Worked example

Worked example: writing classes

Problem

- Create a phone book application that stores names and numbers, allows a user to add entries and retrieve the number for a given name, and load/save the data to file.

Classes

- PhoneBookEntry:
- PhoneBook:

Worked example

`PhoneBookEntry` Implementation

```
public class PhoneBookEntry {
    String name; // instance variables
    String number; // to store data
}
```

1. We want to ensure that every entry has a name and a number - **CONSTRUCTORS**
2. We'd like to control how code accesses the name and number – **ACCESS SPECIFIERS**
3. We'd like to make it easy to output `PhoneBookEntry` objects – **OVERRIDE toString()**

Worked example

Constructors (definition)

```
public class MyClass {
    public MyClass() {
        // Initialisation code here
    }
}
```

Remember:

A **constructor** is a special method that allows you to control how objects are instantiated.

Worked example

Constructors (PhoneBookEntry)

```
public class PhoneBookEntry {
   public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }
}
```

OR

```
public class PhoneBookEntry {
   public PhoneBookEntry( String name, String number ) {
        this.name = name;
        this.number = number;
    }
}
```

this refers instance variables

Notes on constructors:

- Constructors **do not** have a **return value** specified.
- **Constructors** must have the **exact same name** as the **class**.
- If you create a constructor that takes arguments, then you cannot use a default constructor unless you explicitly define one.

Worked example

Access specifiers – **`public`**

The **`public`** keyword means that the following member or class declaration is available to everyone.

MyClass.java

```
public class MyClass {
    public void someMethod() {
        // Implementation
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    public void someOtherMethod() {
        MyClass m = new MyClass();
        m.someMethod();
    }
}
```

Worked example

Access specifiers – `private`

The **`private`** keyword means that no one has access to the method except inside others of that particular class.

MyClass.java

```
public class MyClass {
    private int myField;
    private void someMethod() {
       // Implementation.
    }
    public void otherMethod() {
       System.out.println(myField);
       someMethod();
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    private void print() {
       MyClass m = new MyClass();
       m.someMethod();
       System.out.println(myField);
    }
}
```

Worked example

Access specifiers – `protected and package access`

Definition (**`protected`**)

- The `protected` keyword means that access is only allowed from within the classes that extend the particular class.

Definition (**package access**)

- If no access specifier is included then the member is only available to other classes in the same package. This is sometimes termed as *friendly* access.

Worked example

Access specifiers – `Accessors`

`private data`

```
public class PhoneBookEntry {
     private String name;
     private String number;
}
```

`Accessor`

```
public String getName() {
     return name;
}
```

An ***accessor*** method allows ***read access*** to a private field (instance variable).

Worked example

Access specifiers – Mutators

private data

```
public class PhoneBookEntry {
     private String name;
     private String number;
}
```

Accessor

```
public String setName() {
     name = “matt”;
}
```

An ***mutator*** method allows ***write access*** to a private field (instance variable).

Worked example

`String` and `StringBuffer` classes

- As a quick aside, lets introduce the `StringBuffer` classes.
- `String` objects are *immutable* (i.e. they cannot be modified) whereas `StringBuffer` objects are *mutable* and can be modified.

String

```
String s = “abc”;
s = s.toUpperCase();
s.toLowerCase();
```

StringBuffer

```
StringBuffer sb = new StringBuffer(“ABC”);
sb.append(“DEF”);
```

Worked example

Easy output of the **PhoneBookEntry** class

- We want a way to output an entry to standard output.
- We can achieve this, by overriding the toString() method.
- We'll come back a little later on to look at exactly what is going on here.
- If we add the following method:

```
public String toString() {
    String s = name + "\t(" + number + ") ";
    return s;
}
```

- We can then print phonebook entries.

Finished PhoneBookEntry application

```
class PhoneBookEntry {

    private String name;
    private String number;

    public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }

    public String getName( ) {
        return name;
    }

    public String getNumber( ) {
        return number;
    }

    public void setNumber( String inNumber ) {
        number = inNumber;
    }

    public String toString( ) {
        String s = name + "\t(" + number + ") ";
        return s;
    }
}
```

PhoneBookEntryTest application

Lets now write an application to use our new PhoneBookEntry class and test our code:

```
public class PhoneBookEntryTest {
    public static void main( String[] args ) {
        PhoneBookEntry e = new PhoneBookEntry( "Bob", "+44
          0123456789" );
        System.out.println( e ); }
    }
}
```

Anatomy of a Java class

JavaClass.java

```
public class JavaClass {

        private final String name;
        private String description;
        private static int staticInstanceVariable = 0;
        private int instanceVariable = 1;

        public JavaClass( String id, String description ){
                name = id;
                this.description = description;
        }

        public static void staticMethod() {
                System.out.println("JavaClass static method.");
        }

        public static int staticMethodWithReturn() {
                return staticInstanceVariable;
        }
        public void instanceMethod() {
                System.out.println("JavaClass instance method.");
        }

        public int instanceMethodWithReturn() {
                return instanceVariable;
        }

        public String toString() {
                return name + " -> " + description;
        }

        public static void main( String[] args ) {

                // Using static methods

                JavaClass.staticMethod();
                staticMethod();
                System.out.println( JavaClass.staticMethodWithReturn() );
                System.out.println( staticMethodWithReturn() );

                // Using instance methods - need to call from instantiated object

                JavaClass jc = new JavaClass("matts_class", "Anatomy of a Java Class");
                jc.instanceMethod();
                System.out.println( jc.instanceMethodWithReturn() );
                System.out.println( jc );

        }

}
```

class name

instance variables

constructor

this refers to instance variable, without this it refers to local variable

static methods

note static variable in static method

instance methods

override toString()

instantiate new JavaClass object

invoke constructor

use dot operator to access methods of JavaClass object

automatically invoke toString() method

Worked example

REMEMBER FROM LAST VIDEO SERIES:

Problem

- Create a phone book application that **stores names** and **numbers**, allows a user to **add** entries and **retrieve** the number for a given name, and **load/save** the **data** to **file**.

Classes

- PhoneBookEntry: **// COMPLETED LAST WEEK**
- PhoneBook: **// TO DO**

Worked example

Finished PhoneBookEntry application

```
class PhoneBookEntry {

    private String name;
    private String number;

    public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }

    public String getName( ) {
        return name;
    }

    public String getNumber( ) {
        return number;
    }

    public void setNumber( String inNumber ) {
        number = inNumber;
    }

    public String toString( ) {
        String s = name + "\t(" + number + ") ";
        return s;
    }
}
```

Worked example

PhoneBookEntryTest application

Lets now write an application to use our new PhoneBookEntry class and test our code:

```
public class PhoneBookEntryTest {
    public static void main( String[] args ) {
        PhoneBookEntry e = new PhoneBookEntry( "Bob", "+44
          0123456789" );
        System.out.println( e ); }
    }
}
```

Worked example

The **PhoneBook** class

- **PhoneBook** class will hold instances of **PhoneBookEntry** objects, just as a real Phone Book will contain Entries.
- Lets start by storing our **PhoneBookEntry** objects in an array.

```
PhoneBookEntry[] entries = new PhoneBookEntry[10];
```

- This will create an array of type **PhoneBookEntry** of length 10.
- **But Remember:** we can't resize arrays in Java.
- Now, we don't know how many entries there will be in advance, so we're going to have to overcome this shortcoming.
- But for now, lets assume that there will be less than 10 entries and build a **PhoneBook** class.

