CS1020 Lecture Note #3: **Object Oriented Programming Final Part**

More OOP concepts

Lecture Note #3: **OOP Part 2**

- **Objectives:**
  - Introduce more predefined Java classes
  - Introduce wrapper class
  - Introduce vector class

- **References:**
  - Wrapper classes:
    - Chapter 1, Section 1.1, pages 29 to 30
  - **Object** class:
    - Chapter 1, Section 1.5, pages 56 to 58

Lecture Overview

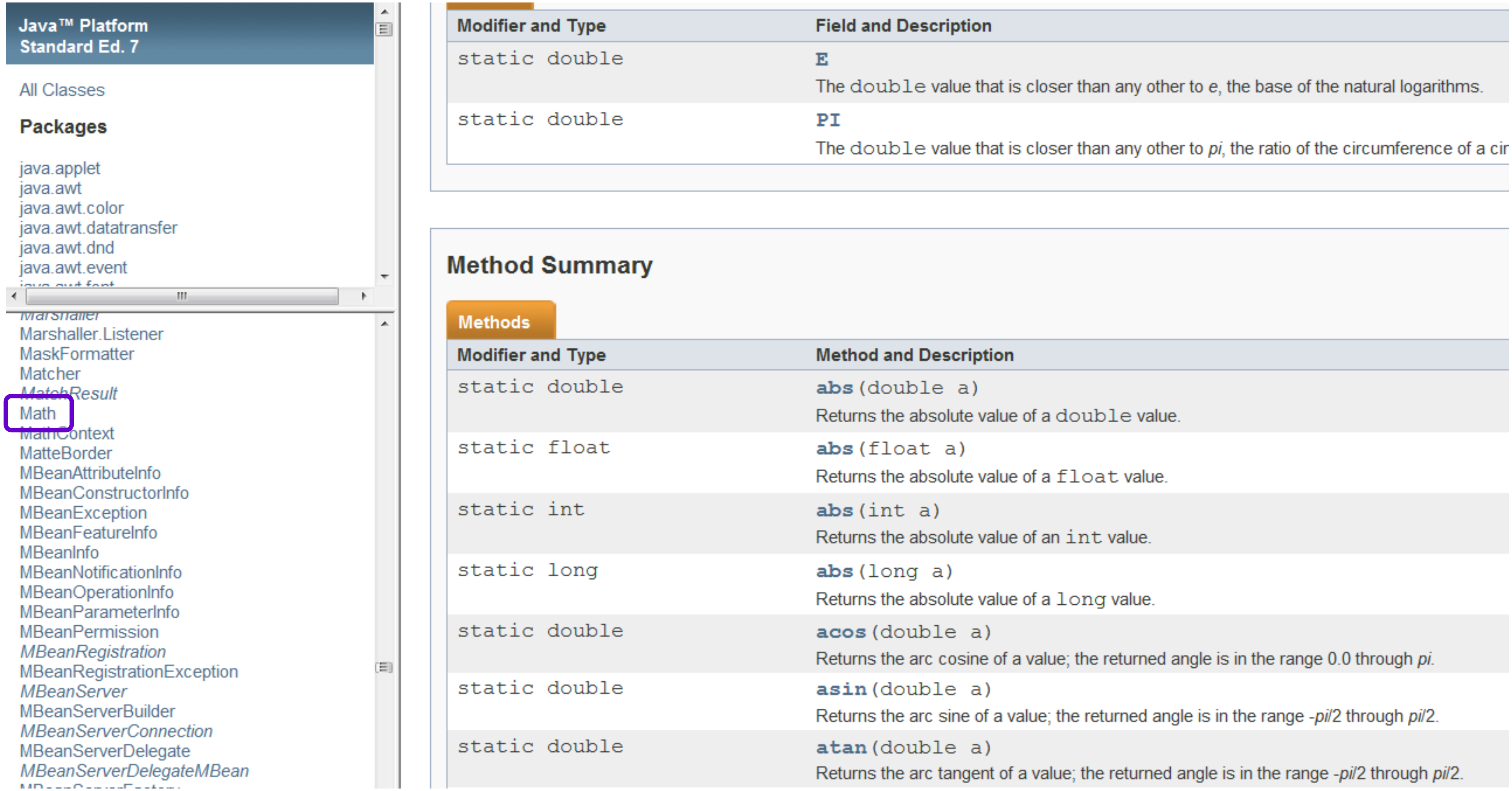
1. More Predefined Classes
   - 1.1 Math (introducing method overloading, static/class methods and class members)
   - 1.2 Wrapper Classes for Primitive Data Types
2. Using Wrapper class
3. Vector Class

