Class 12 Notes

Review: Principles of Object Oriented Programming: Encapsulation. Inheritance. Polymorphism.

- Encapsulation organizes an application into classes and objects. Objects combine data and actions into one bundle. Indeed, objects model real world entities.

- Inheritance facilitates code reuse. New classes can be created directly from old ones.
  Upcasting in an inheritance hierarchy makes it possible for data of one type to be considered data of a more general type.

- A method may have many forms. Polymorphism, through late binding, ensures that the correct form of a method is chosen **at run time.**

- **Dynamic or late binding** is the default for all method calls except calls to final, private, and static methods, which cannot be overridden and have but one form.

- The ***declared*** **type** of an object and ***real*** type of an object are different

  Animal d = new Dog("Fido")
  The declared od d type is Animal
  The real type of d is Dog

- Late binding is implemented as follows. When choosing an appropriate method for call such as x.myMethod(…), Java **first searches the class of the real type of x** and then continues up through the ancestors of x until a method with a matching signature is found.
  If myMethod() is not part of the declared type, the compiler will issue a syntax error.

- You can upcast to an interface. Comparable x = new Dog("Fido") is fine as long as Dog implements the Comparable interface.

- Overriding methods inherited from the Object class makes it possible for classes to exploit polymorphism correctly and safely. That is, equals(...) should be implemented as
  boolean equals(**Object o**)

- A downcast may be necessary when a parent reference refers to a child object. The segment

```
Parent x;
x = new Child();
x.myMethod();
```

  does not compile if Parent does not declare myMethod(), even if Child does.

  In such a case, a downcast can be used:

```
Parent x;
x = new Child();
((Child)x).myMethod();
```

  However, if Parent has a declaration of myMethod(), no downcast is necessary.

New Stuff Wrappers

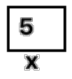
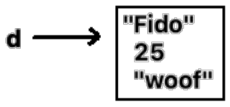
We have seen that the generic sorting method

SelectionSort.sort(Comparable[] x, int size)

Cn sor an array of **objects** of any class that implements the Comparable interface

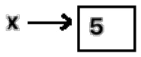
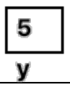
However, this method will not sort an array of int, double, or char.
These are *primitive* data types and are not references to objects.
In fact Java provided just two types of data

1. References
2. Primitive data (int , double, char, float, long, byte)

int x = 5 (x is primitive)	
Dog d = new Dog("Fido", 25, "woof") d is a reference	

Java's Wrapper classes provide genuine classes for each primitive data type.

Java has a class **Integer** → upper case I
Example

1. Integer x = new Integer(5);		"wraps" the value 5 in an object and assigns the address of the object to x
2. int y = 5;		Just stores the value 5 in a memory location labeled "x"

Note:

- (2) is more efficient but sometimes a program needs an object
- The Integer class implements Comparable
- You can use the generic sort to sort a list of integer objects:

```
Integer[] x = new Integer[30];
for (int I = 0; I <10; i++)
{
   int number = input.nextInt(); // number is an int not an Integer
   x[i] = new Integer(number);   // stores an Integer (reference)
}
SelectionSort.sort(x, 10);     // OK since Integer implements Comparable
for(int I = 0; I < 10; i++)
    System.out.println(x[i]);  // uses the toString() of Integer
```

Each primitive data type has a corresponding wrapper class

Primitive	**Wrapper**
int	Integer
double	Double
boolean	Boolean
char	Character
long	Long
float	float
byte	Byte

Note:

- The names of all wrapper classes begin with an upper-case letter

- Each wrapper class has a one argument nonstrucror
  ```
  Integer x = new Integer(5);
  Character c = new Character('A');
   Double d = new Double(3.7);
  ```

- There are no default constructors
  ILLEGAL → Integer x = new Integer()

- Each ***numeric*** wrapper class (Integer, Double, Long, Float, Byte) has a one argument constructor that accepts a String argument:
  ```
  Integer x = new Integer("12345");
  or
  String s = input.next();
  Integer y = new Integer(s);
  ```

Autoboxing and unboxing (Added after Java 6)

Converting from primitive to wrapper can be done automatically. This is called ***autoboxing***
Converting from wrapper to primitive can also be done automatically. This is ***unboxing***

Example:
What we have already seen:

Integer x = new Integer(5) (5 is an int) $x \longrightarrow \boxed{5}$

Can also be accomplished as

Integer x = 5; (5 is a primitive)

A primitive assigned to a wrapper automatically gets "boxed" inside an object (Autoboxing)

Example:
1. Integer x = 10; // x is a reference , 10 is an int (primitive)
2. Integer y = 20;

What is happening here?

