Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Whatever
{
   // contents
}
```

Abstract Example

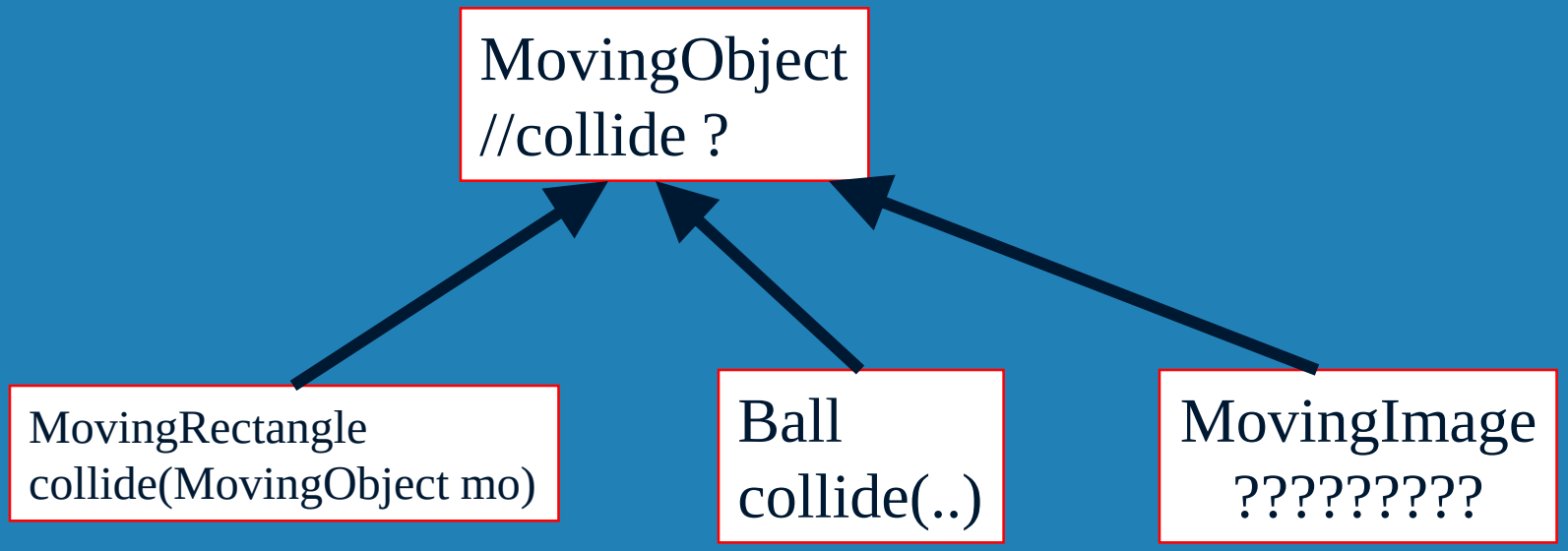
- A good example is the MovingObject class. MovingObject is just an abstract or synthetic concept to help us capture commonalities.

- To make sure nobody creates an instance of class MovingObject, we need to declare it abstract.

Question: Is there another way of preventing others from creating objects of type MovingObject?

Abstract Example

- Let's say you want the MovingObjects to be able to collide with each other, but cannot define a collide method since the outcome of collision depends on specific object



Abstract Classes

- An abstract class often contains abstract methods with no definitions
- In addition to forcing sub-classes to override to become concrete classes, it enables one to write polymorphic methods
- An abstract class typically contains non-abstract methods (with bodies), which can even call abstract methods

  (a framework to build upon)
- A class declared as abstract does not need to contain abstract methods

Vehicle example

```
public abstract class Vehicle {
    private Position position;
    public getPosition() { return position; }
    public abstract void start();
    public abstract void move();
    public abstract void turnLeft();
    public abstract void turnRight();
    public abstract void stop();
    public void goto(Position pos) {
        start();
        if (position.getX() > pos.getX())
            turnLeft();
  .  ……
```

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate

AbstractMethods

- to make sure every MovingObject has a collide() method, you can declare an abstract MovingObject.collide() method without an implementation, to be provided by more specific sub-classes