Worked example

EXAMPLE: PhoneBookBasic Folder

A Basic PhoneBook class

```
public class PhoneBook {
   private PhoneBookEntry[] entries;
   private int size;
   final private static int MAX_ENTRIES = 10;

   public PhoneBook( ) {
      entries = new PhoneBookEntry[MAX_ENTRIES];
   }

   public void add( String name, String number ) {
      System.out.println("Size: " + size);
      if (size != MAX_ENTRIES) {
         entries[size] = new PhoneBookEntry( name, number );
         size++;
      }
   }

   public String toString() {
      StringBuffer temp = new StringBuffer();
      for (int i = 0; i < size; ++i) {
         temp.append( entries[i].toString() + "\n" );
      }
      return temp.toString();
   }
}
```

Worked example

EXAMPLE: PhoneBookBasic Folder

PhoneBookTest application

Lets now write an application to use our new PhoneBook class and test our code:

```
public class PhoneBookTest {

   public static void main( String[] args ) {

      // Create and fill phone book
      PhoneBook pb = new PhoneBook();
      pb.add( "Bob", "+44 (0) 483984" );
      pb.add( "Carl", "38478" );
      pb.add( "Don", "3878" );
      pb.add( "Ed", "3848" );
      pb.add( "Frank", "8478" );

      // Output whole book
      System.out.println();
      System.out.println( pb );
      System.out.println();
   }

}
```

Worked example

- However, this is far from ideal!!
- We don’t want to have to set a fixed size for our phone book in advance.
- With a little more work we can make our storage more intelligent, by following the psuedocode below when we add an entry:

Algorithm Add a new entry:

```
if array is full then
   create new array with double the size
   copy existing entries into new array
   repoint reference to new array
end if
add entry at next free position
increment free position
```

Worked example

EXAMPLE: PhoneBook Folder

```
public class PhoneBook {

   private PhoneBookEntry[] entries;
   private int size;
   private int maxEntriesLength = 1;

   public PhoneBook( ) {
      entries = new PhoneBookEntry[maxEntriesLength];
   }

   public void add( String name, String number ) {
      System.out.println("Size: " + size + " , " + maxEntriesLength);
      if ( size == entries.length ) {
         System.out.println("Doubling");
         maxEntriesLength = 2 * maxEntriesLength;
         PhoneBookEntry[] temp = new PhoneBookEntry[ maxEntriesLength ];
         System.arraycopy( entries, 0, temp, 0, entries.length );
         entries = temp;
      }
      entries[size] = new PhoneBookEntry( name, number );
      System.out.println("Adding : " + name);
      size++;
      System.out.println("Size: " + size + " , " + maxEntriesLength);
   }

}
```

Worked example

EXAMPLE: LinearSearch Folder

Finding Numbers

Lets implement a `numberFor` method, so we can search the number for a given name. We could use a linear search algorithm:

Linear search
Linear search is a simple search algorithm that can find the position of a specified value (called the key) within an array. The key is simply compared to each element in turn and its position is returned if it is found. If it is not found then -1 is returned.

Algorithm 2 Linear Search

```
Require: array, key
for each element of the array do
   if element = key then
      return index of element
   end if
end for
return -1
```

Lets try and make it easier

Worked example

EXAMPLE: PhoneBookWithSearch Folder

Easier `numberFor()` method

Given that the **Strings** are **objects**, we can loop through the phonebook and compare the search string with each name string, using the **`equals()`** method.

```
public String numberFor( String name ) {
    for (int i = 0; i < size; ++i) {
        if ( list[i].getName().equals(name) ) {
            return list[i].getNumber();
        }
    }
    return "NOT FOUND";
}
```

Worked example

- In fact we don't need to write our own code to implement a **collection** (an object that can hold references to other objects) that can grow arbitrarily.
- The core Java API contains several classes that we could use depending on the features we would like our collection to have.
- We will start with the **Vector** class.
- From Java 1.5, the Vector class makes use of a new feature – **generics**.
- This allows us to specify the type of object that the collection will contain when we declare the Vector.
- Previously collections in Java stored elements of type Object that had to be cast when they were accessed.

Worked example

EXAMPLE: PhoneBookVector Folder

public class **Vector<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

```
import java.util.Vector;

public class PhoneBook {

    private Vector<PhoneBookEntry> entries;

    public PhoneBook( ) {
        entries = new Vector<PhoneBookEntry>();
    }

    ........

}
```

Worked example

The usual question at this stage is:

Why Vector?

They're really old!!

Everyone uses ArrayLists now

Ok, that's easy......

Worked example

EXAMPLE: PhoneBookArrayList Folder

public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

```
import java.util.ArrayList;

public class PhoneBook {

    private ArrayList<PhoneBookEntry> entries;

    public PhoneBook( ) {
        entries = new ArrayList<PhoneBookEntry>();
    }

    ........

}
```

Reading Text Files

Reading text files

Using the `Scanner` class

Reading from standard input

```
Scanner in = new
Scanner(System.in);
int x = in.nextInt();
```

Reading from input.txt file

```
Scanner in = new Scanner(new
File("input.txt"));
int x = in.nextInt();
in.close();
```

NOTE: this will not compile as written — working with files in Java **requires** us to handle errors using ***exception handling***

Exception Handling

- ***Exception handling*** provides a flexible mechanism for handling/recovering from errors.
- An ***exception*** is a problem that prevents the normal execution of a method.
- Exceptions are only intended for ***exceptional*** circumstances and should not be used for general flow control.

Example

EXAMPLE: ExceptionTest Folder

```
public class ExceptionTest {

  public static void main( String[] args ) {

    try {
      System.out.println(
              "Trying Integer.parseInt( \"NOT AN INT\" )" );
      int i = Integer.parseInt( "Not an int" );
    }
    catch ( Exception e ) {
      System.out.println("Some error" + e);
      e.printStackTrace();
    }
    finally {
      System.out.println( "Finally always reached" );
    }
  }
}
```

General Form

General form

```
try {
  // Code that may throw exception
}
catch (ExceptionType e) {
  // Handle exception
}
finally { // Optional
  // Tidy up
}
```

Example (Throwing exceptions)

```
if (Boolean-expression) {
  throw new NumberFormatException("any extra information");
}
```

Exceptions are objects of the Exception class (or a class that extends the Exception class). Useful methods defined in this class include:

- toString
- printStackTrace

Stack Traces

EXAMPLE: StackTrace Folder

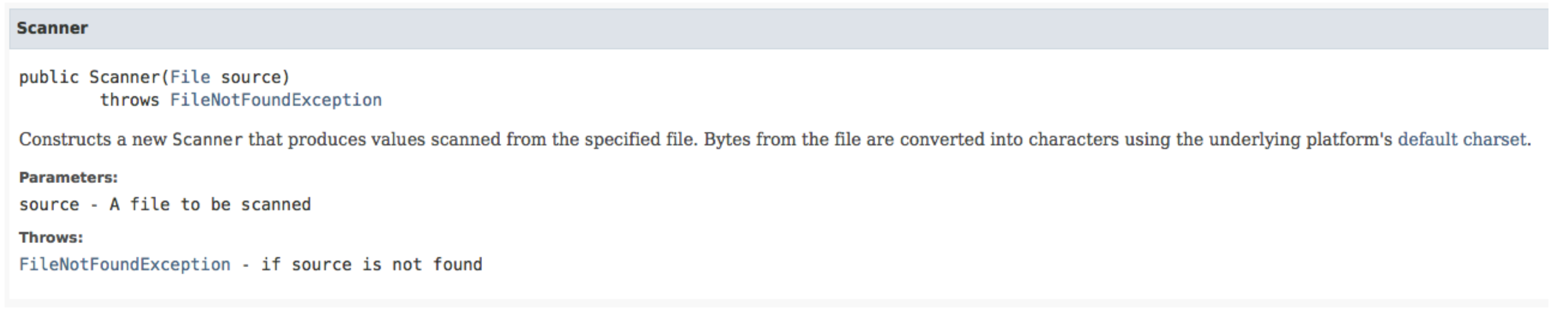
Definition

The Java virtual machine uses a ***call stack*** to track the currently executing method and the method that called the current method. In any Java application the ***main*** method is always the first method on the stack and the last method to leave the stack.

- When an exception is thrown, execution of the current method stops. The JVM looks for a handler in the current method
- If no handler is found, the exception is passed through the call stack until a handler is found
- The application will exit if no handler is found and information including a ***stack trace*** is printed

Checked Exceptions

For certain types of exception, called checked exceptions, Java insists that if a method can throw an exception, you **must** handle it.



Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:
source - A file to be scanned

Throws:
FileNotFoundException - if source is not found

Checked exceptions can be *handled* by:

- Use of try and catch blocks
- Denoting that your method may throw a checked exception:

```
public void myMethod(File f) throws FileNotFoundException {
    Scanner s = new Scanner(f);
    // Other code
}
```

EXAMPLE: ReadandReverse Folder

Example 1 – Read and Reverse

```
import java.util.Scanner;
import java.io.File;

public class Reverse {

  public static void main( String[] args ) {

    try {

      Scanner in = new Scanner( new File(args[0]) );

      while ( in.hasNextLine() ) {
        System.out.println( new StringBuffer(

        in.nextLine()).reverse() );
      }

      in.close();
    }
    catch ( Exception e ) {
      System.out.println( e );
    }
  }
}
```

EXAMPLE: SumDoubles Folder

Example 2 – Sum Doubles

```
import java.util.Scanner;
import java.io.File;

public class Sum {

  public static void main( String[] args ) {

    try {
      Scanner in = new Scanner( new File(args[0]) );

      double total = 0;
      while (in.hasNextDouble()) {
        total += in.nextDouble();
      }

      System.out.printf( "Sum is %.4f%n", total );

      in.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Worked example

Reading a phone book from text file

- We can represent a phone book in a text file by having a particular entry on each line
- Each line would contain a name and a number separated by a space

Worked example

EXAMPLE: PhoneBookLoader Folder

```java
import java.util.Scanner;
import java.io.File;

public class PhoneBookLoader {

    public static PhoneBook loadPhoneBook( String filename ) throws Exception {
        File fileIn = new File( filename );

        try {
            Scanner in = new Scanner( fileIn );
            PhoneBook book = new PhoneBook();

            while( in.hasNextLine() ) {
                // Get a line of the text file
                String line = in.nextLine();

                // Separate the line into name and number
                String[] parts = line.split(",");
                String name = parts[0];
                String number = parts[1];

                // Create an entry and add it to the phone book
                book.add( name, number );
            }
            in.close();

            return book;
        }
        catch( Exception e ) {
            System.out.println( "Problem reading file: " + filename );
            throw e;  // re-raise exception
        }
    }

    public static void main( String[] args ) {
        try {

            String filename = args[0];
            PhoneBook pb = loadPhoneBook( filename );
            System.out.println( pb );
        }
        catch( Exception e ) {
            // Do nothing
        }
    }
}
```

Writing files

- So we've used Scanner(File) to read text files. This is a convenience function provided by **Scanner**
- The package **java.io** provides many more classes for **reading** and **writing text** and **binary** files.
- The **Reader** and **Writer** classes and their subclasses are used to handle **text** files.
- The Reader and Writer classes are **abstract**.
- To perform **text input/output** we use one the **subclasses** of Reader or Writer

Writing files

- **FileReader**: constructs a Reader that can read from a file. Only provides low-level read methods.
- **BufferedReader**: **wraps** around a **FileReader** and provides a high-level **readLine** method.

ReaderTest.java

+

WriterTest.java

Use subclasses of `InputStream` and `OutputStream` classes:

```
Date d = new Date();
ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream( "test.dat" ) );
out.writeObject( d );
out.close();
ObjectInputStream in = new ObjectInputStream( new FileInputStream( "test.dat" ) );
Date sc = (Date)in.readObject();
in.close();
```

Note the object read from file must be *cast* to the correct type

- Java provides a mechanism, called object **serialization** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

- After a serialized object has been written into a file, it can be read from the file and deserialized. That is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

- The JVM allows an object that has been be serialized on one Operating System to be deserialized on an entirely different one.

- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

ReadTest.java

+

WriteTest.java

Binary vs Text

Binary	Text
Compact files	Easy for user to read
Easier to code	More portable
Includes some error checking	More complex to code
Awkward if classes change	Error checking needs to be coded

Class Re-Use

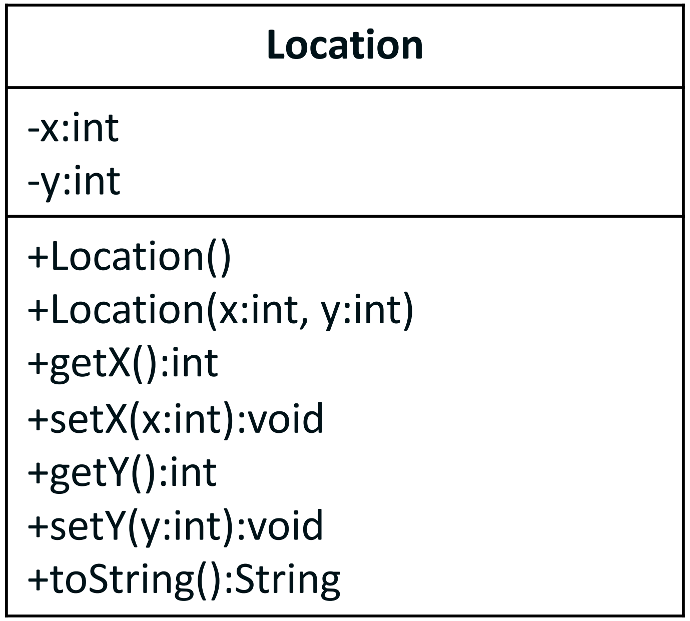
- Lets look again at how **reusing existing** classes, brings a very important dimension to the Java language.
- Namely, the ability to utilise its **Object Oriented Programming (OOP)** approach.
- There are TWO ways in which existing classes can be re-used:

Composition

Inheritance

Composition (aka aggregation)

- We've seen this one already ☺
- You define a **new** class, composed of **existing** classes.
- Consider an example:
  - Suppose we had written a class called `Location`
  - Its class diagram is as follows:



Location

-x:int
-y:int

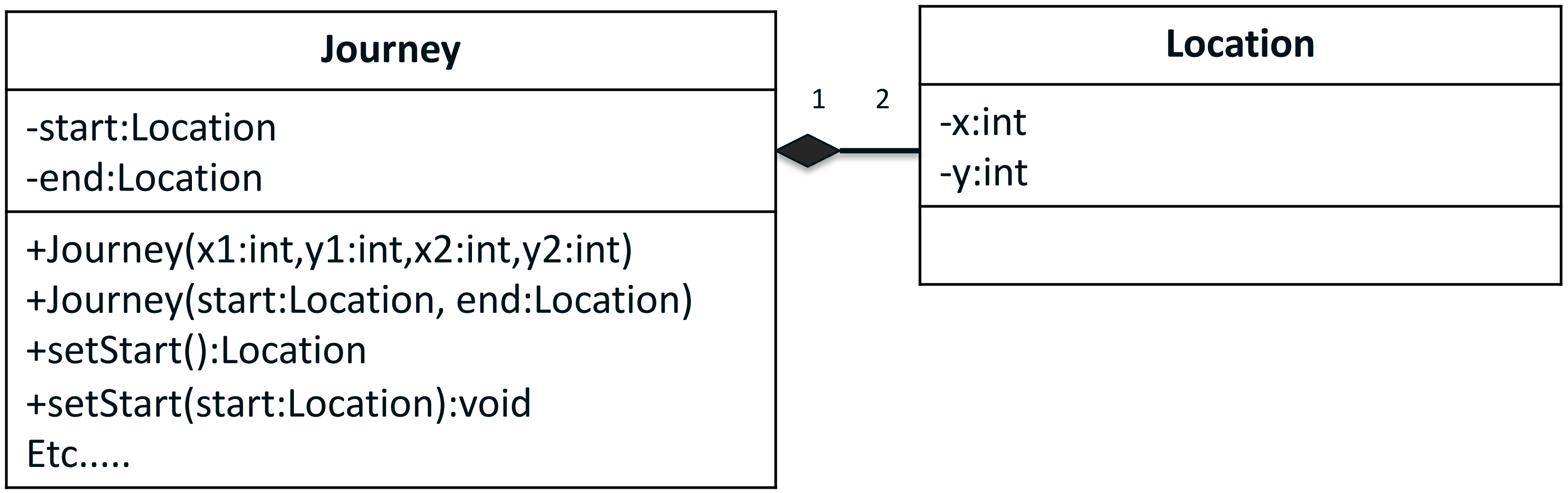
+Location()
+Location(x:int, y:int)
+getX():int
+setX(x:int):void
+getY():int
+setY(y:int):void
+toString():String

Composition (aka aggregation)

An example

- Suppose we decide to write a **class** called **Journey**.
- We could now use our Location class via **composition**.
- We can say:
  - "A Journey is composed of two locations"
  - Or "Journey has two locations"
- **Composition** exhibits a **"has-a"** relationship.
- The following slide shows this using UML notation.
  - Composition is represented as a diamond head pointing to its constituent class(es).

Composition (aka aggregation)

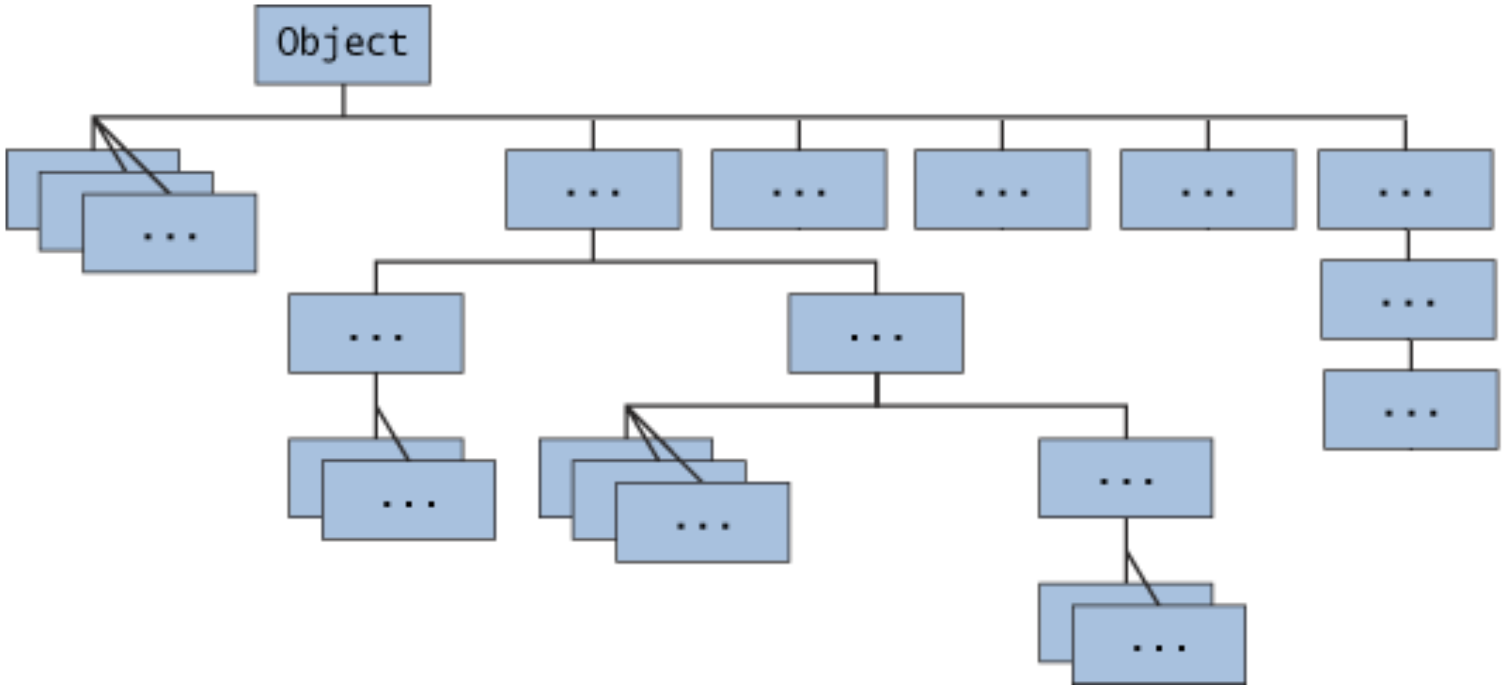

Journey
-start:Location
-end:Location
+Journey(x1:int,y1:int,x2:int,y2:int)
+Journey(start:Location, end:Location)
+setStart():Location
+setStart(start:Location):void
Etc.....
1
2
Location
-x:int
-y:int

Inheritance

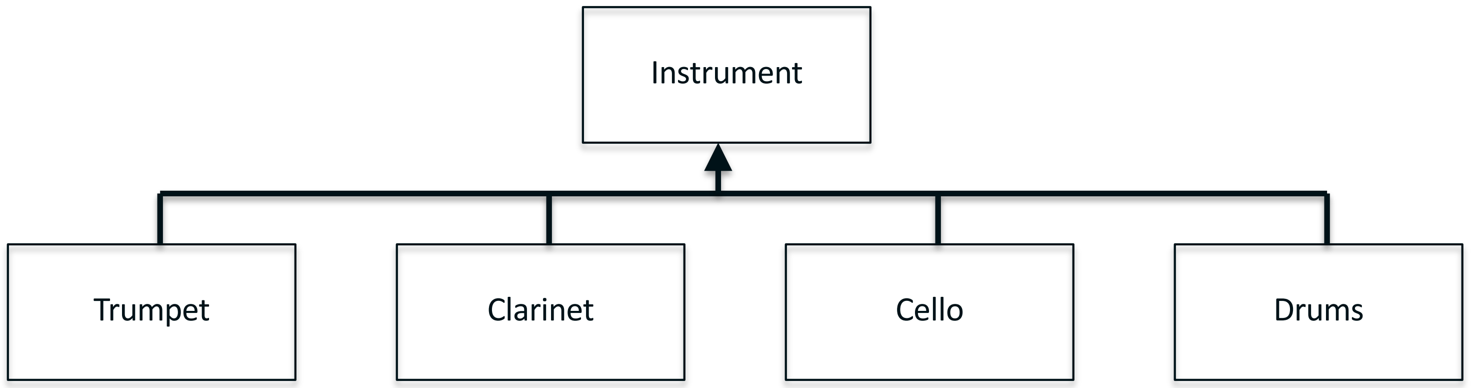
- In OOP, we can organise classes **hierarchically**, in attempt to **avoid duplication** and **reduce redundancy**.
- The notion of inheritance is simple but powerful.
- When you create new classes, if there is an existing class that includes some of the code your want, you can **derive** your **new class** from the **existing one**.
- This allows you to **reuse** the variables and methods of the existing class, without the need to write (and debug!) new code.
- **Inheritance** exhibits a **“is-a”** relationship

Java Platform Class Hierarchy

- The `Object` class in the java.lang package, defines and implements behaviour common to ALL classes (including any you write).
- Many classes derive directly from Object, other classes from those to form a hierarchy of classes:



Hierarchy



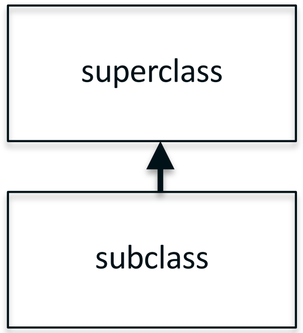
- Classes in the lower hierarchy **inherit** all the **variables** (static attributes) and **methods** (dynamic behaviours) from the higher hierarchies.
- A class in the lower hierarchy is called a **subclass** (or derived, child, extended class).
- A class in the upper hierarchy is called a **superclass** (or base, parent class).
- Separating out all the common variables and methods into the **superclasses**, and leaving the specialised variables and methods in the **subclasses**, allows **redundancy** to be greatly **reduced** or even **eliminated**.
- The common variables do **not** need to be **repeated** in every subclass.

Hierarchy

- Each subclass **inherits ALL** variables and methods from its superclasses, including its immediate parent and any ancestors.
- But please note: a subclass is **NOT** a "subset" of a superclass.
- In fact, subclass is a "superset" of a superclass.
- It is because a subclass inherits all the variables and methods of the superclass; but also, it extends the superclass by providing **MORE** variables and methods.

Defining a subclass

- In Java, you define a subclass using the keyword "**extends**", e.g:
  - Class Trumpet **extends** Instrument {.....}
  - Class Teenager **extends** Human {....}

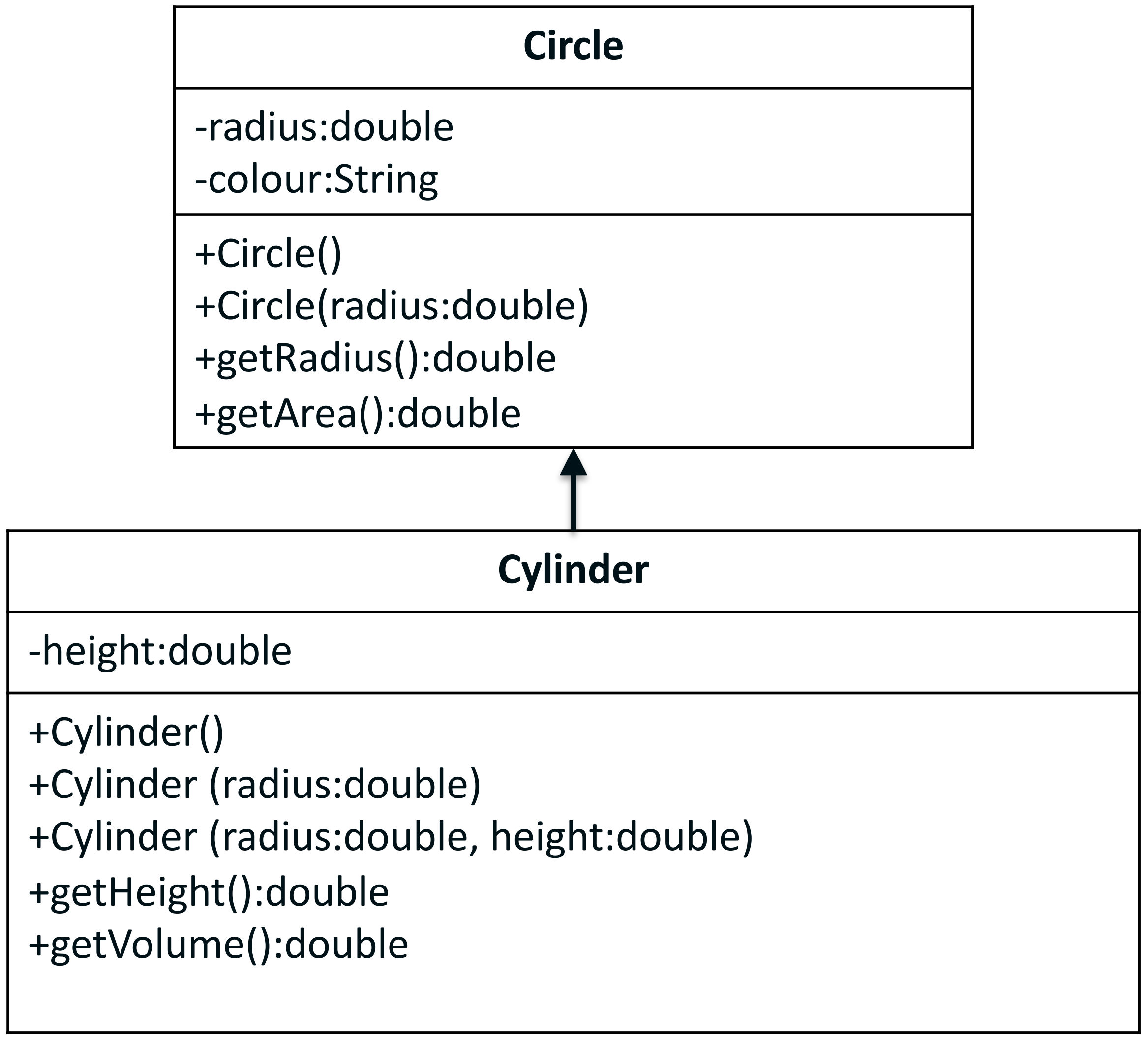


  - Excepting `Object`, which has no superclass, every class has **ONE** and **ONLY ONE** direct superclass (single inheritance).
  - In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

Worked example

- Assuming we have a class called **Circle** (our **superclass**), lets derive a **subclass** called **Cylinder**.
- The Cylinder class will **inherit all member variables** (radius and colour) and **methods** (getRadius(), getArea(), etc..) **from** the **superclass** Circle
- The Cylinder class will further define a **new variable** for **height**, and two public **methods** called **getHeight()** and **getVolume()** and its own **constructors**.
- Code re-use like this is one of the most important properties of OOP. (Why write it again, if it already exists!!).

Inheritance Example



Inheritance Example

- You can use keyword "this" to refer to ***this*** instance inside a class definition.
- One of the main usage of keyword this is to resolve ambiguity.

