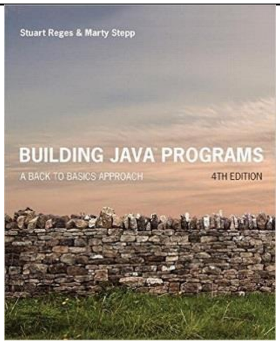
Important announcements

- Midterm examination scheduled on March7, 2018
- Study guides and sample exam will be available on Feb 26, 2018

Java Classes

Reading Assignment: Read Chapters 8.

Building Java Programs (Stuart Reges and Marty Stepp)



Objects and Classes

- Classes are concepts, objects are instances that embody those concepts.

objects
class
car

- A **class** captures the common properties of the objects instantiated from it.
- A class characterizes the common behaviors of all the objects that are its instances.

A simplified view of classes, objects, and methods

1. A class can be considered as a *user defined data type*.
2. An object is an instance of a class, i.e. a *value* of an user defined data type.
3. Methods are *functions/procedures* that can be applied to objects, i.e. *operators* can be applied to data values.

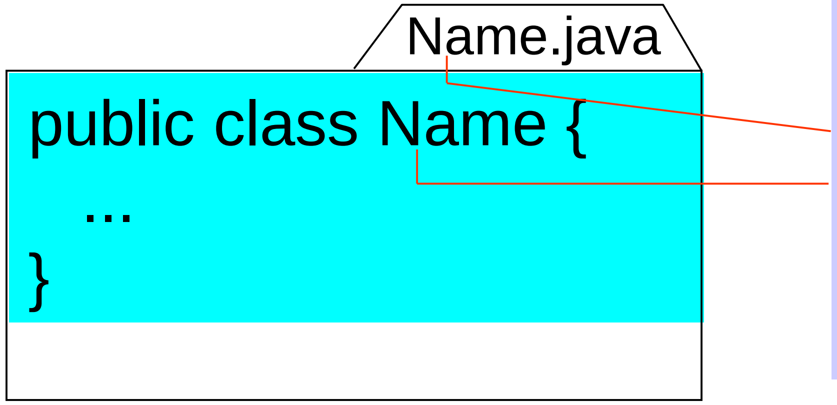
class	↔	user defined data type
objects	↔	data values/state
methods	↔	operators/behaviors

Java's predefined classes

1. arrays
2. strings
3. vectors
4. Hash tables
5. wrapper classes

Classes and Source Files

- Each class is stored in a separate file.
- The name of the file must be the same as the name of the class, with the extension **class**.
- A source file must have the extension **java**.



```
public class Name {
   ...
}
```

By convention, the name of a class (and its source file) always starts with a capital letter.

(In Java, all names are case-sensitive.)

Class definition

SomeClass.java

```
import ...

class-modifier class SomeClass {

  • Fields/members

  • Constructors

  • Methods

}
```

import statements

Class header

Variables that define the object’s state; can hold numbers, characters, strings, other objects

Functions for constructing a new object of this class and initializing its fields

Actions that an object of this class can take (behaviors)

Using classes

1. Declaring object variables:
   `class name variablename;`
2. Creating objects:
   `variable = new classname( );`
3. Accessing variable members
   `variable.v_member;`
4. Accessing method members:
   `variable.method( ) ;`

Object reference

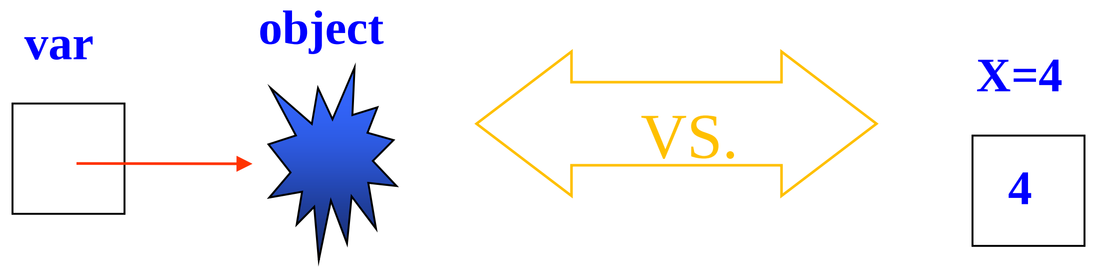
- The new operation instantiates an object of a particular class, and returns a **reference** to it. This reference is a handle to the location where the object resides in memory.

  var = new classname( );

  The reference is stored in the variable “var."
- A declaration like this

  classname var;

  creates a reference variable without instantiating any object.
- Java uses null to represent such un-initialized reference variables.