1. More Predefined Java Classes

- We introduced the **`String`** class in the last lecture.
- There are more predefined Java classes:
  - **`Math`** class
  - Wrapper classes
  - Vector class
- Have you familiarised yourself with the Java API documentation?

1.1 The **Math** class

- From the API documentation:



Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font

Marshaller
Marshaller.Listener
MaskFormatter
Matcher
MatchResult
Math
MathContext
MatteBorder
MBeanAttributeInfo
MBeanConstructorInfo
MBeanException
MBeanFeatureInfo
MBeanInfo
MBeanNotificationInfo
MBeanOperationInfo
MBeanParameterInfo
MBeanPermission
MBeanRegistration
MBeanRegistrationException
MBeanServer
MBeanServerBuilder
MBeanServerConnection
MBeanServerDelegate
MBeanServerDelegateMBean
MBeanServerFactory

Modifier and Type	Field and Description
`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a cir

Method Summary

Methods

Modifier and Type	Method and Description
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.
`static float`	**`abs`**`(float a)` Returns the absolute value of a `float` value.
`static int`	**`abs`**`(int a)` Returns the absolute value of an `int` value.
`static long`	**`abs`**`(long a)` Returns the absolute value of a `long` value.
`static double`	**`acos`**`(double a)` Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
`static double`	**`asin`**`(double a)` Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
`static double`	**`atan`**`(double a)` Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

1.1 The **Math** class

- Package: java.lang.Math (default)
- Some useful **`Math`** methods:
  - `abs()`
  - `ceil()`
  - `floor()`
  - `max()`
  - `min()`
  - `pow()`
  - `random()`
  - `sqrt()`
- Note the presence of many overloaded methods
  - **`abs(double a)`**, **`abs(float a)`**, **`abs(int a)`**, etc.

1.1 Method Overloading

- Overloading methods – 2 or more methods within the same class with the same name but different parameters
  - Very useful feature of Java
- Example: **`abs()`** method in Math class

```
public static int abs(int num)
```

Returns the absolute value of **`num`**.

```
public static double abs(double num)
```

Returns the absolute value of **`num`**.

- Hence, you may use **`abs()`** like this:

```
int num = Math.abs(-40);
double x = Math.abs(-3.7);
```

1.1 Method Overloading: Quiz (1/2)

- Given the following overloaded methods:

```
public static void f(int a, int b) {
  System.out.println(a + b);
}

public static void f(double a, double b) {
  System.out.println(a - b);
}
```

- What are the outputs of the following codes?

```
f(3, 6);
f(3.0, 6.0);
f(3, 6.0);
```

1.1 Method Overloading: Quiz (2/2)

- How about this?

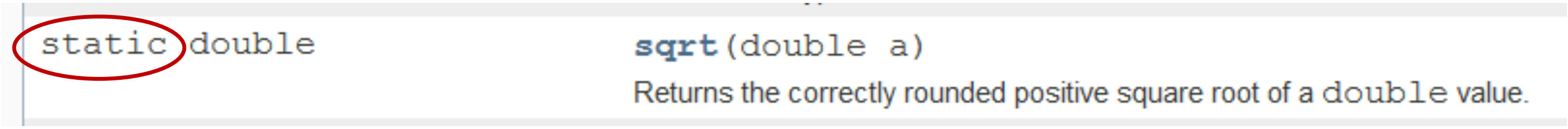
```
public static void g(int a, double b) {
  System.out.println(a + b);
}
public static void g(double a, int b) {
  System.out.println(a - b);
}
```

- What is the output of the following code?

```
g(3, 6);
```

1.1 Static/Class methods

- Note that in the definition of every **`Math`** method, the keyword "**`static`**" appears.