Integer x = 10; **// x is a reference** **// 10 is an int (primitive)**	10 is autoboxed	$x \longrightarrow \boxed{10}$
Integer y = 20;	20 is autoboxed	$y \longrightarrow \boxed{20}$
Integer z = x *y ; **// remember x and y are** **//references**	Unbox x (get 10 (int)) Unbox y (get 20 (int)) Multiply 10*20 (get 200 (int)) autobox 200 in z	$z \longrightarrow \boxed{200}$

Obviously, using wrapper classes is not as efficient as using primitive classes. But sometimes it is more convenient to use objects. For example if you want to use a generic sort on integers you would use Integer objects.

This may seem a little strange:

```
public class TestEquals

{
   public static void main(String[] args)
   {
      int a = 5;
      int b = 5;

      Integer c = new Integer(5);
      Integer d = new Integer(5);

      Integer e = 5;
      Integer f = 5;

      System.out.println("a == b "+ (a== b));
      System.out.println("a == c "+ (a == c));
      System.out.println("c == d "+ (c == d));
       System.out.println("c.equals(d) "+c.equals(d));
      System.out.println("e == f "+ (e == f));
   }
}
```

Output:
a == b ________

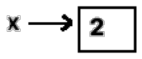
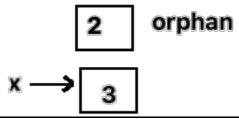
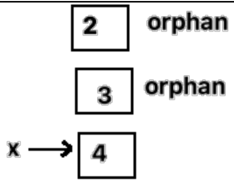
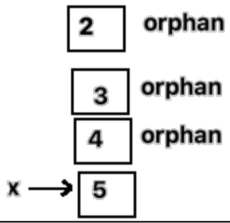
a == c ________

c == d _______

c.equals(d) ________

e == f ________

Like Strings **wrapper objects are immutable.** Once created the value stored in a wrapper object cannot be changed.

Integer x = 2;	 x is a reference
x = 3;	
x = 4;	
x = 5;	

Of course, the unreferenced memory is recycled by the ***garbage collector.***

Here are some methods of the wrapper classes. All are static and can be called using the name of the class, e.g. Integer.something().

Method	return type	Description	Example
Integer.valueOf(String s)	Integer	Returns reference to an Integer object initialized to the numeric value of s	Integer x = Integer.valueOf("345");
Double.valueOf(String s)	Double	Returns a reference to a Double object initialized to the numeric value of s	Double x = Double.valueOf("3.14159");
Integer.parseInt(String s)	int	Returns the numeric value of s as a primitive	int x = Integer.parseInt("345");
Double.parseDouble(String s)	double	Returns the numeric value of s as a primitive	double x = Double.parseDouble("3.14159");
Integer.toString(int x)	String	Returns the integer x as a String	String s = Integer.toString(123);
Double.toString(double x)	String	Returns the double x as a String	String s = Double.toString(3.14159);

Some static methods of the Double and Integer classes. Similar methods are defined for Byte, Long, and Float.

Method	return type	Description	Example
Character.isDigit(char ch)	boolean	Returns *true* if ch is a digit	Character.isDigit('w') returns *false* W is not a digit
Character.isLetter(char ch)	boolean	Returns *true* if ch is a letter	Character.isLetter('w') returns *true*
Character.isLetterOrDigit(char ch)	boolean	Returns *true* if ch is a letter or a digit	Character.isLetterOrDigit('$') returns *false*
Character.isLowerCase(char ch)	boolean	Returns *true* if ch is a lower case letter	Character.isLowerCase('w') returns *true*
Character.isUpperCase(char ch)	boolean	Returns *true* if ch is an uppercase letter	Character.isUpperCase('w') returns *false*
Character.isWhitespace(char ch)	boolean	Returns *true* if ch is a blank, a tab, a form feed, or a line separator	Character.isWhitespace('x') returns *false*
Character.toLowerCase(char ch)	char	Returns the lowercase version of ch if ch is an alphabetical character, otherwise returns ch	Character.toLowerCase('A') returns 'a' Character.toLowerCase('#') returns '#'
Character.toUpperCase(char ch)	char	Returns the uppercase version of ch if ch is an alphabetical character, otherwise returns ch	Character.toUpperCase('r') returns 'R' Character.toUpperCase('#') returns '#'

Some static methods of the Character class

Some of the more important methods:

int Integer.parseInt(String s) // returns an int (primitive)
int number = Integer.parseInt(“567”);   //number is the int 567

Integer .valueOf(String s) // returns an Integer reference (wrapper)
Integer x = Integer.valueOf(“567”); // x is an Integer reference (object)

