Abstract Classes

- An *abstract class* in a class hierarchy represents a generic concept

  - Common elements in a hierarchy that are too generic to instantiate
  - Cannot be instantiated

- `abstract` on the class header:

  ```java
  public abstract class Product {
  // contents
  }
  ```
Abstract Classes

- abstract classes typically have:
  - abstract methods with no definitions (like an interface)
  - probably also non-abstract methods with full definitions
- Does not have to contain abstract methods -- simply declaring it as abstract makes it so
- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
Abstract Classes

abstract class A1 {
    abstract void m1();
    abstract String m2();
}

class C1 extends A1 {
    void m1() { System.out.println("C1-m1"); }
    String m2() { return "C1-m2"; }
}

abstract C2 extends A1 {
    void m1() { System.out.println("C2-m1"); }
}

→ C2 must be abstract, because it does not implement the abstract method m2.
Abstract Classes

- Abstract methods cannot be defined as `final` or `static`
  - `final` cannot be overridden (contradiction!)
  - `static` could be invoked by just using the name of the class – can’t invoke it with no implementation
Implementing Interfaces

- An interface can be implemented by multiple classes.
- Each implementing class can provide their own unique versions of the method definitions.

```java
interface I1 {
    void m1();
}

class C1 implements I1 {
    public void m1() { System.out.println("Implementation in C1"); }
}

class C2 implements I1 {
    public void m1() { System.out.println("Implementation in C2"); }
}
```
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

```java
class ManyThings implements interface1, interface2 {
    // all methods of both interfaces
}
```
Implementing More Than One Interface

interface I1 {
    void m1();
}

interface I2 {
    void m2();
    void m3();
    // C must implement all methods in I1 and I2.
}

class C implements I1, I2 {
    public void m1() { System.out.println("C-m1"); }
    public void m2() { System.out.println("C-m2"); }
    public void m3() { System.out.println("C-m3"); }
}
Resolving Name Conflicts Among Interfaces

- Since a class may implement more than one interface, the names in those interfaces may collide.
- To solve name collisions, Java uses a simple mechanism.
- Two methods that have the same name will be treated as follows in Java:
  - If they are different signature, they are considered to be overloaded.
  - If they have the same signature and the same return type, they are considered to be the same method and they collapse into one.
  - If they have the same signature and different return types, a compilation error will occur.
interface I1 {
    void m1();
    void m2();
    void m3();
}
interface I2 {
    void m1(int a);
    void m2();
    int m3();
}
class C implements I1, I2 {
    public void m1() { ... }  // implementation of m1 in I1
    public void m1(int x) { ... }  // implementation of m1 in I2
    public void m2() { ... }  // implementation of m2 in I1 and I2
}

There will be a compilation error for m3.
Inheritance Relation Among Interfaces

- Same as classes, interfaces can hold inheritance relation among them

```
interface I2 extends I1 { ... }
```

- Now, I2 contains all abstract methods of I1 plus its own abstract methods.

- The classes implementing I2 must implement all methods in I1 and I2.
Interfaces as Data Types

- Interfaces (same as classes) can be used as data types.
- Different from classes: We cannot create an instance of an interface.

```java
interface I1 { ... }
class C1 implements I1 { ... }
class C2 extends C1 { ... }
```

// a variable can be declared as type I1
I1 x;

- A variable declared as I1, can store objects of C1 and C2.
  - More later...
The Iterator Interface

- An iterator is an object that provides a means of processing a collection of objects one at a time

- An iterator is created formally by implementing the `Iterator` interface, which contains three methods

- 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
The Iterator Interface

- By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators.

- The programmer must decide how best to implement the iterator functions.

- Once established, the for-each version of the `for` loop can be used to process the items in the iterator.
Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (Comparable).
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways.
- Interfaces are a key aspect of object-oriented design in Java.
When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways through different implementations
  - These subclasses extend the same Abstract class and provide different implementations for the abstract methods
- Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.
Why do we use Interfaces? Reason #1

- To reveal an object's programming interface (functionality of the object) without revealing its implementation
  - This is the concept of encapsulation
  - The implementation can change without affecting the caller of the interface
  - The caller does not need the implementation at the compile time
    - It needs only the interface at the compile time
    - During runtime, actual object instance is associated with the interface type
Why do we use Interfaces? Reason #2

- To have unrelated classes implement similar methods (behaviors)
  - One class is not a sub-class of another
- Example:
  - Class Line and class MyInteger
    - They are not related through inheritance
    - You want both to implement comparison methods
      - checkIsGreater(Object x, Object y)
      - checkIsLess(Object x, Object y)
      - checkIsEqual(Object x, Object y)
  - Define Comparison interface which has the three abstract methods above
Why do we use Interfaces? Reason #3

- To model multiple inheritance
  - A class can implement multiple interfaces while it can extend only one class
Interface vs. Abstract Class

- All methods of an Interface are abstract methods while some methods of an Abstract class are abstract methods
  - Abstract methods of abstract class have abstract modifier
- An interface can only define constants while abstract class can have fields
- Interfaces have no direct inherited relationship with any particular class, they are defined independently
  - Interfaces themselves have inheritance relationship among themselves
Problem of Rewriting an Existing Interface

- Consider an interface that you have developed called Dolt:
  
  ```java
  public interface Dolt {
      void doSomething(int i, double x);
      int doSomethingElse(String s);
  }
  ```

- Suppose that, at a later time, you want to add a third method to Dolt, so that the interface now becomes:
  
  ```java
  public interface Dolt {
      void doSomething(int i, double x);
      int doSomethingElse(String s);
      boolean didItWork(int i, double x, String s);
  }
  ```

If you make this change, all classes that implement the old Dolt interface will break because they don't implement all methods of the interface anymore.
Solution of Rewriting an Existing Interface

- Create more interfaces later
- For example, you could create a DoItPlus interface that extends Dolt:

  ```java
  public interface DoItPlus extends Dolt {
      boolean didItWork(int i, double x, String s);
  }
  ```

- Now users of your code can choose to continue to use the old interface or to upgrade to the new interface
When to use an Abstract Class over Interface?

- For non-abstract methods, you want to use them when you want to provide common implementation code for all sub-classes
  - Reducing the duplication
- For abstract methods, the motivation is the same with the ones in the interface – to impose a common behavior for all sub-classes without dictating how to implement it
- Remember a concrete can extend only one super class whether that super class is in the form of concrete class or abstract class