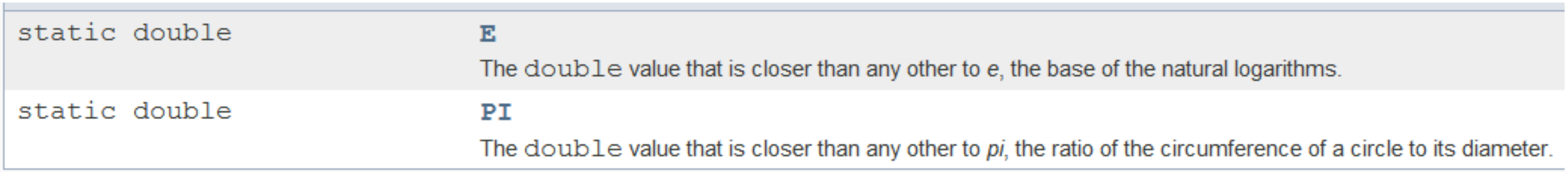


```
static double                sqrt(double a)
```
Returns the correctly rounded positive square root of a `double` value.

- Such a method is called a static method (or class method).
- This means that no object (instance) of the **`Math`** class is required to use the method.
- Any **`Math`** method is called by preceding its name with the name of the class:
  - Example: **`Math.sqrt(area)`**

1.1 Class attributes

- The **Math** class also has two class attributes



static double	**E** The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	**PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

- A class attribute (or class member) is associated with the class, not the individual instances (objects). Every instance of a class shares a class attribute.
- How to use it?
  - Example: **Math.PI**

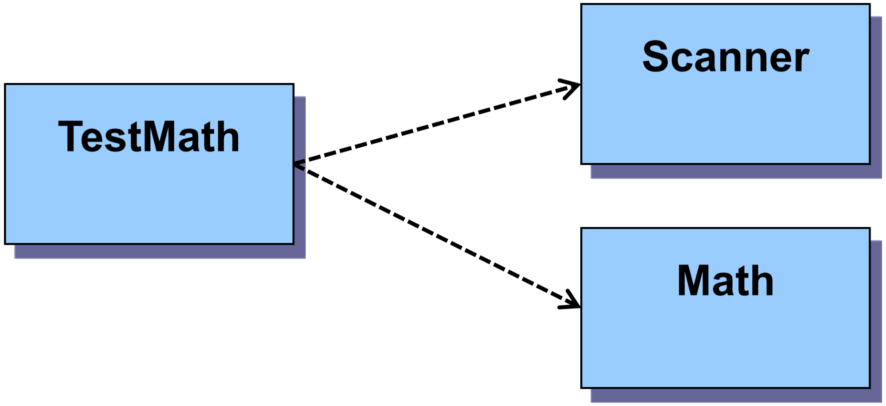
1.1 The **Math** class: Sample usage

TestMath.java

```
import java.util.*;

// To find the area of the largest circle inscribed
// inside a square, given the area of the square.

class TestMath {

  public static void main(String[] args) {

    double areaSquare, radius, areaCircle;

    Scanner myScanner = new Scanner(System.in);

    System.out.print("Enter area of a square: ");
    areaSquare = myScanner.nextDouble();

    radius = Math.sqrt(areaSquare) / 2;
    areaCircle = Math.PI * Math.pow(radius,2);

    System.out.printf("Area of circle = %.4f\n", areaCircle);
  }
}
```

1.1 Dependency Relationship

- The dependency relationship in **`TestMath.java`**



- **`TestMath`** class depends on both **`Scanner`** and **`Math`** classes.