```
public class Circle {
   double radius;  // Member variable called "radius"
   public Circle(double radius) { // Method's argument also called "radius"
      this.radius = radius;
         // "this.radius" refers to this instance's member variable
         // "radius" resolved to the method's argument.
   }
   ...
}
```

- In the above codes, there are two identifiers called radius - a member variable of the class and the method's argument. This causes **naming conflict**.
- To avoid the naming conflict, you could name the method's argument r instead of radius. However, radius is more approximate and meaningful in this context. Java provides a keyword called ***this*** to **resolve** this **naming conflict**. "this.radius" refers to the member variable; while "radius" resolves to the method's argument.

Using keyword “this”

- this.*varName* refers to *varName* of this instance; this.*methodName*(...) invokes *methodName*(...) of this instance.
- In a constructor, we can use this(...) to call another constructor of this class.
- Inside a method, we can use the statement "return this" to return this instance to the caller.

Keyword "super"

- As we have seen, inside a class definition, you can use the keyword **this** to refer to ***this instance***.
- Similarly, the keyword **super** refers to the **superclass**, which could be the immediate parent or its ancestor.
- The **keyword super** allows the subclass to access superclass' methods and variables within the subclass' definition.
- For example, super() and super(*argumentList*) can be used invoke the superclass' constructor.
- If the subclass overrides a method inherited from its superclass, says getArea(), you can use super.getArea() to invoke the superclass' version within the subclass definition.
- Similarly, if your subclass hides one of the superclass' variable, you can use super.*variableName* to refer to the hidden variable within the subclass definition.

Constructors and Inheritance

- The subclass inherits all the **variables** and **methods** from its superclasses.
- Importantly, the subclass does **NOT inherit** the **constructors** of its superclasses.
- Each class in Java defines its **OWN constructors**.
- In the body of a constructor, you can use super(*args*) to invoke a constructor of its immediate superclass.
- Note that super(*args*), if it is used, must be the *first statement* in the subclass' constructor.
- If it is not used in the constructor, Java compiler automatically insert a super() statement to invoke the no-arg constructor of its immediate superclass. This follows the fact that the parent must be born before the child can be born. You need to properly construct the superclasses before you can construct the subclass.

Default no-arg constructor

- If no constructor is defined in a class, Java compiler automatically create a *no-argument (no-arg) constructor*, that simply issues a super() call, as follows:

```
// If no constructor is defined in a class, compiler inserts this no-arg
constructor
public ClassName () {
   super(); // call the superclass' no-arg constructor
}
```

- The default no-arg constructor will not be automatically generated, if one (or more) constructor was defined. In other words, you need to define no-arg constructor explicitly if other constructors were defined.
- If the immediate superclass does not have the default constructor (it defines some constructors but does not define a no-arg constructor), you will get a compilation error in doing a super() call. Note that Java compiler inserts a super() as the first statement in a constructor if there is no super(args).

Polymorphism

- The word "*polymorphism*" means "*many forms*".
- It comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*).
- E.g. carbon exhibits polymorphism because it can be found in more than one form:
  - Graphite
  - Diamond
- Each of the forms has it own distinct properties.

Substitutability

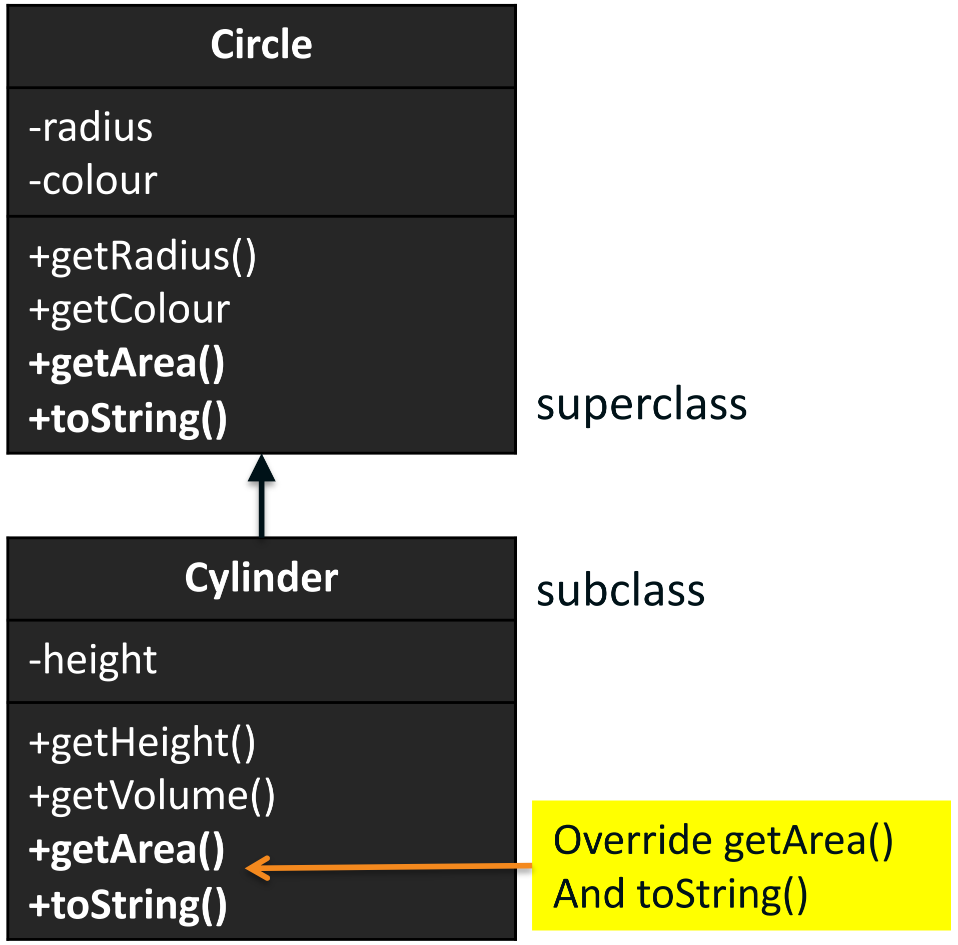
- A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass).
- This means that a subclass object can do whatever its superclass can do. As a result, we can *substitute* a subclass instance when a superclass instance is expected, and everything shall work fine. This is called *substitutability*.
- In last lectures example of Circle and Cylinder: Cylinder is a subclass of Circle. We can say that Cylinder "*is-a*" Circle (actually, it "*is-more-than-a*" Circle). Subclass-superclass exhibits a so called "*is-a*" relationship.

Substitutability

- Via *substitutability*, we can create an instance of Cylinder, and assign it to a Circle (its superclass) reference, as follows:

```
// Substitute a subclass
instance to its superclass
reference
Circle c1 = new
Cylinder(5.0);
```

- You can invoke all the methods defined in the Circle class for the reference c1, (which is actually holding a Cylinder object), e.g. c1.getRadius() and c1.getColor(). This is because a subclass instance possesses all the properties of its superclass.



Substitutability

- However, you cannot invoke methods defined in the Cylinder class for the reference c1, e.g. c1.getHeight() and c1.getVolume(). This is because c1 is a reference to the Circle class, which does not know about methods defined in the subclass Cylinder.
- c1 is a reference to the Circle class, but holds an object of its subclass Cylinder. The reference c1, however, *retains its internal identity*. In our example, the subclass Cylinder overrides methods getArea() and toString(). c1.getArea() or c1.toString() invokes the *overridden* version defined in the subclass Cylinder, instead of the version defined in Circle. This is because c1 is in fact holding a Cylinder object internally.

Substitutability Summary

1. A subclass instance can be assigned (substituted) to a superclass' reference.
2. Once substituted:
   - we can invoke methods defined in the superclass
   - we cannot invoke methods defined in the subclass.
3. However, if the subclass overrides inherited methods from the superclass, the subclass (overridden) versions will be invoked.

Upcasting

Upcasting a Subclass Instance to a Superclass Reference

- Substituting a subclass instance for its superclass is called "*upcasting*".
- This is because, in a UML class diagram, subclass is often drawn below its superclass.
- Upcasting is *always safe* because a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do.
- The compiler checks for valid upcasting and issues error "incompatible types" otherwise.

Upcasting Example

Example

```
// Compiler checks to ensure that R-value is a
subclass of L-value
Circle c1 = new Cylinder();

// Compilation error: incompatible types
Circle c2 = new String();
```

Downcasting

Downcasting a Substituted Reference to Its Original Class

- You can revert a substituted instance back to a subclass reference.
- This is called "*downcasting*".

Example

```
// upcast is safe
Circle c1 = new Cylinder(5.0);

// downcast needs the casting operator
Cylinder aCylinder = (Cylinder) c1;
```

Downcasting

- Downcasting requires *explicit type casting operator* in the form of prefix operator (*new-type*).
- Downcasting is not always safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass.
- A subclass object can be substituted for its superclass, but the reverse is not true.

Downcasting

- Compiler may not be able to detect error in explicit cast, which will be detected only at runtime.

Example

```
Circle c1 = new Circle(5);
Point p1 = new Point();

c1 = p1; // compilation error: incompatible types (Point is not
a subclass of Circle)

c1 = (Circle)p1; // runtime error:
java.lang.ClassCastException: Point cannot be casted to Circle
```

instanceOf Operator

- Java provides a binary operator called instanceof which returns true if an object is an instance of a particular class.

Syntax

```
                <Object> instanceof <Class>
```

Example

```
Circle c1 = new Circle();
System.out.println(c1 instanceof Circle);   // true

if (c1 instanceof Circle) { ...... }
```

instanceOf Operator

Note

- An instance of subclass is also an instance of its superclass.

Example

```
Circle c1 = new Circle(5);
Cylinder cy1 = new Cylinder(5, 2);
System.out.println(c1 instanceof Circle);     // true
System.out.println(c1 instanceof Cylinder);   // false
System.out.println(cy1 instanceof Cylinder); // true
System.out.println(cy1 instanceof Circle);    // true

Circle c2 = new Cylinder(5, 2);
System.out.println(c2 instanceof Circle);     // true
System.out.println(c2 instanceof Cylinder);   // true
```

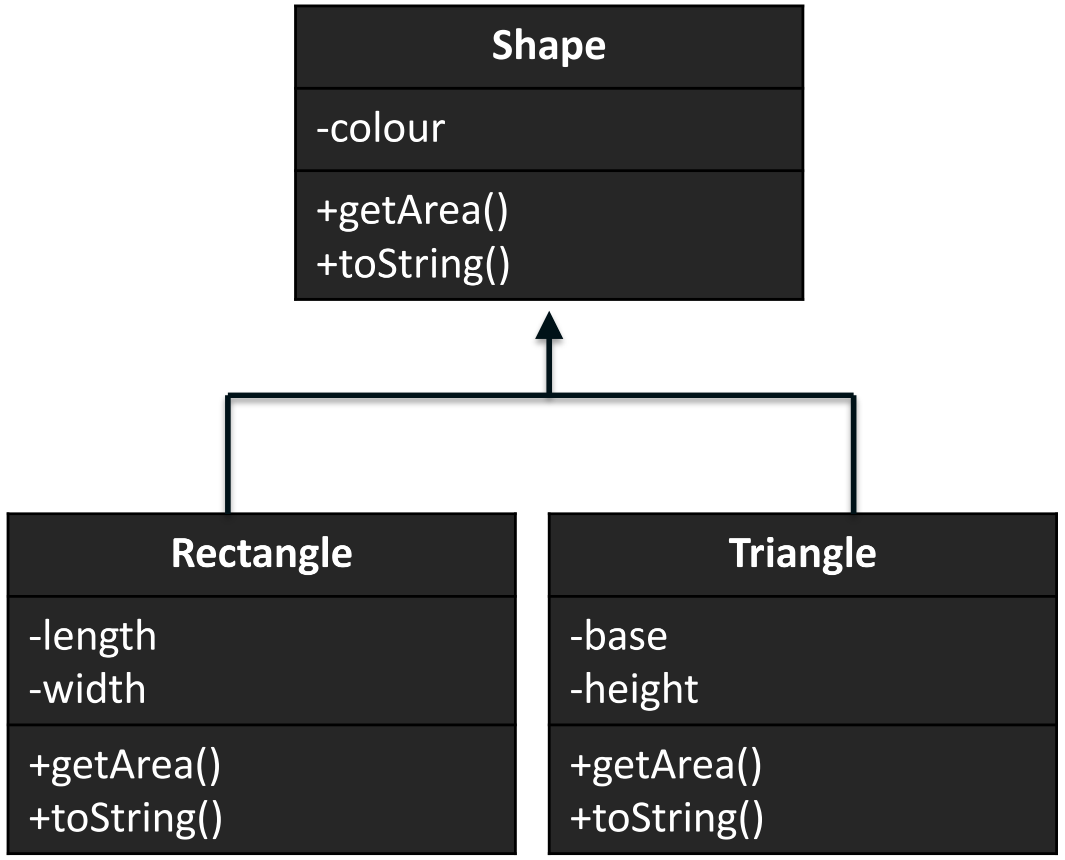
Polymorphism Summary

1. A subclass instance processes all the attributes and operations of its superclass. When a superclass instance is expected, it can be substituted by a subclass instance. In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - it is called substitutability.
2. If a subclass instance is assigned to a superclass reference, you can invoke the methods defined in the superclass only. You cannot invoke methods defined in the subclass.
3. However, the substituted instance retains its own identity in terms of overridden methods and hiding variables. If the subclass overrides methods in the superclass, the subclass's version will be executed, instead of the superclass's version.

Polymorphism Example

- Polymorphism is very powerful in OOP to *separate the interface and implementation* so as to allow the programmer to *program at the interface* in the design of a *complex system*.
- Lets go back to our shape example. Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. We should design a superclass called Shape, which defines the public interface (or behaviors) of all the shapes.
- E.g., we would like all the shapes to have a method called getArea(), which returns the area of that particular shape.

Polymorphism Example



Polymorphism Example – Shape.java

