Abstract Classes & Inheritance Hierarchies
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 if 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
Hierarchy of Swing Components

• Java libraries contain several examples of hierarchical classes related by inheritance
• User interface component classes in the swing library present one such example
Components

- Base of the component hierarchy is the Component class; includes methods:
  - `getWidth()`, `getHeight()`, `setSize(int,int)`
  - `setBackground(Color c)`, `getBackground()`
  - `setVisible(boolean)`, `setEnabled(boolean)`
  - `repaint(Graphics)`, `paint(Graphics)`
  - `setFont(Font)`, `getFont()`
  - `addMouseListener(MouseListener)`, `addKeyListener(KeyListener)`
Containers

• Container class is subclass of Component; most important property is its ability to contain components

• Methods include:
  – setLayout(LayoutManager)
  – add(Component), remove(Component)
A little history ...

• First came AWT, Abstract Window Toolkit
  – Used *native* components, resulting in subtle platform inconsistencies
  – Write once, run anywhere -> Write once, debug everywhere

• Swing library developed as platform-independent solution
Characteristics of Swing

• All components painted on blank windows
  – all components drawn pixel by pixel
  – when component changes state (e.g. button pushed), toolkit redraws it

• Swing has complete control over behavior of components - does not use native components
Hierarchy of Swing Components
Hierarchy of standard geometric shapes

• Original AWT classes: integer coordinates
  – Point
  – Rectangle
  – Polygon
• Java 2 introduced java.awt.geom package - more sophisticated shapes with floating-point coordinates
• Legacy classes are folded into new hierarchy
Hierarchy of standard geometric shapes

- Point2D
- Point
- Shape
  - Line2D
  - Rectangular Shape
    - Rectangle2D
      - Rectangle
    - Round Rectangle2D
    - Ellipse2D
    - Arc2D
  - GeneralPath
  - Polygon
RectangularShape

• Superclass for Rectangle2D, RoundRectangle2D, Ellipse2D and Arc2D

• Methods include:
  – getCenterX, getCenterY
  – getMinX, getMinY, getMaxX, getMaxY
  – getWidth, getHeight
  – setFrameFromCenter, setFrameFromDiagonal
Rectangle2D class

- Has inner classes Float and Double; hence the name Rectangle2D.Double
- Rectangle2D is an abstract class; inner classes Float and Double are concrete subclasses that define a small number of methods, including getX(), getY(), getWidth() and getHeight()
Template Method design pattern

- Superclass defines a method that calls primitive operations that a subclass needs to supply
- Each subclass can supply the primitive operations most appropriate for it
- Template method contains knowledge of how to combine primitive operations into more complex operation
Template method pattern: context

• An algorithm is applicable for multiple types
• The algorithm can be broken down into primitive operations
• The primitive operations can be different for each type
• The order of the primitive operations doesn't depend on the type
Template method pattern: solution

• Define a superclass that has a method for the algorithm and abstract methods for the primitive operations.

• Implement the algorithm to call the primitive operations in the appropriate order.
Template method pattern: solution

• Do not define the primitive operations in the superclass, or define them to have appropriate default behavior.

• Each subclass defines the primitive operations but not the algorithm.
Template Method Pattern

```
AbstractClass
{abstract}

- templateMethod()
- primitiveOp1()
- primitiveOp2()

ConcreteClass

- primitiveOp1()
- primitiveOp2()

Calls primitiveOp1(), primitiveOp2()
Abstract methods
```
Rectangle2D and the Template Method Pattern

• Most of the work is done by Rectangle2D methods, not inner classes; parent class methods call inner class (primitive) methods as needed

• Example:
  
  public boolean contains(double x, double y)
  {
    double x0 = getX();
    double y0 = getY();
    return x >= x0 && y >= y0 && x < x0 + getWidth()
    && y < y0 + getHeight();
  }

Template Method pattern & Rectangle2D

• Pattern name:
  – AbstractClass
  – ConcreteClass
  – templateMethod
  – primitiveOpn

• Actual name
  – Rectangle2D
  – Rectangle2D.Double
  – contains()
  – getX(), getY(), etc.
Hierarchy of Exception Classes

• All exceptions extend class Throwable
• Throwable has two subclasses:
  – Error: subclasses of this denote fatal errors (e.g. divide by 0, out of memory, etc.)
  – Exception: superclass for all exceptions that occur on an application level
Exception class

• Has several subclasses: notable among these is RuntimeException
  – superclass for all unchecked exceptions
  – examples include NullPointerException and IndexOutOfBoundsException

• All subclasses of Exception that are not subclasses of RuntimeException are checked exceptions
Checked Exceptions

• These are the type of exceptions that require either a throws clause or try/catch block
• Example: IOException and its subclasses
Hierarchy of Exception Classes

- Throwable
  - Exception
    - Runtime Exception
      - NullPointer Exception
      - IndexOutOfBoundsException
    - IOException
    - ClassNotFoundException
      - FileNotFoundException
Defining Exception Classes

• Decide whether or not exception should be checked
  – Use checked exception if error condition is out of programmer control (e.g. a network failure)
  – Use unchecked exception if error caused by programmer inattention (e.g. null pointer exception)

• Subclass Exception or RuntimeException - note that unchecked exceptions must subclass RuntimeException
Defining Exception Classes

• Provide two constructors:

    public class IllegalFormatException extends Exception
    {
        public IllegalFormatException() {}
        public IllegalFormatException(String reason)
        {
            super(reason);
        }
    }

• Throw exception when needed:

    throw new IllegalFormatException("number expected");