```
public class MovingObject {
…..
    public abstract void collide (MovingObject other);
……..
}
```

A framework for sequential range search

- Let's say that we want to search a list of objects to find all objects having values between start and end objects.
- Before starting an implementation, we have to ask the following questions:
  - what do we have to know about our objects? in other words, what is the proper abstraction we should make of our objects?
  - What do we have to know about the list that stores our objects?
- the idea is to know as little about our objects as possible to write more general code, the more we know about our actors, the more restrictive our methods will get
- lets create two classes, Searchable for our objects, List for our list, that can be sub-classed later.

Searchable Class

- we don't have to know the actual value of an object. for example a method like "public int getValue()" would be too restrictive, what if our objects are strings?
- all we need is to know is, if an object is greater or less than another :

```
public abstract class Searchable {
    public boolean isLess(Searchable other);
    public boolean isGreater(Searchable other);
}
```

- Any class that extends Searchable can be used by our method
- Now lets look at a List class that will contain Searchable objects

List class

- as a part of being as general as possible, we don't want to just implement a list class. we want our search method to work on any structure that has the characteristics of a list. A specific implementation can inherit from our abstract List class.
- Here are the things we need from a typical list
  - a method that tells us if we have more elements in the list
  - a method to get the next element
  - a method to advance to the following element
  - a method to go to the first element
- note that we don't care how the elements got inserted into the list, that is irrelevant to our searching

List abstract class

```
public abstract class List {
    public boolean hasNext();
    public Searchable getNext();
    public void advance();
    public void reset(); // to go to first element
}
```

- note that a List doesn't have to know what it contains, we could change the getNext() to return an Object reference.

the search method

```
public static ArrayList rangeSearch(List list, Searchable
    start, Searchable end) {
    ArrayList result = new ArrayList();
    while (list.hasNext()) {
            Searchable current = list.getNext();
            if (current.isGreater(start) && current.isLess(end))
                    result.add(current);
            list.advance();
    }
    return result;
}
```

- could have made it a method of List class

Example usage of search

```
public class YearAndMonth extends Searchable {
    public int year, month;
    public boolean isLess(Searchable other) {
        YearAndMonth ym = (YearAndMonth) other;
        if (ym.year > year)
            return true;
        else if (ym.year < year)
            return false;
        if ym.month > month
            return true;
        return false;
    }
    public boolean isGreater(Searchable other) { …. }
```

A List example

```
public class VectorList extends List {
    private ArrayList v = new ArrayList();
    private int next = 0;
    public boolean hasNext() {
        return (next < v.size());
    }
    public Searchable getNext() {
        return v.get(next);
    }
    public void advance() {
        next++;
    }
    public void reset() {
        next = 0;
    }
    .....
```

Multiple Roles

- What if a class can satisfy several abstractions ? For example a car can be seen/viewed as a MovingObject(velocity), personal property (date of purchase, owner name ..), a rental item (cost per day,..), product (manufacture date, serial number..), a vehicle (number of seats) ….
- In order to inherit implementation, can choose one parent
- how can we fit our Car objects into different situations even though it has only one line of ancestors?

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish, as a formal contract, a set of methods that a class will implement

Interfaces

interface is a reserved word

```
public interface Product
{
    public static final int USA = 1;
    public static final int TURKEY = 90;
    …..

    public int getCountry();
    public String getSerialId();
    public int getManufactureYear();
    public void setQualityTester(Employee e);
    public String getModel();

}
```

None of the methods in an interface are given a definition (body)

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface

Interfaces

```
public class Car extends Vehicle implements Product
{

    public String getModel()
    {
        ….
    }

        ...

