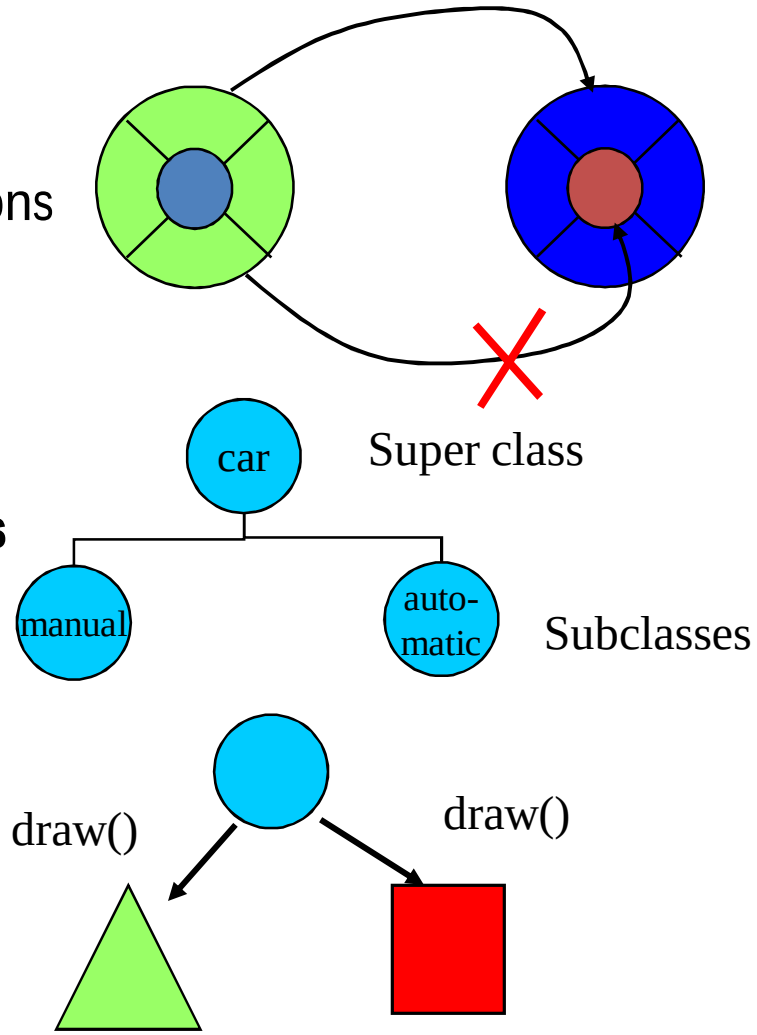
Java Methods & Classes

Introduction: Classes are Object

- OOP - object oriented programming
- code built from objects
- Java these are called ***classes***
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name

The three principles of OOP

- Encapsulation
  - Objects hide their functions (**methods**) and data (**instance variables**)
- Inheritance
  - Each **subclass** inherits all variables of its **superclass**
- Polymorphism
  - Interface same despite different data types



Simple Class and Method

```
Class Fruit{
    int grams;
    int cals_per_gram;

    int total_calories() {
        return(grams*cals_per_gram);
    }
}
```

Methods

- A method is a named sequence of code that can be invoked by other Java code.
- A method takes some parameters, performs some computations and then optionally returns a value (or object).
- Methods can be used as part of an expression statement.

```
public float convertCelsius(float tempC) {
      return( ((tempC * 9.0f) / 5.0f) + 32.0 );
  }
```

Method Signatures

- A method signature specifies:
  - The name of the method.
  - The type and name of each parameter.
  - The type of the value (or object) returned by the method.
  - The checked exceptions thrown by the method.
  - Various method modifiers.
  - *modifiers type name (parameter list) [throws exceptions]*

```
public float convertCelsius (float tCelsius ) {}
public boolean setUserInfo ( int i, int j, String name ) throws
    IndexOutOfBoundsException {}
```

Application: Public/private

- Methods/data may be declared ***public*** or ***private*** meaning they may or may not be accessed by code in other classes ...
- Good practice:
  - keep data private
  - keep most methods private
- well-defined interface between classes - helps to eliminate errors

Using objects

- Here, code in one class creates an instance of another class and does something with it ...

```
Fruit plum=new Fruit();
int cals;
cals = plum.total_calories();
```

- ***Dot operator*** allows you to access (public) data/methods inside Fruit class

Constructors

- The line

  plum = new Fruit();

- invokes a constructor method with which you can set the initial data of an object
- You may choose several different type of constructor with different argument lists

  eg Fruit(), Fruit(a) ...

