Inheritance
Inheritance

• One class inherits from another if it describes a specialized subset of objects

• Terminology:
  – the class that *inherits* is called a *child class* or *subclass*
  – the class that is *inherited from* is called a *parent class* or *superclass*
Characteristics of Inheritance

• Child classes are both extensions and contractions of parent classes:
  – extensions because they can add characteristics beyond those inherited, as well as override inherited characteristics
  – contractions because they are more specialized, and thus more restricted than parent class

• Inheritance is always transitive: a class can inherit features from a superclass that has inherited from another superclass, etc.
Forms of Inheritance: Specification

• Inheritance used to guarantee that classes share a common interface
• Subclass is realization of incomplete abstract specification, not refinement of existing type
• Mechanisms for this type of inheritance include interfaces and abstract classes and methods
Forms of Inheritance: Specialization

• Most common use of inheritance
• New class is specialized variety of parent
• Satisfies specifications of parent, but extends capabilities:
  – One or more inherited methods may be overridden
  – New fields/methods may be introduced
Inheritance in Java

• Subclass denoted by keyword *extends*
• Subclass declaration only contains differences from superclass
  – additional fields/methods
  – overridden methods
Inheritance hierarchies

• Real world: Hierarchies describe general/specific relationships
  – General concept at root of tree
  – More specific concepts are children

• Programming: Inheritance hierarchy
  – General superclass at root of tree
  – More specific subclasses are children
Designing class hierarchy

• Collect common properties in superclasses at base of hierarchy (root)
• Further down the tree get more specialization; classes at leaves are most specialized
Substitutability

- Idea that type given in variable declaration does not have to match type associated with value variable is holding
- Can occur through inheritance: for example, variable of superclass type can hold subclass objects
- Can also occur through use of interfaces - specifically with parameter passing
Substitutability

• Liskov substitution principle: since subclass inherits superclass behavior, can substitute subclass object when superclass object expected

• Polymorphism: client calls method on object (perceived to be) of parent type; if child object used instead, overriding method is executed
Invoking superclass methods

• Can call inherited (parent) method from overriding (child) method using keyword `super`:

```java
public void aMethod()
{
    super.aMethod(); // calls parent version
}
```
Notes on use of super

• Without keyword, method call in previous example would be recursive
• super is not a variable - doesn’t hold a reference to a superclass object - more like invoking a static method
• Turns off polymorphic call mechanism, frees superclass method to be called
Invoking superclass constructors

• Use super keyword in subclass constructor:
  ```java
  public Manager(String aName)
  {
      super(aName);
      // calls superclass constructor
      bonus = 0;
  }
  ```

• Here, super is used as name of method
Invoking superclass constructors

• Call to super must be *first* statement in subclass constructor

• If subclass constructor doesn't call super, superclass must have constructor without parameters
Preconditions & inherited methods

• Subclass method cannot require stronger precondition than method it overrides
• If superclass method had no precondition, subclass method can’t require one
Postconditions & inherited methods

- Subclass method:
  - postcondition must be at least as strong as postcondition of original
  - cannot be more private than original
  - cannot throw more more checked exceptions than original
Graphics programming with inheritance

• To draw shapes, can implement Icon interface, as we have seen

• More versatile method: subclass JPanel, override paintComponent() method

• Advantage: inherit rich behavior set
  – For example, can attach mouse listener to panel
  – With interface, you start from scratch; with inheritance, you get readymade features of superclass
Overriding paintComponent()

Example - draw a car:
public class MyPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D)g;
        car.draw(g2);
    }

    ...

}
Problem: corrupted panel

- Screen corrupted when moving pane
- Solution: call `super.paintComponent(g)` as first line in overriding method to refresh panel each time image is redrawn
Java Event Model

• Event: an action, e.g. mouse click, key press, menu selection

• Listener: object whose purpose is to wait for an event to occur, and perform appropriate action when event is detected

• Buttons & scrollbars are examples of objects that must incorporate listeners
Mouse Listeners

- There are 5 different mouse-related events, so the MouseListener interface accounts for all:

```java
public interface MouseListener {
    public void mouseClicked (MouseEvent e);
    public void mouseEntered (MouseEvent e);
    public void mouseExited (MouseEvent e);
    public void mousePressed (MouseEvent e);
    public void mouseReleased (MouseEvent e);
}
```
Mouse Listeners

• If a class implements this interface, it must provide definitions for all of these operations (whether you need them or not)
• If all operations aren’t needed, can use class MouseAdaptor instead
  – Implements MouseListener methods as empty method bodies
  – Can inherit from MouseAdaptor, then just override the methods you actually want to use
Mouse Listeners

• Alternatively, you can implement the MouseListener interface and create your own empty methods for operations you don’t need - for example:
  ```java
  public void mouseExited(MouseEvent e) {} 
  ```

• Method addMouseListener is inherited from Frame - takes a MouseListener instance as its object
Using anonymous class to add mouse listener

addMouseListener(new MouseAdapter(){
    public void mousePressed (MouseEvent event)
    {
        mouse action goes here
    }
});
Abstract classes & methods

Keyword abstract applied to a class guarantees that subclass must be constructed

Individual methods in an abstract class can be declared abstract; such methods must be overridden by child classes

Abstract classes almost always contain abstract methods, but not all methods have to be abstract
Abstract classes & methods

Abstract methods are like interface methods: specified but undefined

Abstract classes are used to factor out common behavior from sets of related (sub)classes

Abstract classes cannot be instantiated; but, as with interfaces, can declare variables of abstract class type
Abstract classes

Abstract classes are created strictly as superclasses
Can have instance fields and methods
Can tag any class as abstract; means can’t create objects from this class, but can inherit from it
Abstract classes Vs. interfaces

Advantage of abstract class is ability to define some behavior as well as specify it; more can be inherited from an abstract class than from an interface. Disadvantage is lack of multiple inheritance; can only inherit from one class, but can implement as many interfaces as desired.
Abstract classes Vs. interfaces

Abstract classes can have fields

Interface types can only have constants (public static final)

Abstract classes can define methods

Interface types can only declare methods
Abstract classes & interfaces

Good idea to supply both an interface type and a class that implements some of its methods with convenient defaults; gives best of both worlds

Java library has many examples of such pairs
Modifiers and Inheritance

Public data field or method can be accessed outside its class definition; a public class can be accessed outside its package.

Protected data field or method can be accessed within its class, within other classes in its package, or within subclasses.

Private data field or method can be accessed only within its class.
Modifiers and Inheritance

Static fields are shared by all instances of a class & can be invoked even is no class instance has been created; static methods cannot be overridden

Abstract classes cannot be instantiated, and can only be used as parent classes
Modifiers and Inheritance

Modifier final is opposite of abstract:

- when applied to a class, means the class cannot be subclassed
- when applied to a method, means method cannot be overridden
- when applied to a variable, the variable becomes a constant
Benefits of Inheritance

Software reusability
  code sharing
  reusable components
Increased reliability: code that is executed frequently tends to have fewer bugs
Interface consistency
Information hiding
Costs of Inheritance

Execution speed - inherited methods often slower than code written for specific purpose

Program size

Message-passing overhead

Program complexity
Inheritance Vs. Composition

Both are techniques for software reuse

Inheritance is appropriate in a situation when one object is a specialized type of another object -- e.g. a professor is a teacher

Composition is appropriate when an object is made up of other objects -- e.g. a blender has a motor, a set of controls, and a container to hold the contents