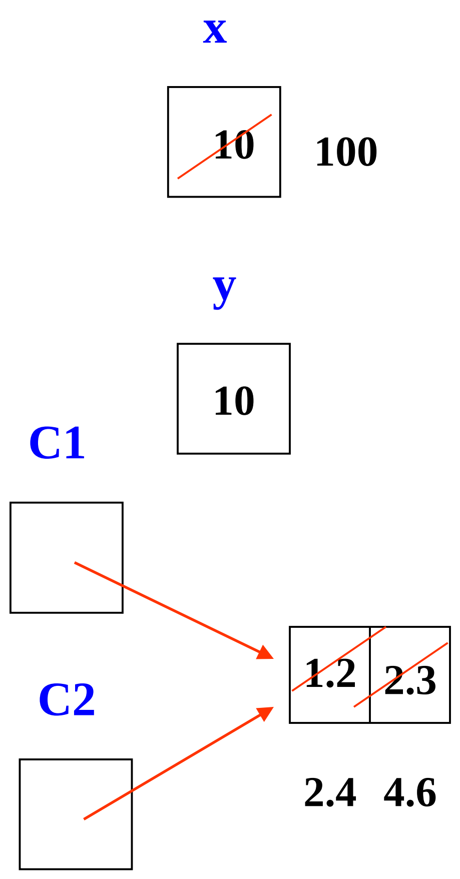


Value vs. Reference

- int x=1, y=1;
  if (x==y) …
- Integer I1, I2;
  I1 = new Integer(5);
  I2 = new Integer(5);
  if (I1==I2)….
- If (I1.equals(I2))….

Value vs. Reference conti.

```
public class primitiveVSreference {
    public static class Complex {
        double re, im;
        public Complex(double x, double y) { re = x; im = y; }
        public String toString() {
            String tmp = "("+re+","+im+")";
            return tmp;
        }
    }
    public static void main(String[] args) {
        int x = 10;
        int y = x;
        x = 100;
        System.out.println("x="+x+", y="+y);
        Complex C1 = new Complex(1.2, 2.3);
        Complex C2 = C1;
        C1.re = 2.4; C1.im = 4.6;
        System.out.println("C1 = "+ C1.toString());
        System.out.println("C2 = "+ C2.toString());
    }
}
```



Method Declaration

```
double add(int x, int y) {
   double sum = x + y;
   return sum;
 }
```

- A method begins with a ***method header***
- The method header is followed by the ***method body***
- In the method header, double is the **return type**, add is the **method name**, and the items within parentheses , i.e. x and y constitute the ***parameter list.***

Method Declaration conti.

- The parameter list specifies the type and name of each parameter
- The return type indicates the type of value that the methods sends back to the calling location
- A method that does not return a value has a **void** return type
- The return statement specifies the value to be returned
- Its expression must conform to the return type

Passing Parameters to Methods

- When a method invocation is made, any actual parameters included in the invocation are passed to the method
- All parameters are **passed by value**. ie, a copy is made.
  - The value of fundamental data types are copied.
  - The value of object references (ie memory addresses) are copied
- Parameters become local variables within the method.

One more time: All parameters are passed by value!

