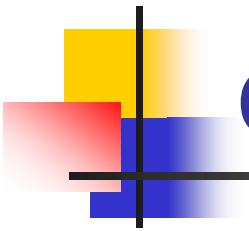
```
if (something_bad_happens()) {
    throw new Exception("BAD!");
}
```

- exceptions are classes (need new)
- exception constructors can take arguments
  - usually just pass a string with a message



# checked exceptions (1)

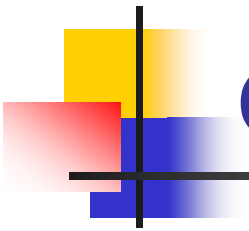
- most exceptions in java are *checked exceptions*
- this means that code that can throw an exception must either
  - be in a try/catch block
  - be in a method that declares that it throws that exception



# checked exceptions (2)

- say your method can throw an IOException

```
public void myMethod() {
     try {
          // code that may throw IOException
     }
     catch (IOException e) {
          // handle the exception
     }
}
```

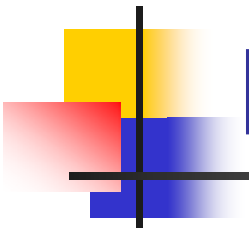


# checked exceptions (3)

- alternatively...

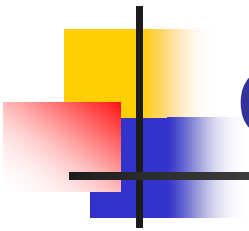
```
public void myMethod() throws IOException {
    // code that may throw IOException
}
```

- now don't need the try/catch blocks
- could just declare that all methods throw Exception (but that's lame)



# RuntimeExceptions

- some exceptions are so unpredictable or can happen in so many places that having to check for them would be unbearably tedious *e.g.*
  - divide by zero (ArithmeticException)
  - array bounds violations (ArrayIndexOutOfBoundsException)
- these are RuntimeExceptions
  - don't have to be caught or declared
  - can still catch or declare if you want

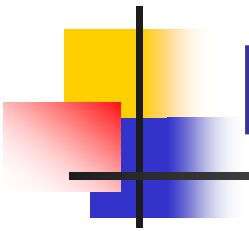


# catch-all exception handlers

- can catch ALL exceptions this way:

```
try {
    // code that can throw exceptions
} catch (Exception e) {
    // handle all exceptions e.g.
    System.out.println(e);
}
```

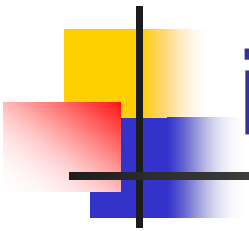
- NOT good practice
- should catch specific exceptions; use catch-all exceptions only after catching specific exceptions if at all



# back to OOP -- inheritance

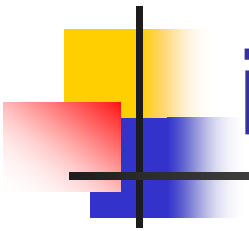
- consider:

```
// base class
public class Vehicle {
    private double accel;
    protected int numberOfWheels;
    public Point2d getPosition() { /* ... */ }
    public float getVelocity() { /* ... */ }
}
```



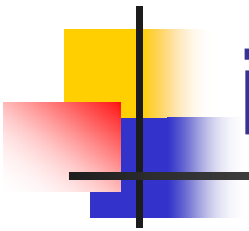
# inheritance (2)

```
// subclass
public class MotorVehicle extends Vehicle {
    private double fuel;
    public float getFuelRemaining() { /* ... */ }
    public String getLicensePlate() { /* ... */ }
    // override base class method:
    public float getVelocity() { /* ... */ }
}
```



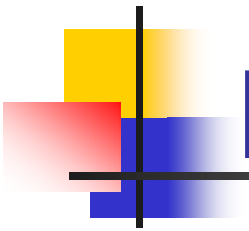
# inheritance (3)

- subclass has an "is-a" relationship to base class (also called "superclass")
- **`MotorVehicle`** "is-a" **`Vehicle`**
  - we say it "inherits from" **`Vehicle`**
- instance of MotorVehicle can call **`getPosition()`**, **`getVelocity()`** (superclass methods) – gets them "for free"
- can also call **`getFuelRemaining()`** or **`getLicensePlate()`** (new methods)



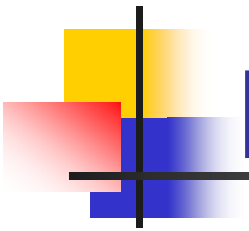
# inheritance (4)

- can override superclass methods in subclass
  - *e.g.* new **`getVelocity()`** method in **`MotorVehicle`**
- can also add new fields or new methods
- **`private`** fields and methods not accessible to subclasses
- **`protected`** fields and methods are accessible to subclasses



# polymorphism (1)

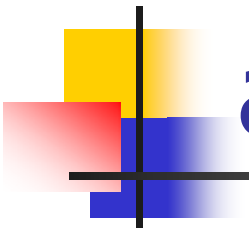
- the "big idea" in object-oriented programming
- basic idea: the appropriate method is chosen based on the object that the method is called on
- vague enough for you?
- example...