1.2 Wrapper Classes: Motivation

- Other than the primitive data types, all other data in Java are in object form
  - Accessed through an object reference
  - Provide a number of methods and/or attributes

- The primitive data types are exceptions to the norm mainly due to efficiency considerations:
  - Object representation takes up more memory space
  - Object access is slower

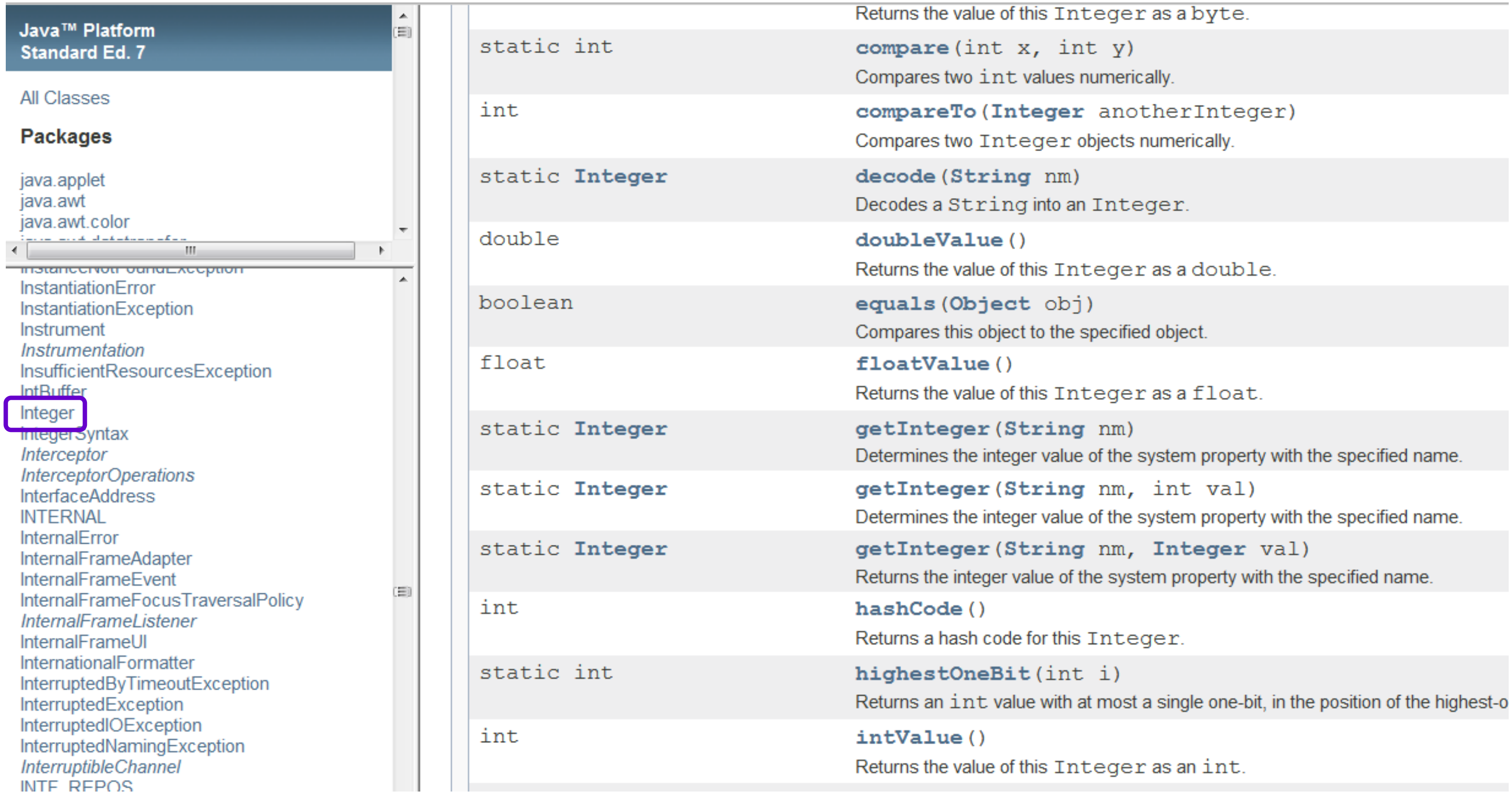
1.2 Wrapper Classes: Motivation

- There are situations where we need an object representation of the primitive data types:
  - Java provides a number of wrapper classes for this purpose

Primitive Data Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`

1.2 Wrapper Classes Example: Integer

- From the API documentation:
  - Package: java.lang.Integer (default)



Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer

InstanceNotFoundException
InstantiationError
InstantiationException
Instrument
Instrumentation
InsufficientResourcesException
IntBuffer
Integer
IntegerSyntax
Interceptor
InterceptorOperations
InterfaceAddress
INTERNAL
InternalError
InternalFrameAdapter
InternalFrameEvent
InternalFrameFocusTraversalPolicy
InternalFrameListener
InternalFrameUI
InternationalFormatter
InterruptedByTimeoutException
InterruptedException
InterruptedIOException
InterruptedNamingException
InterruptibleChannel
INTF_REPOS

	Returns the value of this `Integer` as a `byte`.
`static int`	**`compare`**`(int x, int y)` Compares two `int` values numerically.
`int`	**`compareTo`**`(`**`Integer`**` anotherInteger)` Compares two `Integer` objects numerically.
`static` **`Integer`**	**`decode`**`(`**`String`**` nm)` Decodes a `String` into an `Integer`.
`double`	**`doubleValue`**`()` Returns the value of this `Integer` as a `double`.
`boolean`	**`equals`**`(`**`Object`**` obj)` Compares this object to the specified object.
`float`	**`floatValue`**`()` Returns the value of this `Integer` as a `float`.