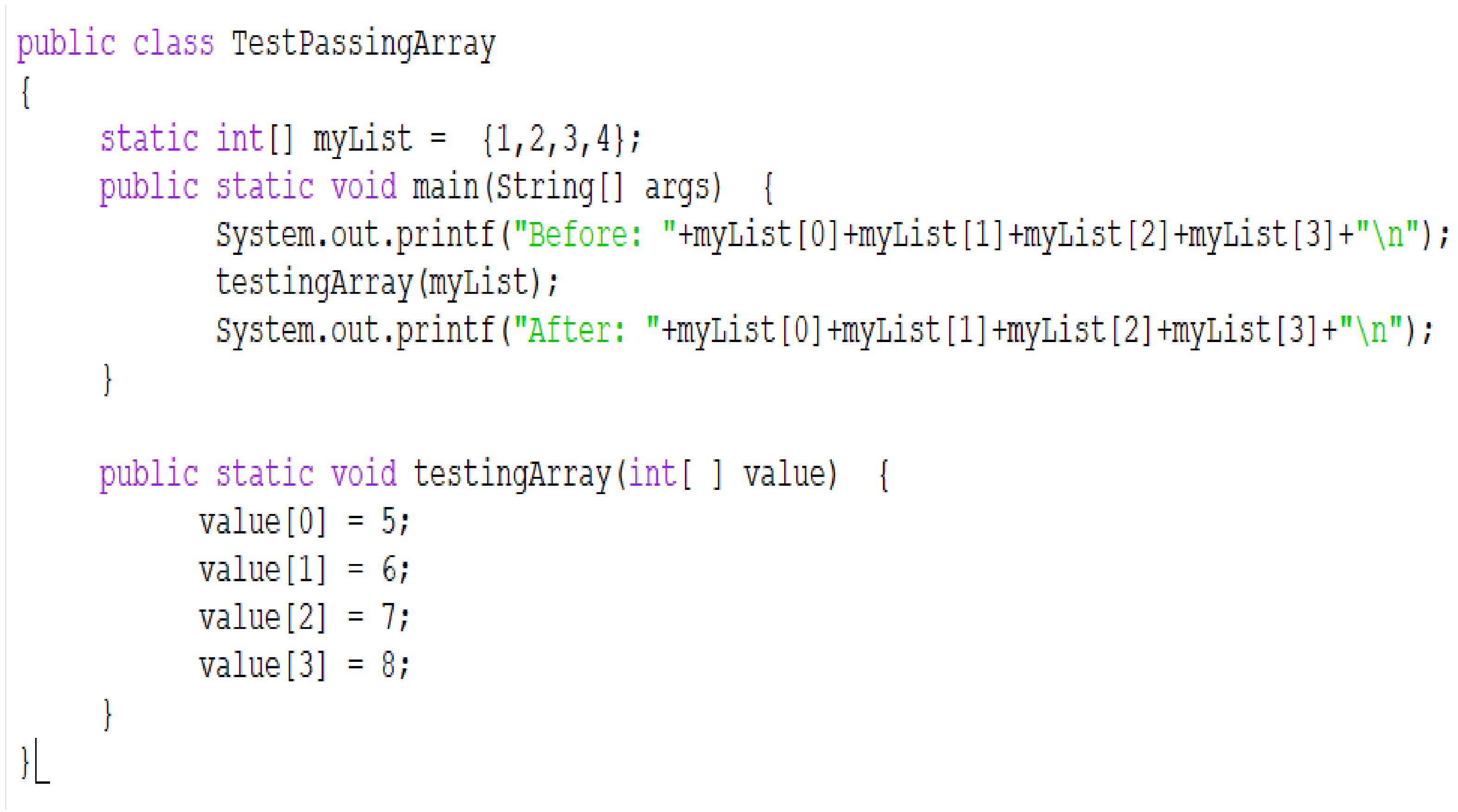
Primitive VS Reference parameters

```
public class primitiveVSreference2 {
    public static class Complex {
        double re, im;
        public Complex(double x, double y) { re = x; im = y; }
        public String toString() {
            String tmp = "("+re+","+im+")";
            return tmp;
        }
    }
    static void add1(int x) { x++; };
    static void Add1(Complex C) { C.re ++; C.im++; }
    public static void main(String[] args) {
        int x = 10;
        add1(x);
        System.out.println("x="+x);
        Complex C = new Complex(1.2, 2.3);
        Add1(C);
        System.out.println("C = "+ C.toString());
    }
}
```