# polymorphism (2)

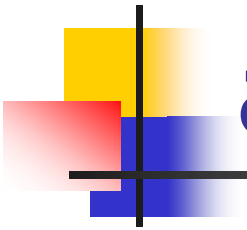
```
Vehicle v = new Vehicle();
MotorVehicle mv = new MotorVehicle();
mv = v;  // DOES NOT COMPILE!
         // Not all Vehicles are MotorVehicles!
v = mv;  // OK: all MotorVehicles are Vehicles.
         // DOES NOT "convert" mv to a Vehicle.
System.out.println(v.getVelocity());
// Which getVelocity() method gets called?

// ANSWER: the MotorVehicle getVelocity() method.
// v is still a MotorVehicle.
```



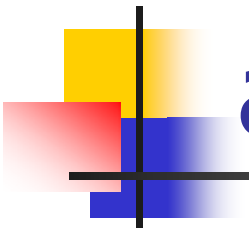
# abstract classes (1)

- a class with some methods unimplemented is an abstract class
- such a class cannot be instantiated
- can be used to make subclasses that can be instantiated ("concrete" subclasses)
- methods declared with the keyword `abstract` aren't defined
- class also has to have the `abstract` keyword



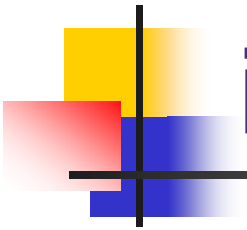
# abstract classes (2)

```
abstract class Shape {
    Point2d center;
    abstract void draw();
}
class Circle extends Shape {
    void draw() { /* draw a Circle */ }
}
class Rectangle extends Shape {
    void draw() { /* draw a Rectangle */ }
}
```



# abstract classes (3)

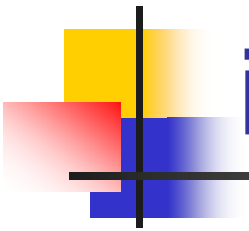
```
Shape s = new Shape(); // COMPILE ERROR
Shape s = new Rectangle();  // OK
s.draw();  // draws a Rectangle
```



# interfaces (1)

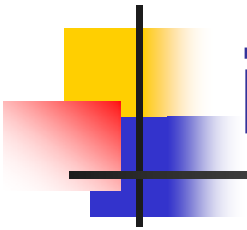
- an abstract class with no fields and *only* abstract methods can also be called an interface

```
interface Drawable {
    public void draw();
}
```



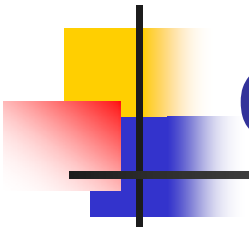
# interfaces (2)

- why bother having interfaces?
- why not just have abstract classes?
- answer: a class can extend only one class (abstract or not), but can implement any number of interfaces
- this is how java handles what would require multiple inheritance in *e.g.* C++
- interfaces specify behavior ONLY – *not* implementation



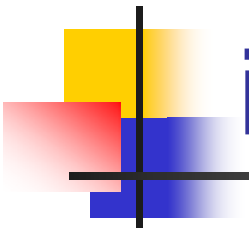
# interfaces (3)

- subtle point:
- if a class is a subclass of a class which implements an interface Foo, then the subclass also implements interface Foo (you don't have to re-declare it)



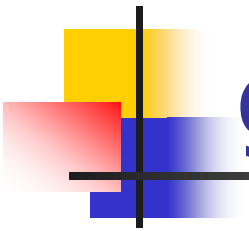
# default constructors

- every class has a no-argument constructor
- if you don't explicitly supply one, one is created for you that does nothing
  - this is the "default constructor"



# inheritance and constructors

- `super(arg1, arg2, ...);`
  - calls the superclass constructor with arguments arg1, arg2, etc.
  - can only be called inside a subclass constructor
  - if called, must be the first statement in the constructor
  - if not called, the default superclass constructor is automatically called
  - used to make initialization easier
    - only need to initialize new fields



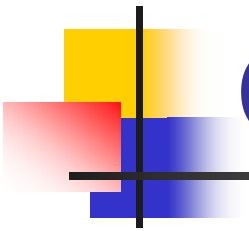
# graphical user interfaces (GUIs)

- GUIs are composed of buttons, frames, windows, menus, sliders, etc.
- usual way of interfacing with interactive programs
- in java, two main sets of libraries for GUIs:
  - AWT (Abstract Windowing Toolkit)
  - Swing (more advanced and nicer-looking)

# GUI inheritance hierarchy

- long inheritance tree for GUI classes:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.AbstractButton
                          |
                          +--javax.swing.JButton
```

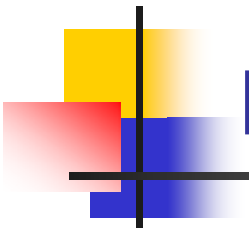


# GUI objects

- first do this:

```
import javax.swing.*;  // all of the Swing classes
import java.awt.*;     // fundamental GUI classes
```

- GUI objects:
  - JFrame: a window
  - Container: window, not incl. menus, titlebar, etc.
  - JButton: a button
  - and lots more...



# next week

- arrays
- listener classes
- inner classes
- anonymous inner classes