```
// Define superclass Shape
public class Shape {
   // Private member variable
   private String colour;

   // Constructor
   public Shape (String colour) {
      this.colour = colour;
   }

   @Override
   public String toString() {
      return "Shape of colour=\"" + colour + "\"";
   }

   // Problem - all shapes must have a method called getArea()
   public double getArea() {
      System.err.println("Shape unknown! Cannot compute area!");
      return 0;    // Need a return to compile the program
   }
}
```

Polymorphism Example

- NOTE: we have a problem on writing the getArea() method in the Shape class.
- The area obviously cannot be computed unless the actual shape is known.
- For now, we shall print an error message. We will come back in a few slides to see how this problem can be resolved.
- Moving on with our example:
  - We can now derive subclasses, such as Triangle and Rectangle, from the superclass Shape.

Polymorphism Example

Polymorphism Example – Rectangle.java

```java
// Define Rectangle, subclass of Shape
public class Rectangle extends Shape {
   // Private member variables
   private int length;
   private int width;

   // Constructor
   public Rectangle(String colour, int length, int width) {
      super(colour);
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle of length=" + length + " and width=" + width + ",
           subclass of " + super.toString();
   }

   @Override
   public double getArea() {
      return length*width;
   }
}
```

Polymorphism Example – Triangle.java

```
// Define Triangle, subclass of Shape
public class Triangle extends Shape {
   // Private member variables
   private int base;
   private int height;

   // Constructor
   public Triangle(String colour, int base, int height) {
      super(colour);
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle of base=" + base + " and height=" + height + ",
           subclass of " + super.toString();
   }

   @Override
   public double getArea() {
      return 0.5*base*height;
   }
}
```

Polymorphism Example – TestShape.java

- We can now create references of Shape, and assign them instances of subclasses:

```java
// Test program for Shape and its subclasses
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());
   }
}
```

- The beauty of this code is that *all the references are from the superclass* (i.e., *programming at the interface level*). You could instantiate a different subclass instance, and the code still works.
- You could easily extend your program by adding in more subclasses, such as Circle, Square, etc...

Shape – Back To The Problem – getArea()

- We know the definition of the Shape class has a problem.
- If someone instantiates a Shape object and invokes its getArea() method, our program breaks:

```
public class TestShape {
   public static void main(String[] args) {
      // Constructing a Shape instance poses problem!
      Shape s3 = new Shape("green");
      System.out.println(s3);
      System.out.println("Area is " + s3.getArea());
   }
}
```

- This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation.
- We do not want anyone to instantiate a Shape instance.
- **Solution** -> using the so-called abstract class.

Abstract Classes

- An abstract method is a method with only a signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method.
- For example, in the Shape class, we can declare three abstract methods getArea(), draw(), as follows::

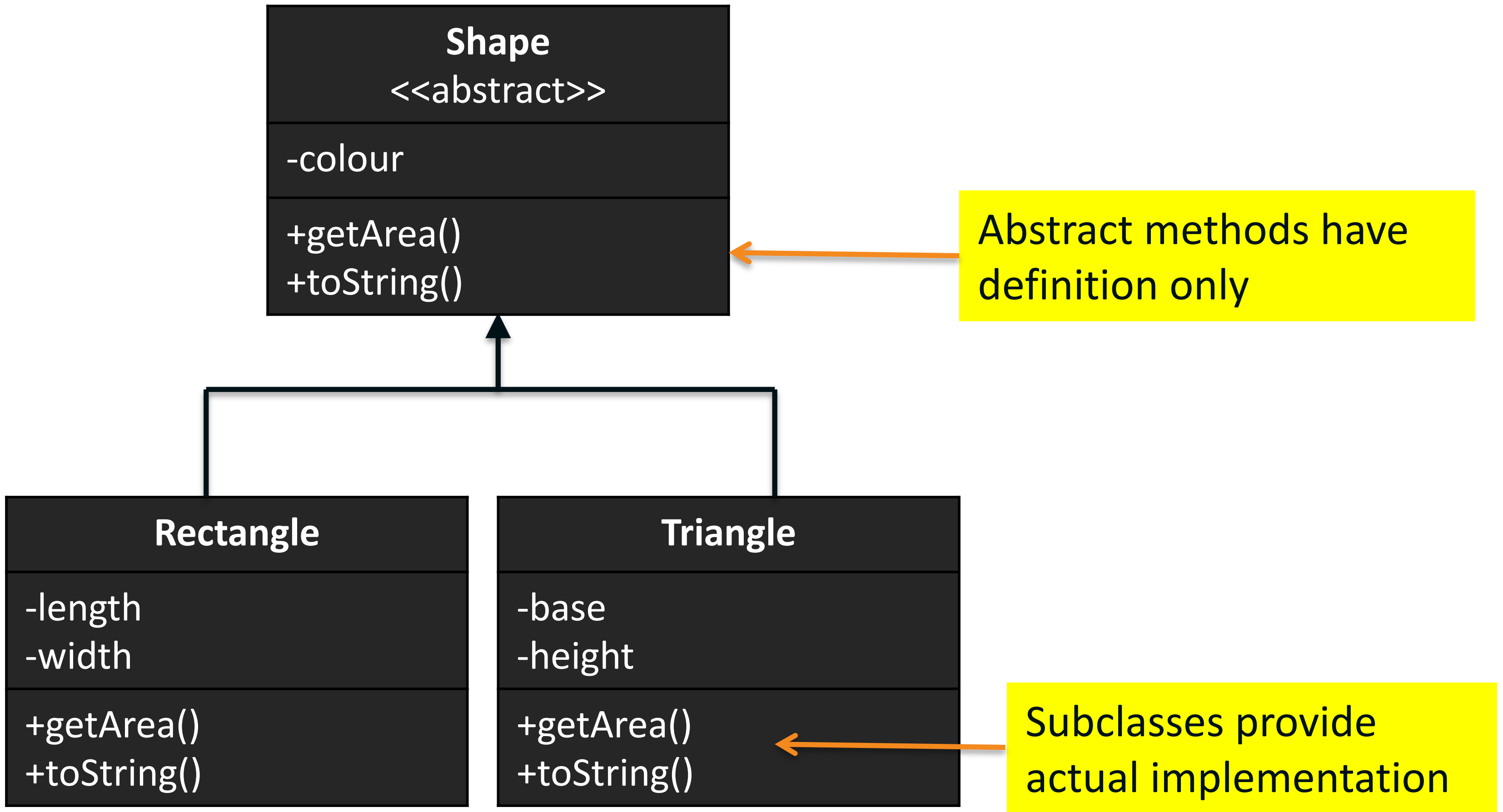
```
abstract public class Shape {
   ......
   public abstract double getArea();
   public abstract void draw();
}
```

- Implementation of these methods is not possible in the Shape class, as the actual shape is not yet known. (How to compute the area if the shape is not known?).
- Implementation of these abstract methods will be provided later once the actual shape is known.
- These abstract methods cannot be invoked because they have no implementation.

Abstract Classes

- A class containing one or more abstract methods is called an abstract class.
- An abstract class must be declared with a class-modifier abstract.
- Lets now go back to our Shape class and rewrite it as an abstract class, containing an abstract method for getArea().

Shape Example – Abstract Class



Abstract Class – Shape.java