Output:

```
x=10
C = (2.2,3.3)
```

Passing Array



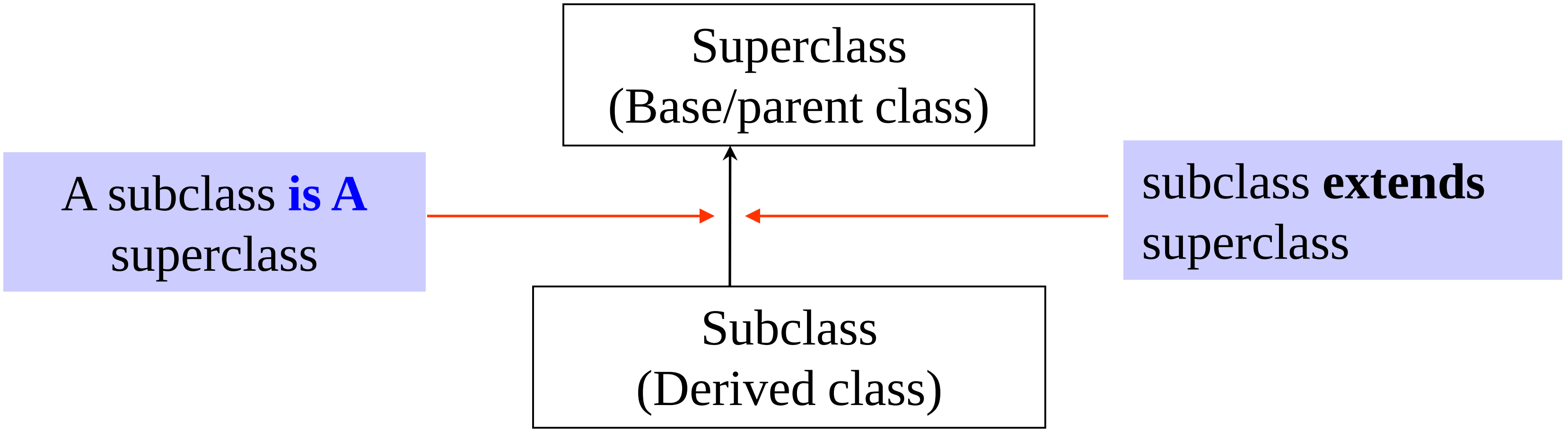
```
public class TestPassingArray
{
     static int[] myList =  {1,2,3,4};
     public static void main(String[] args)  {
            System.out.printf("Before: "+myList[0]+myList[1]+myList[2]+myList[3]+"\n");
            testingArray(myList);
            System.out.printf("After: "+myList[0]+myList[1]+myList[2]+myList[3]+"\n");
     }

     public static void testingArray(int[ ] value)  {
           value[0] = 5;
           value[1] = 6;
           value[2] = 7;
           value[3] = 8;
     }
}
```

Quiz:
Output ?

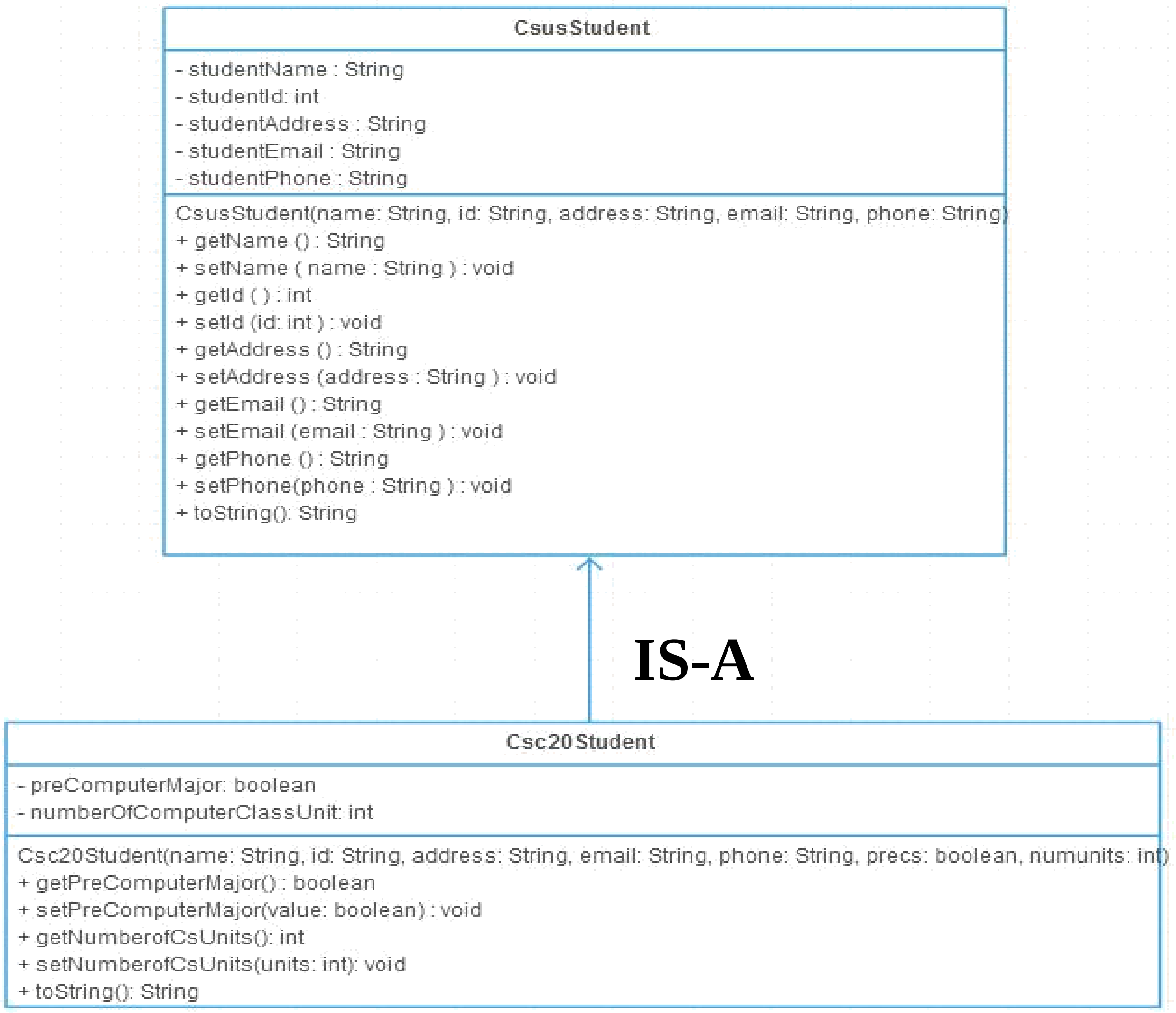
Deriving Classes

```
class classname extends parent-class {
        [variable declaration;]
        [method declaration;]
}
```