Overloading

- Can have several versions of a method in class with different types/numbers of arguments

```
Fruit() {grams=50;}
Fruit(a,b) { grams=a; cals_per_gram=b;}
```

- By looking at arguments Java decides which version to use

Java Development Kit

- javac - The Java Compiler
- java - The Java Interpreter
- jdb - The Java Debugger
- appletviewer -Tool to run the applets

- javap - to print the Java bytecodes
- javaprof - Java profiler
- javadoc - documentation generator
- javah - creates C header files

Stream Manipulation
Streams and I/O

- basic classes for file IO
  - FileInputStream, for reading from a file
  - FileOutputStream, for writing to a file
- Example:

Open a file "myfile.txt" for reading

```
FileInputStream fis = new FileInputStream("myfile.txt");
```

Open a file "outfile.txt" for writing

```
FileOutputStream fos = new FileOutputStream ("myfile.txt");
```

Display File Contents

```
import java.io.*;
public class FileToOut1 {
   public static void main(String args[]) {
      try   {
         FileInputStream infile = new FileInputStream("testfile.txt");
         byte buffer[] = new byte[50];
         int nBytesRead;
         do   {
            nBytesRead = infile.read(buffer);
              System.out.write(buffer, 0, nBytesRead);
         } while (nBytesRead == buffer.length);
      }
      catch (FileNotFoundException e)  {
         System.err.println("File not found");
      }
      catch (IOException e) { System.err.println("Read failed"); }
   }
}
```

Filters

- Once a stream (e.g., file) has been opened, we can attach filters
- Filters make reading/writing more efficient
- Most popular filters:
  - For basic types:
    - DataInputStream, DataOutputStream
  - For objects:
    - ObjectInputStream, ObjectOutputStream

Writing data to a file using Filters

```
import java.io.*;
public class GenerateData {
   public static void main(String args[]) {
      try   {
         FileOutputStream fos = new FileOutputStream("stuff.dat");
         DataOutputStream dos = new DataOutputStream(fos);
         dos.writeInt(2);
         dos.writeDouble(2.7182818284590451);
         dos.writeDouble(3.1415926535);
         dos.close(); fos.close();
      }
      catch (FileNotFoundException e) {
          System.err.println("File not found");
      }
      catch (IOException e) {
         System.err.println("Read or write failed");
      }
   }
}
```

Reading data from a file using filters

```
import java.io.*;
public class ReadData {
   public static void main(String args[]) {
      try {
         FileInputStream fis = new FileInputStream("stuff.dat");
         DataInputStream dis = new DataInputStream(fis);
         int n = dis.readInt();
         System.out.println(n);
         for( int i = 0; i < n; i++ ) { System.out.println(dis.readDouble());
         }
         dis.close(); fis.close();
      }
      catch (FileNotFoundException e) {
         System.err.println("File not found");
      }
      catch (IOException e) { System.err.println("Read or write failed");
      }
   }
}
```

Object serialization

Write objects to a file, instead of writing primitive types.

Use the ObjectInputStream, ObjectOutputStream classes, the same way that filters are used.

Scope: Write an object to a file

```
import java.io.*;
import java.util.*;
public class WriteDate {
   public WriteDate () {
      Date d = new Date();
      try {
            FileOutputStream f = new FileOutputStream("date.ser");
            ObjectOutputStream s = new ObjectOutputStream (f);
            s.writeObject (d);
            s.close ();
      }
      catch (IOException e) { e.printStackTrace(); }

    public static void main (String args[]) {
      new WriteDate ();
    }
}
```

Read an object from a file

```
import java.util.*;
public class ReadDate {
  public ReadDate () {
    Date d = null;
    ObjectInputStream s = null;
    try {  FileInputStream f = new FileInputStream ("date.ser");
      s = new ObjectInputStream (f);
    } catch (IOException e) { e.printStackTrace(); }
    try { d = (Date)s.readObject (); }
    catch (ClassNotFoundException e) { e.printStackTrace(); }
    catch (InvalidClassException e) { e.printStackTrace(); }
    catch (StreamCorruptedException e) { e.printStackTrace(); }
    catch (OptionalDataException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }
    System.out.println ("Date serialized at: "+ d);
  }
  public static void main (String args[]) { new ReadDate ();  }
}
```