```
abstract public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

   // All Shape subclasses must implement a method called getArea()
   abstract public double getArea();
}
```

Abstract Classes

- Our abstract class is *incomplete* in its definition, since the implementation of its abstract methods are missing. Therefore, our abstract class *cannot be instantiated*. In other words, we cannot create instances from our abstract class (otherwise, you will have an incomplete instance with missing method's body).
- To use our abstract class, we have to derive a subclass from the abstract class. In the derived subclass, we have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is then complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.)
- This property of the abstract class solves our earlier problem. In other words, you can create instances of the subclasses such as Triangle and Rectangle, and upcast them to Shape (so as to program and operate at the interface level), but you cannot create instances of Shape, which avoids the pitfall that we faced.

Abstract Classes Summary

- In summary, an abstract class provides *a template for further development*.
- The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses.
- E.g., in the abstract class Shape, you can define abstract methods such as getArea() and draw(). No implementation is possible because the actual shape is not known. However, by specifying the signature of the abstract methods, all the subclasses are *forced* to use these methods' signature. The subclasses should provide the proper implementations.
- Coupled with polymorphism, you can upcast subclass instances to Shape, and program at the Shape level, i,e., program at the interface.
- The separation of interface and implementation enables better software design, and ease in expansion.
- E.g., Shape defines a method called getArea(), which all the subclasses must provide the correct implementation. You can ask for a getArea() from any subclasses of Shape, the correct area will be computed.
- Furthermore, you application can be extended easily to accommodate new shapes (such as Circle or Square) by deriving more subclasses.

Abstract Classes Summary

- **Rule of Thumb:** Program at the interface, not at the implementation. (That is, make references at the superclass; substitute with subclass instances; and invoke methods defined in the superclass only.)

Notes:

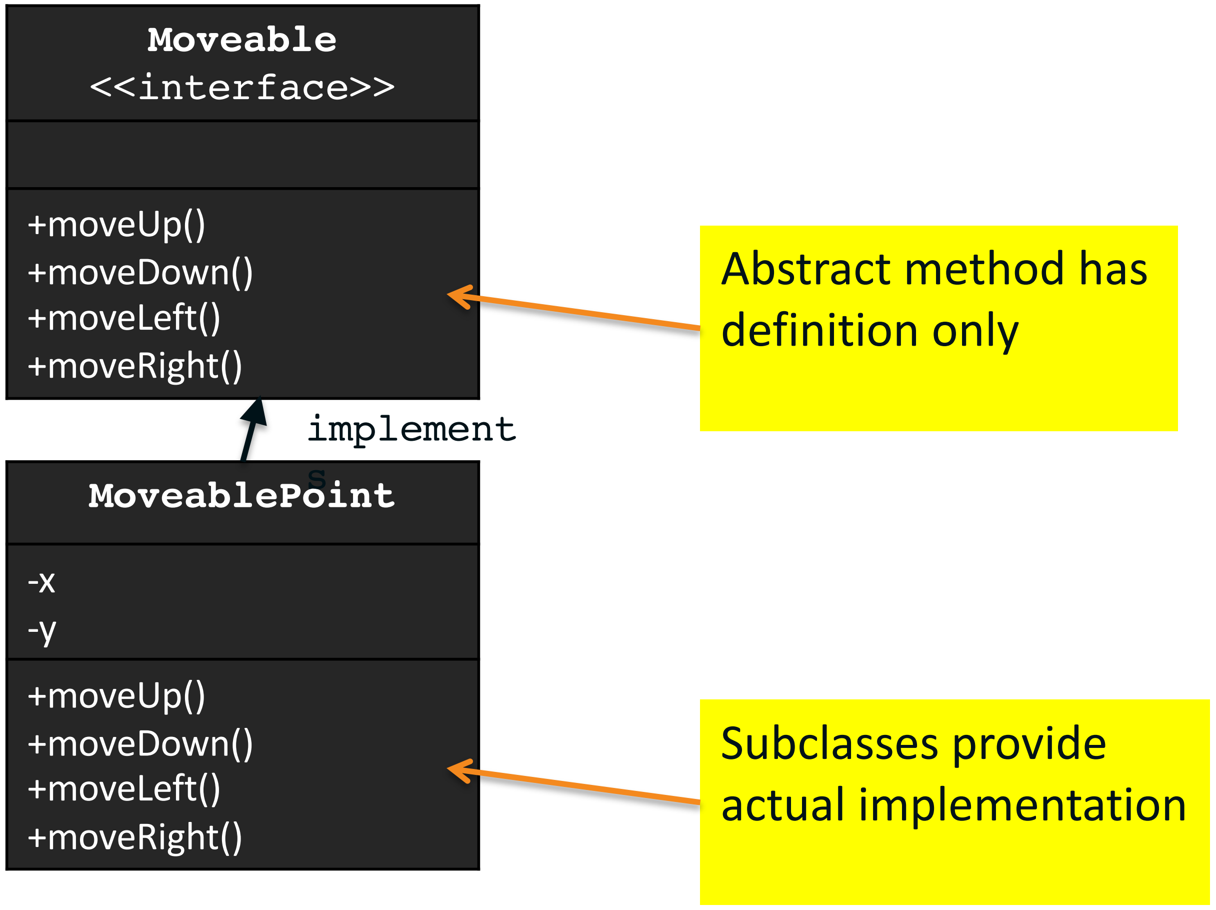
- An abstract method cannot be declared final, as a final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendent before it can be used.
- An abstract method cannot be private (which generates a compilation error). This is because private methods are not visible to the subclass and thus cannot be overridden.

The `interface`

- A Java interface is a *100% abstract superclass* which defines a set of methods that its subclasses must support.
- An interface only contains public *abstract methods* (methods with signature and no implementation) and possibly *constants* (public static final variables).
- You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes).
- The keyword public and abstract are not needed for its abstract methods as they are mandatory.
- An interface is a *contract* for what the classes can do.
- It, however, does not specify how the classes should do it.
- **Interface Naming Convention:** Use an adjective (typically ends with "able") consisting of one or more words. Each word shall be initial capitalised (camel-case). For example, Serialisable, Extenalisable, Movable, Clonable, Runnable, etc.

Interface Example

- Suppose that our application involves many objects that can move.
- We could define an interface called movable, containing the signatures of the various movement methods.



Interface Moveable

```
public interface Movable {
    // abstract methods to be implemented by the subclasses
    public void moveUp();
    public void moveDown();
    public void moveLeft();
    public void moveRight();
}
```

- Similar to an abstract class, an interface cannot be instantiated; because it is incomplete (the abstract methods' body is missing).
- To use an interface, again, you must derive subclasses and provide implementation to all the abstract methods declared in the interface.
- The subclasses are only then complete and can be instantiated.

Keyword “implements”

- To derive subclasses from an interface, a new keywords "implements”.
- This is instead of "extends" for deriving subclasses from an ordinary class or an abstract class.
- It is important to note that the subclass implementing an interface needs to override ALL the abstract methods defined in the interface; otherwise, the subclass cannot be compiled

Example

Implementing Moveable – MovablePoint.java

```
public class MovablePoint implements Movable {
   // Private member variables
   private int x, y;   // (x, y) coordinates of the point

   // Constructor
   public MovablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   @Override
   public String toString() {
      return "Point at (" + x + "," + y + ")";
   }

   // Implement abstract methods defined in the interface Movable
   @Override
   public void moveUp() {
      y--;
   }
```

Implementing Moveable – MovablePoint.java

```
    @Override
    public void moveDown() {
       y++;
    }

    @Override
    public void moveLeft() {
       x--;
    }

    @Override
    public void moveRight() {
       x++;
    }
}
```

- Other classes in the application can similarly implement the Movable interface and provide their own implementation to the abstract methods defined in the interface Movable.

Example

Testing – TestMovable.java

- We can also upcast subclass instances to the Movable interface, via polymorphism, similar to an `abstract` class.

```
public class TestMovable {
   public static void main(String[] args) {
      Movable m1 = new MovablePoint(5, 5);  // upcast
      System.out.println(m1);
      m1.moveDown();
      System.out.println(m1);
      m1.moveRight();
      System.out.println(m1);
   }
}
```

Implementing Multiple `interfaces`

- As already mentioned, Java supports only *single inheritance*.
- That is, a subclass can be derived from one and only one superclass.
- Java does not support *multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses.
- A subclass, however, can implement more than one interfaces. This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces.
- In other words, Java indirectly supports multiple inheritances via implementing multiple interfaces.

```
// One superclass but implement multiple interfaces
public class Circle extends Shape implements Movable, Displayable {
   .......
}
```

Interface Syntax

interface Syntax

```
[public|protected|package] interface interfaceName
[extends superInterfaceName] {
   // constants
   static final ...;

   // abstract methods' signature
   ...
}
```

- All methods in an interface are public and abstract (default). You cannot use any other access modifier such as private, protected and default, or modifiers such as static or final.
- All variables should be public, static and final (default).
- An interface may "extend" from a super-interface.

Summary

- An interface is a *contract* (or a protocol, or a common understanding) of what the classes can do.
- When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface.
  - The interface defines a set of common behaviors.
  - The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors.
- This allows you to program at the interface, instead of the actual implementation.
- One of the main uses of an interface is provide a *communication contract* between two Objects.
- If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely.
- In other words, two Objects can communicate based on the contract defined in the interface, instead of their specific implementation.