Superclass
(Base/parent class)
A subclass **is A** superclass
subclass **extends** superclass
Subclass
(Derived class)

- Inheritance is used to model the Is-A relationship.
- **Inheritance-** inherits fields and methods of its superclass.
- Advantage of inheritance: Code reuse
- Overriding methods - redefine methods of the superclass.
- Overloading methods – more than one method with the same name (but with different parameters) **in a class**.

Inheritance Example 1


CsusStudent
- studentName : String
- studentId: int
- studentAddress : String
- studentEmail : String
- studentPhone : String
CsusStudent(name: String, id: String, address: String, email: String, phone: String)
+ getName () : String
+ setName (name : String) : void
+ getId ( ) : int
+ setId (id: int) : void
+ getAddress () : String
+ setAddress (address : String) : void
+ getEmail () : String
+ setEmail (email : String) : void
+ getPhone () : String
+ setPhone(phone : String) : void
+ toString(): String
IS-A
Csc20Student
- preComputerMajor: boolean
- numberOfComputerClassUnit: int
Csc20Student(name: String, id: String, address: String, email: String, phone: String, precs: boolean, numunits: int)
+ getPreComputerMajor() : boolean
+ setPreComputerMajor(value: boolean) : void
+ getNumberofCsUnits(): int
+ setNumberofCsUnits(units: int): void
+ toString(): String

Inheritance Example 2

```
public class Student {
    int student_id;
    int year;
    String name;

    public Student(String nm, int id, int y)  {
        name = new String(nm);
        student_id = id;
        year = y;
    }
    ….  .
}
```

Note: We will revise the code by declaring the field attributes as private later (with justification).

Inheritance example 2 conti.

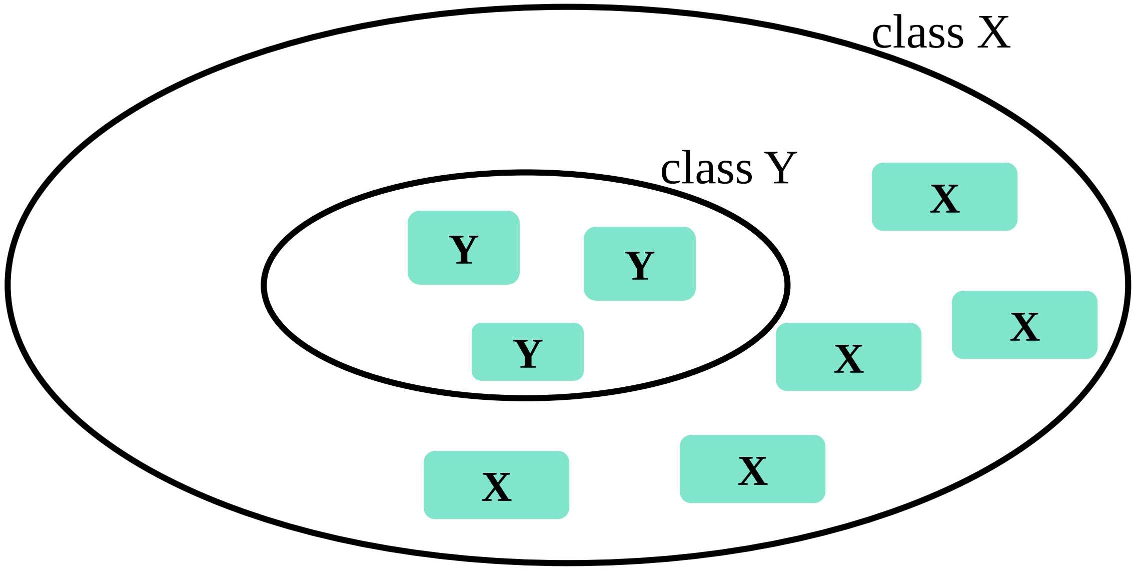
```
public class GradStudent extends Student {
  String dept;
  String thesis;
  // constructor
   public GradStudent(String nm, int id,
                     int y, String d, String th) {
      super(nm, id, y); // call superclass constructor
      /* Or
      name = new String(nm);
      student_id = id;
      year = y;
      */
      dept = new String(d);
      thesis = new String(th);
  }
}
```