`static` **`Integer`**	**`getInteger`**`(`**`String`**` nm)` Determines the integer value of the system property with the specified name.
`static` **`Integer`**	**`getInteger`**`(`**`String`**` nm, int val)` Determines the integer value of the system property with the specified name.
`static` **`Integer`**	**`getInteger`**`(`**`String`**` nm, `**`Integer`**` val)` Returns the integer value of the system property with the specified name.
`int`	**`hashCode`**`()` Returns a hash code for this `Integer`.
`static int`	**`highestOneBit`**`(int i)` Returns an `int` value with at most a single one-bit, in the position of the highest-o
`int`	**`intValue`**`()` Returns the value of this `Integer` as an `int`.

1.2 Wrapper Classes: Sample usage

TestWrapper.java

```java
class TestWrapper {

  public static void main(String[] args) {
    Integer intRefA, intRefB;
    int intPrimitive;

    intRefA = new Integer(4);
    intRefB = 4;

    if (intRefA == intRefB)
      System.out.println("Both refer to the same object");

    if (intRefA.equals(intRefB))
      System.out.println("Both contain the same value");

    intPrimitive = intRefA.intValue();
    intPrimitive = intRefB;
  }
}
```

`intRefA` and `intRefB` are references!

Object Instantiation

Alternative: Known as **auto boxing**

False

True

Conversion to primitive type

Alternative: Known as **auto unboxing**

1.3 The "`Object`" class

- In Java, **all classes** are descendant of a predefined class called "**`Object`**"
  - **`Object`** class specifies some basic behaviors common to all objects
  - Any methods that works with **`Object`** reference will work on **object of any class**
  - Methods defined in the **`Object`** class are inherited in all classes
  - Two inherited **`Object`** methods are
    - **`toString()`** method
    - **`equals()`** method
  - However, these inherited methods usually don’t work (!) because they are not customised