The Character methods can be very convenient:
Character.isLetterOrDigit(‘5’) returns true
Character.isUpperCase(‘a’) returns false
Character.toUpperCase(‘a’) returns ‘A’

Problem:
When using a Scanner to read ints (or doubles) if you accidentally enter a character you program may crash:

```
public class DataDemo
{
  public static void main(String[] args)
  {
    Scanner input = new Scanner(System.in);
    System.out.println("Enter five numbers");
    for (int i = 1; i <= 5; i++)
    {
      int x = input.nextInt();
      System.out.println("The number is "+ x);
    }
  }
}
```

Output:

```
Enter five numbers
 2
The number is 2
3
The number is 3
e
java.util.InputMismatchException
	at java.util.Scanner.throwFor(Scanner.java:840)
	at java.util.Scanner.next(Scanner.java:1461)
	at java.util.Scanner.nextInt(Scanner.java:2091)
	at java.util.Scanner.nextInt(Scanner.java:2050)
	at DataDemo.main(DataDemo.java:10)
```

Design a class **ReadData** with a utility method **readInt()** which can be used to read an integer from the keyboard and flags errors if input is not correct. Assume that a number can be negative.

Algorithm:
We will read a number as a String and check to see whether or not each character is valid.
For example “4567” has four valid characters but “45&6” does not.
We will use two boolean flags

1. boolean correct → true idf the input is OK
2. boolean negative → true if the number is negative (begins with -)

Here is the algorithm:

```
Negative = false;
Loop
          correct = true;
          Read a String, number   (e.g. number =  “246”)
          If the first character of number is a  minus sigh
             set negative = true;
             remove the minus sign
          For each character ch  of number
             if the ch is not a digit
              {
                   correct = false
                   print a message
               }
while not correct
if negative is true return -Integer.parseInt(number)
else
return Integer.parseInt(number)
```

Here is the code:

```
import java.util.*;
public class ReadData
{
   public static int readInt()
   {
      // returns a valid integer that is supplied interactively
      Scanner input = new Scanner(System.in);

      boolean correct;                    // is the input  correct?
      boolean negative = false;          // is the number negative?
      String number;                     // input string

      do
      {
         correct = true;
         number = input.next();              // read a string
         if (number.charAt(0) == '-')        // negative number?
         {
            negative = true;
            number = number.substring(1,number.length());  // removes the - sign
         }

         for( int i = 0; i < number.length(); i++)
            if (!Character.isDigit(number.charAt(i)))       // input  error
            {
               correct = false;
               System.out.print("Input error, reenter: ");
               break;
            }
      }while(!correct);

      if (negative)
         return - Integer.parseInt(number);
      return Integer.parseInt(number);
   }
```

public class DataDemo1
{
 public static void main(String[] args)
 {
  System.out.print("Enter an int: ");
  int x= ReadData.readInt();
  System.out.println("The square is "+ x*x);
  }
}
	Output:
	Enter an int: **23r5**
	Input error, reenter: **23w**
	Input error, reenter: **24**
	The square is 576

What about a Double? ReadData.readDouble()?

You have to deal with the decimal point. Find the decimal point and check the characters to the left of the decimal point and to the right ensuring that they are digits. readDouble() Should also be able to handle numbers that do not have a decimal.

Here is the code.

```
public static double readDouble()
   {
      //returns a valid double that is supplied interactively
      Scanner input = new Scanner(System.in);

      boolean correct;
      boolean negative = false;     // negative number?
      String number;
      int decimalPlace;       // index of the decimal point , if no decimal then it will be -1

      do
      {
         correct = true;
         number = input.next();   // read a string e,g -345.67

         if (number.charAt(0) == '-')
         {
            negative = true;
            number = number.substring(1,number.length());  //remove minus sign
         }

         decimalPlace = number.indexOf(".");
          // -1 if no decimal point
```

```
        // validate that the characters up to the decimal are digits
        // that is check the characters to the left of the decimal
        // this loop is skipped if there is no decimal point
        // or the decimal occurs as the first character

        for(int i = 0; i < decimalPlace; i++)                  // skipped if decimalPlace == -1
            if (!Character.isDigit(number.charAt(i)))      // input error
            {
                correct = false;
                System.out.print("Input error, reenter: ");
                break;
            }

        // validate that the characters after the decimal are digits
        for(int i = decimalPlace+1; i < number.length(); i++)
                    // If decimalPlace is -1, i is initially 0

                 if (!Character.isDigit(number.charAt(i)))  // input error
                 {
                             correct = false;
                            System.out.print("Input error, reenter: ");
                           break;
                 }
    }while (!correct); // end of do loop
    if (negative)
        return -Double.parseDouble(number);
    return Double.parseDouble(number);
}

}
```