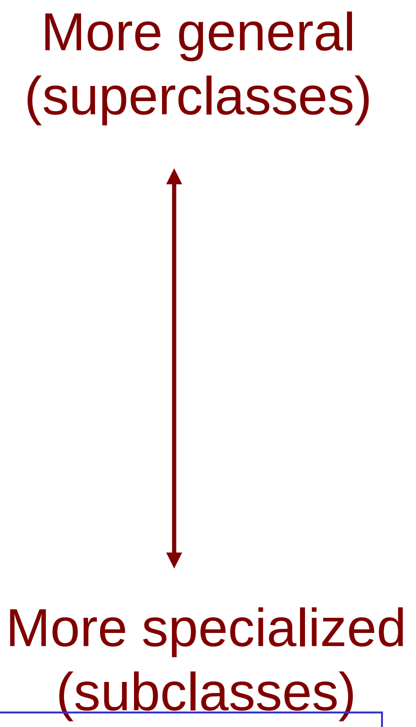
Inheritance example 2 conti.

```
public class inheritanceTest {
  public static void main(String[] args) {
      Student S = new Student(“John”, 1234, 3);
      GradStudent GS = new GradStudent(“Tom”,
                         1111, 2, "CSC", "Algorithm");
      S.student_id = 3690;
      GS.student_id = 2468;
      ….
  }
}
```

```
class Y extends X {
.....
}
```

// y is more specialized





```
X x = new X();
Y y = new Y();
x = y;  ok?
y = x;  ok?
```

instanceof

- instanceof is a keyword that tells you whether a **variable** "is a" member of a class
- Example

  X x;

  if (x instanceof Y) ….

```
// Example: string s is instance of String Class
public class MainClass {
  public static void main(String[] a) {
    String s = "Hello";
    if (s instanceof java.lang.String) {
      System.out.println(s + " is a String");
    }
  }
}
```

Another example - instanceof

```
class Parent {
  public Parent() {
  }
}

class Child extends Parent {
  public Child() {
    super();
  }
}

public class MainClass {
  public static void main(String[] a) {

    Child child = new Child();
    if (child instanceof Parent) {
      System.out.println("true");
    }
  }
}
```

Since a subclass '**is a**' type of its superclass, the above if statement, where Child is a subclass of Parent, returns true.

Java casting

- A narrowing conversion requires an explicit typecast
  - E.g. y = (Y) x;
- The narrowing conversion produces a **runtime** error if x does not contain to a Y object
- The compiler takes the programmer’s word for the validity of the explicit typecast
- The virtual machine checks the validity at the run time

Method Overloading

- Overloading is reusing the same method name with different argument types and perhaps a different return type.
- Methods with overloading names are effectively independent methods.
- overloaded methods can call one another simply by providing a normal method call with an appropriately formed argument list.
- Constructors can also be overloaded after all they are also methods.

```
public class Date {
   private int year, month, day;

   Date() {
      year = month = day = 0;
   }
   Date(int y, int m, int d) {
            year = y; month = m; day = d;
   }
}
```

```
public class program {
     public static void main(String[] args) {

                  Date D1 = new Date();
                  Date D2 = new Date(2002, 7, 20);
                  ...
     }
}
```

Recall: Primitive VS Reference parameters

```
public class primitiveVSreference2 {
    public static class Complex {
        double re, im;
        public Complex(double x, double y) { re = x; im = y; }
        public String toString() {
            String tmp = "("+re+","+im+")";
            return tmp;
        }
    }
    static void add1(int x) { x++; };
    static void Add1(Complex C) { C.re ++; C.im++; }
    public static void main(String[] args) {
        int x = 10;
        add1(x);
        System.out.println("x="+x);
        Complex C = new Complex(1.2, 2.3);
        Add1(C);
        System.out.println("C = "+ C.toString());
    }
}
```

Can you revise this program using Overloading?