2 Recapitulation

Let’s consolidate what we have learned so far

2.1 User-defined **`Ball`** class

- In this this section we will create the **`Ball`** class to illustrate concepts covered:
  - Class and instance attributes
  - Overloaded constructors
  - Assessors and mutators
  - “this” keyword
- We will use **`BallV2`** class to illustrate
  - Overriding methods: **`toString()`** and **`equals()`**

2.1 Ball class (1/2)

Ball.java

```
// Version 1: basic
class Ball {
   /************** Data members **********************/
   // Assuming the inventory code for Ball is 12345
   private static int code = 12345;

   private String colour;
   private double radius;

   /************** Constructors **********************/
   public Ball() {
        setColour("yellow");  // default colour
        setRadius(10.0);      // default radius

        // the statements below work too
        // colour = new String("yellow");
        // radius = 10.0;
   }

   public Ball(String newColour, double newRadius) {
        setColour(newColour);
        setRadius(newRadius);

        // the statements below work too
        // colour = newColour;
        // radius = newRadius;
   }
```

Class attribute, shared by all objects of this class.

Instance attributes, owned by each instance (object).

Overloaded constructors

Could replace these 2 statements with:

```
   this("yellow", 10.0);
```

2.1 **Ball** class (2/2)

Ball.java

```
   /**************** Accessors ***********************/
   public static int getCode() { return code; }

   public String getColour() { return colour; }

   public double getRadius() { return radius;}

   /**************** Mutators ***********************/
   // Why is "this" necessary here? How can the methods
   // be rewritten such that "this" becomes unnecessary?

   public static void setCode(int code) {
        Ball.code = code;
   }

   public void setColour(String colour) {
        this.colour = colour;
   }

   public void setRadius(double radius) {
        this.radius = radius;
   }

}
```

2.1 TestBall program (1/2)

TestBall.java

```
import java.util.*;
class TestBall {
   public static void main(String[] args) {

       Scanner scanner = new Scanner(System.in);
       int inputCode;
       String inputColour;
       double inputRadius;

       // Create a default Ball object
       Ball myBall = new Ball();

       // What is myBall's code at this point?
       System.out.println("myBall's default code: " + myBall.getCode());

       // Read inputs from user
       System.out.print("Enter code: ");
       inputCode = scanner.nextInt();
       System.out.print("Enter colour: ");
       inputColour = scanner.next(); // What's difference between next()
                                     // and nextLine()?
       System.out.print("Enter radius: ");
       inputRadius = scanner.nextDouble();
```

2.1 TestBall program (2/2)

TestBall.java

```
      // Set the code, colour and radius of this Ball object
      // Note that we may call a static method on an instance:
      //    myBall.setCode(inputCode);
      // but this will be as good as the statement below
      Ball.setCode(inputCode);
      myBall.setColour(inputColour);
      myBall.setRadius(inputRadius);

      // Display the contents of the Ball object
      // Note also that we may call:
      //    myBall.getCode();
      // but again, it is as good as the statement below
      System.out.println("Code is " + Ball.getCode());
      System.out.println("Colour is " + myBall.getColour());
      System.out.println("Radius is " + myBall.getRadius());

      // What output do you get for the following statement?
      // (We will learn how to deal with it now.)
      System.out.println("Ball's contents are " + myBall);
   }
}
```

2.2 Overriding methods

- The **`Ball`** class inherited the **`toString()`** and **`equals()`** methods from **`Object`** class.
- The **`toString()`** is automatically invoked when an instance is printed:

Equivalent

```
System.out.println(myBall);
System.out.println(myBall.toString());
```

- Need to customise **`toString()`** and **`equals()`** to override the inherited ones
- We will create **`BallV2`** class and add the new codes.