    // etc.
}
```

implements is a reserved word

Each method listed in Product is given a definition

Car-Product

- Now an object of type Car can also be regarded as a product:

```
public void displayProductInfo(Product p) {
…..
}
….
Car c = new Car(…);
Product p = c; // nothing happens to the actual object here
displayProductInfo(c);
```

- Interfaces cannot be instantiated but can be used as parameter and reference types

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the implements clause
- The class must implement all methods in all interfaces listed in the header

```
class Car extends Vehicle implements Product,
   MovingObject, Property, RentalItem
{
    // all methods of all interfaces
}
```

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

Searchable and List

- the abstract classes Searchable and List defined earlier are good candidates for being interfaces.
- the name List could be misleading, since our abstraction doesn't care about the structure itself, it just cares about iterating through elements of any structure.
- the Searchable name is not very good, too. What about a method that finds the maximum valued object? It can also use Searchable objects. What we really care is that the objects must be compared to each other.
- often the logic of isLess() and isGreater() is closely related, why have two separate methods?

Interfaces

- The Java standard class library contains many helpful interfaces
- The `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects (similar to Searchable)
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order
- The `Iterator` interface contains methods that allow the user to move easily through a collection of objects (similar to our List interface)

The Comparable Interface

- The `Comparable` interface provides a common mechanism for comparing one object to another

```
if (obj1.compareTo(obj2) < 0)
    System.out.println ("obj1 is less than obj2");
```

- The result is negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`
- When a programmer writes a class that implements the `Comparable` interface, it should follow this intent
- It's up to the programmer to determine what makes one object less than another

The Iterator Interface

- The `Iterator` interface provides a means of moving through a collection of objects, one at a time
- The `hasNext` method returns a boolean result (true if there are items left to process)
- The `next` method returns the next object in the iteration
- The `remove` method removes the object most recently returned by the `next` method
- A class can change its data structures, but as long as it is accessed by Iterator interface, no problem

Iterator Example

```
public String toString() {
        String report = "";
        for (int cd = 0; cd < collection.size (); cd++) {
                CD currentcd = (CD) collection.get (cd);
                report += currentcd.toString() + "\n";
        }
        // or ….
        Iterator it = collection.iterator ();
        while (it.hasNext ()) {
                CD currentcd = (CD) it.next ();
                report += currentcd.toString() + "\n";
        }
        return report;
 }
```

Iterator Example

```
public class Database {
    private ArrayList items;

   public Database()  {
        items = new ArrayList();
    }

   public void addItem(Item theItem)  {
        items.add(theItem);
    }

   public String toString() {
       String result = "";
        for(Iterator iter = items.iterator(); iter.hasNext(); )  {
            Item item = (Item)iter.next();
            result += item.toString();
        }
        return result;
    }
}
```

Flexibility of Interfaces

- When your program expects an object of a concrete class, you can only use types of objects that are descendants of that concrete class.
- If you use interfaces, then regardless of its location in inheritance hierarchy, any object can be used
- Therefore it is more flexible, for a method for example, to accept an Interface type rather than a concrete type. This way, it is possible to change underlying object without changing your code.

Extending Interfaces

- An Interface can extend other interfaces.

```
interface ABC extends A, B, C
{
....
}
```

Call-back methods

- Lets say that your class needs to be notified when something happens by some other class (like an alarm clock, barometer, thermometer, a window …)
- It is clear that we should have two classes, Let's say a thermometer and some other class communicating with each other.
- What should a thermometer class look like?
- What does a thermometer need to know about the object that is going to be "called-back" ?

Thermometer

- It makes sense if the thermometer called some method of that object when the temperature exceeds some given limit.

```
public class Temperature {
….
    if (currentTemp > limit)
            object.handleTemperature(currentTemp);
```

- How does a Temperature know which object(s) to notify?

Thermometer

- we could get the object from the constructor or via a method :

```
public class Temperature {
    public Temperature(TemperatureListener tl, int  limit)

    public addListener(TemperatureListener tl, int limit)
…
}
```

- What about TemperatureListener ?

Thermometer

- it makes sense to define TemperatureListener to be an interface, since any type of object should be allowed to be a listener to be most flexible :

```
interface TemperatureListener {
    void handleTemperature(int currentTemp);
}
```