Instance Fields

- Can’t override fields
- Can:
  - Inherit a field: All fields from the superclass are automatically inherited
  - Add a field: Supply a new field that doesn’t exist in the superclass
- What if you define a new field with the same name as a superclass field?
  - Each object would have two instance fields of the same name
  - Fields can hold different values
  - Legal but extremely dangerous.

Invoking Super class Methods/Fields

```
// Demonstration program: Dangerous (bad) code
class A {
  int x=1;
  public void printName() {
    System.out.println( "Method A: " + x);
  }
}
class B extends A{
  int x=2;
  public void printName() {
    System.out.println( "Method B: " + x );
    x = 3;
    super.printName();
    super.x = 4; // Note: Keyword super
    System.out.println( "Method B: " + x );
  }
  public static void main(String[] args)
  {
    A myA = new A();
    B myB = new B();
    // myA.printName();
    myB.printName();
  }
}
```

Quiz: Program Output ?

Better way of Initializing or Accessing Data Members

- When there too many items to update/access and also to develop a readable code, generally it is done by defining specific method for each purpose.
- Most important: Safety (avoid corrupting object states)
- To initialise/Update a value:
  - aCircle.setX(10)
- To access a value:
  - aCircle.getX()
- These methods are informally called as Setters/Getters Methods.

Accessors – “Getters/Setters” - Example

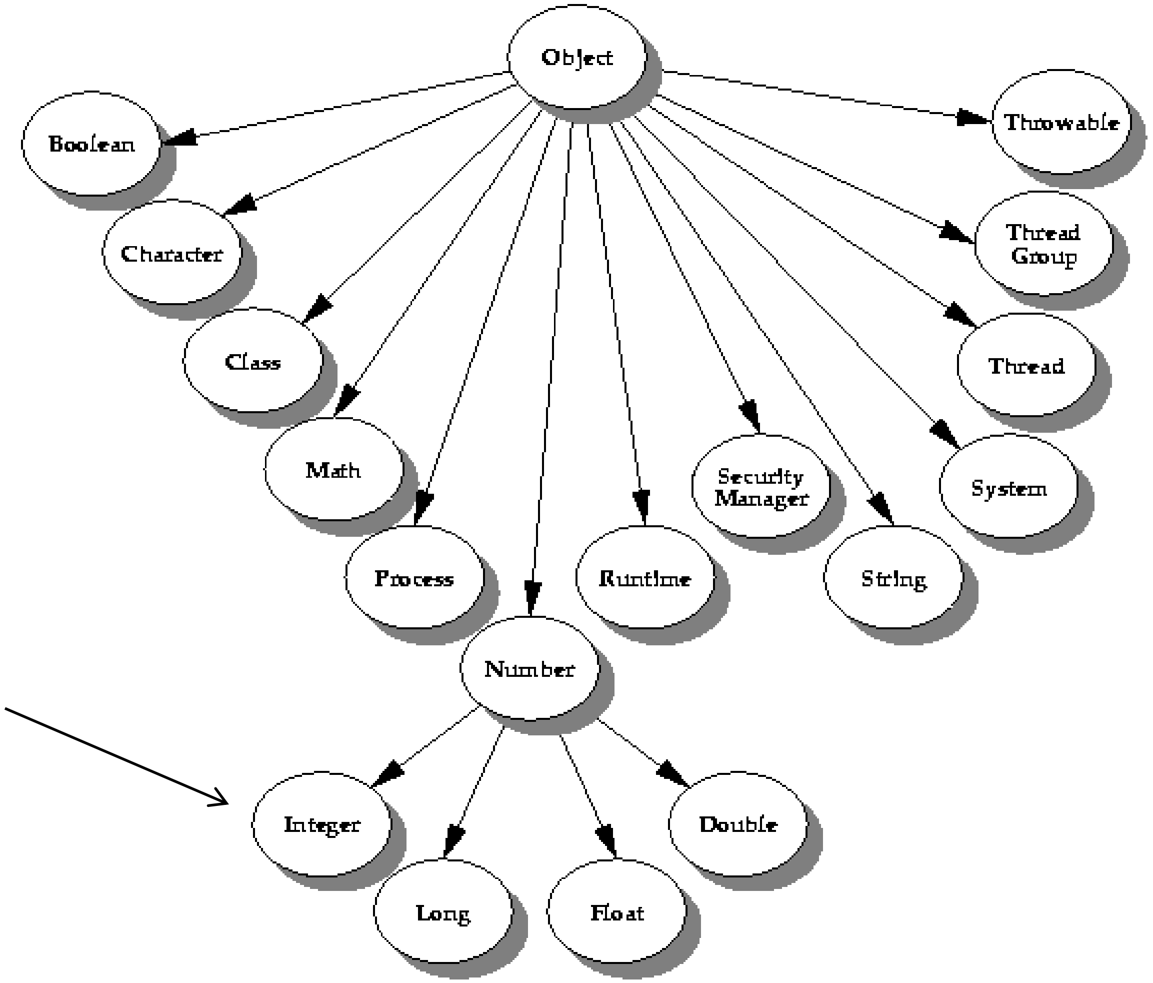
```
public class Circle {
     private double x,y,r;

    //Methods to return circumference and area
    public double getX() { return x;}
    public double getY() { return y;}
    public double getR() { return r;}
    public void setX(double x_in) { x = x_in;}
    public void serY(double y_in) { y = y_in;}
    public void setR(double r_in) { r = r_in;}

}
```