2.2 Overriding methods: toString() & equals()

BallV2.java

```
// Version 2
class BallV2 {
   // omitted attributes, constructors, assessors, mutators

   /***************** Overriding methods ******************/
   // Overriding toString() method
   public String toString() {
     return "[" + getColour() + ", " + getRadius() + "]";
   }

   // Overriding equals() method
   public boolean equals(Object obj) {
     if (obj instanceof BallV2) {
       BallV2 ball = (BallV2) obj;
       return this.getColour().equals(ball.getColour()) &&
              this.getRadius() == ball.getRadius();
     }
     else
       return false;
   }
}
```

2.2 Overriding methods: TestBallV2 (1/2)

TestBallV2.java

```java
import java.util.*;

class TestBallV2 {

   // This method reads ball's input data from user, creates
   // a ball object, and returns it to the caller.
   public static BallV2 readBall(Scanner sc) {

      System.out.print("Enter colour: ");
      String inputColour = sc.next();
      System.out.print("Enter radius: ");
      double inputRadius = sc.nextDouble();

      // Create a BallV2 object using the alternative constructor
      return new BallV2(inputColour, inputRadius);
   }

// Code continues to next slide
```

2.2 Overriding methods: TestBallV2 (2/2)

TestBallV2.java

```
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      // Read ball's input and create a ball object
      BallV2 myBall1 = readBall(scanner);
      System.out.println();
      // Read another ball's input and create a ball object
      BallV2 myBall2 = readBall(scanner);
      System.out.println();

      // Testing toString() method
      // How would output be like if there's no toString() in BallV2?
      System.out.println("1st ball: " + myBall1);
      System.out.println("2nd ball: " + myBall2);

      // Testing ==
      System.out.println("myBall1 == myBall2 is " +
                        (myBall1 == myBall2));

      // Testing equals() method
      System.out.println("myBall1.equals(myBall2) is " +
                         myBall1.equals(myBall2));
   }
}
```

3 Vector

Dynamic-size array

4.1. Vector: **Motivation**

- Array has one major drawback:
  - Once initialized, the array size is fixed
  - Reconstruction is required if the array size changes
  - Note that Java has an Array class.
    - Check API documentation and explore it yourself
- Java offers a **Vector** class to provide:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods
- Use array if the size is fixed, use Vector if the size may change.

4.2. Vector: **API documentation** (1/2)

PACKAGE

```
import java.util.Vector;
```

SYNTAX

```
//Declaration of a Vector reference
Vector<E> myVector;

//Initialize a empty Vector object
myVector = new Vector<E>;
```

Commonly Used Method Summary	
`boolean`	***`isEmpty`*** `()` Tests if this vector has no components.
`int`	***`size`*** `()` Returns the number of components in this vector .

4.2. Vector: **API documentation** (2/2)

Commonly Used Method Summary (continued)	
`boolean`	***add***`(E o)` Appends the specified element to the end of this Vector.
`void`	***add***`(int index, E element)` elements behind will be moved to the next position Inserts the specified element at the specified position in this Vector.
`E`	***remove***`(int index)` Removes the element at the specified position in this Vector.
`boolean`	***remove***`(Object o)` false is not found Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
`E`	***get***`(int index)` Returns the element at the specified position in this Vector.
`int`	***indexOf***`(Object elem)` -1 if not found Searches for the first occurence of the given argument, testing for equality using the equals method .
`boolean`	***contains***`(Object elem)` Tests if the specified object is a component in this vector.

4.3 Vector: Example

TestVector.java

```
import java.util.Vector;

class TestVector {

   public static void main(String[] args) {

      Vector<String> courses;

      courses = new Vector<String>();

      courses.add("CS1020");
      courses.add(0, "CS1010");
      courses.add("CS2010");

      System.out.println(courses);
      System.out.println("At index 0: " + courses.get(0));

      if (courses.contains("CS1020"))
         System.out.println("CS1020 is in vector");

      courses.remove("CS1020");
      for (String c: courses)
         System.out.println(c);
   }
}
```

Vector class has a nice **toString()** method that prints all elements

The enhanced for-loop is applicable to vector object too!