Object: The Cosmic Superclass

Wrapper Classes



The java.lang package contains the collection of base types (language types) that are always imported into any given compilation unit. This is where you'll find the declarations of Object (the root of the class hierarchy) and Class, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.

Source: http://www.oracle.com/technetwork/java/libraries-140433.html

Object: The Cosmic Superclass

- The *Object* class is the highest super class (ie. *root* class) of Java.
- All classes defined without an explicit **extends** clause automatically extend *Object* .
- Most useful methods:
  - String toString()
  - boolean equals(Object otherObject)
  - Object clone() // Create copies
- **Good idea to override these methods in your classes**

Packages - Grouping related classes

1. Declaring packages.

```
package myPackage;
class MyClass {
   ...
}
```

2. Import packages

```
import   myPackage.MyClass;
```

3. **A package must be in a directory with the same name as the package**. Moreover, the directory must be listed in the environment variable CLASSPATH or be a subdirectory of the current directory.

Package declaration & order

1. package name
2. import statements
3. public/non public classes
4. main method - Signature of the main() method can be any of these:
   - public static void main(String args[])
   - public static void main (String [] args)
   - static public void main (String [] args)

Wapper classes

Type wrappers **are classes used to enclose a simple value of a primitive type into an object.**

1. **Integer**
2. **Long**
3. **Boolean**
4. **Character**
5. **Double**
6. **Float**

Notes:

- **All wrapper objects are immutable. Once created the contained value can't be changed.**
- **Wrapper classes are final and can't be subclassed**

Using wrapper classes

Boxing: Wrap a value of a primitive type into a wrapper class, e.g.

```
Chacter C = new Character('x');
```

Unboxing: Extract a value of a primitive type from a wrapper class, e.g. `char c = C.charValue();`

Autoboxing & Unboxing

For Java 1.5 or later, these are legal:

```
Integer X = 5;            //5 is boxed automatically.
Integer Y = 2*X + 3;  //X is unboxed & 13 is boxed automatically.
```

Note: Boxing and unboxing cost CPU time and memory space.

Inner classes

- ❑Most classes we have seen are “top level” classes.
- ❑It is possible (and useful) to define a class inside another class.
- ❑An inner class or nested class is a class declared entirely within the body of another class or interface.
- ❑Every class compiles to a separate `.class` file
- ❑Inner classes compile to files with a `$` in their names
- ❑Inner classes were not in Java 1.0

Member classes

- A *member class* is a class that is declared as a non-static member of a containing class.

```
class Outer {
      int n;
      class Inner {
                int m;
                void setN(int x) { n = x; }
      }
     void f ( ) {
           new Inner( ).setN(5);
     }
}
```

Compiled to Outer$Inner.class file.

Inner has the access to outer's variable n