Chapter 7
The Java Object Model
Chapter Topics

• The Java Type System
• Type Inquiry
• The Object Class
• Shallow and Deep Copy
• Serialization
• Reflection
• Generic Types
• JavaBeans Components
Types

• Type: set of values and the operations that can be applied to the values
• Strongly typed language: compiler and run-time system check that no operation can execute that violates type system rules
  • Compile-time check
  • Employee e = new Employee();
  • e.clear(); // ERROR
  • Run-time check:
  • e = null;
  • e.setSalary(20000); // ERROR
Java Types

• Primitive types:
  • int short long byte
  • char float double boolean
• Class types
• Interface types
• Array types
• The null type
• Note: void is not a type
• Exercise: What kind of type?
  • int
  • Rectangle
  • Shape
  • String[]
  • double[][]
Java Values

• value of primitive type
• reference to object of class type
• reference to array
• null
• Note: Can't have value of interface type
• Exercise: What kind of value?
  • 13
  • new Rectangle(5, 10, 20, 30);
  • "Hello"
  • new int[] { 2, 3, 5, 7, 11, 13 }
  • null
Subtype Relationship

• S is a subtype of T if:

  • S and T are the same type
  • S and T are both class types, and T is a direct or indirect superclass of S
  • S is a class type, T is an interface type, and S or one of its superclasses implements T
  • S and T are both interface types, and T is a direct or indirect superinterface of S
  • S and T are both array types, and the component type of S is a subtype of the component type of T
  • S is not a primitive type and T is the type Object
  • S is an array type and T is Cloneable or Serializable
  • S is the null type and T is not a primitive type
Subtype Examples

• Container is a subtype of Component
• JButton is a subtype of Component
• FlowLayout is a subtype of LayoutManager
• ListIterator is a subtype of Iterator
• Rectangle[] is a subtype of Shape[]
• int[] is a subtype of Object
• int is not a subtype of long
• long is not a subtype of int
• int[] is not a subtype of Object[]
Subtype Examples

Object

Component

Container

JComponent

Abstract Button

JButton

«interface» Layout Manager

Flow Layout

«interface» Iterator

«interface» ListIterator
The ArrayStoreException

- Rectangle[] is a subtype of Shape[]
- Can assign Rectangle[] value to Shape[] variable:
  - Rectangle[] r = new Rectangle[10];
  - Shape[] s = r;
- Both r and s are references to the same array
- That array holds rectangles
- The assignment
  - s[0] = new Polygon();
- Compiles because Polygon is a subtype of Shape.
- Throws an ArrayStoreException at runtime because:
  - Each array remembers its component type
Wrappers for Primitive Types

• Primitive types aren't classes
• Use wrappers when objects are expected
• Wrapper for each type:
  • Integer Short Long Byte
  • Character Float Double Boolean
• Auto-boxing and auto-unboxing
• `ArrayList<Integer> numbers = new ArrayList<Integer>();`
• `numbers.add(13); // calls new Integer(13)`
• `int n = numbers.get(0); // calls intValue();`
Enumerated Types

- Finite set of values
- Example:
  ```java
  enum Size { SMALL, MEDIUM, LARGE }
  ```
- Typical use:
  ```java
  Size imageSize = Size.MEDIUM;
  if (imageSize == Size.SMALL) . . .
  ```
- Safer than integer constants: only methods of the Size class can construct new instances, and there are no such methods.
  ```java
  public static final int SMALL = 1; //not a good alternative
  public static final int MEDIUM = 2; // allows definition of other
  public static final int LARGE = 3; //sizes elsewhere
  ```
Typesafe Enumerations

• Enum keyword defines a class with a private constructor and a finite number of instances.

• It is equivalent to:

```java
public class Size
{
    private /* ! */ Size() { }
    public static final Size SMALL = new Size();
    public static final Size MEDIUM = new Size();
    public static final Size LARGE = new Size();
}
```
Enumerated Types

• Because enumerated types are classes, you can add methods and instance fields.
• You can also supply constructors, but they can only be used to construct the enumeration values.
• Example:

```java
public enum Size
{
    SMALL (0.5), MEDIUM(1), LARGE(2);

    private Size(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    private double value;
}
```
Enumerated Types

• All enumeration classes extend the Enum class, from which they inherit a number of methods such as toString()

• The enum class implements the Comparable interface. The compareTo method compares instances in the order in which they are defined.
Type Inquiry

• Test whether e is a Shape:
  • if (e instanceof Shape) . . .
• Common before casts:
  • Shape s = (Shape) e;
  • Don't know exact type of e
  • Could be any class implementing Shape
  • If e is null, test returns false (no exception)
The Class Class

- getClass method gets class of any object
- Returns object of type Class
- Class object describes a type
- Object e = new Rectangle();
- Class c = e.getClass();
- System.out.println(c.getName()); // prints java.awt.Rectangle
- Class.forName method yields Class object:
- Class c = Class.forName("java.awt.Rectangle");
- .class suffix yields Class object:
- Class c = Rectangle.class; // java.awt prefix not needed
- Class is a misnomer. Class objects can describe any type including primitives, classes, and interfaces.
- Class objects can be described with int.class, void.class, Shape.class
An Employee Object vs. the Employee.class Object

: Employee

name = "Jane Doe"
salary = 50000

: Class

name = "Employee"
superclass = 

: Class

name = "java.lang.Object"
superclass = null
Type Inquiry

• Test whether e is a Rectangle:
  • if (e.getClass() == Rectangle.class) . . .
  • Ok to use ==
  • A unique Class object for every class
  • Test fails for subclasses
• Use instanceof to test for subtypes:
  • if (e instanceof Rectangle) . . .
    • Will pass if the object is a subclass of Rectangle
Array Types

- Can apply getClass to an array
- Returned object describes an array type

```java
double[] a = new double[10];
Class c = a.getClass();
if (c.isArray())
    System.out.println(c.getComponentType());
    // prints double
```

- getName produces strange names for array types
  - [D for double[])
  - [[[java.lang.String; for String[][]]
Object: The Cosmic Superclass

- All classes extend Object
- Most useful methods:
  - String toString()
  - boolean equals(Object otherObject)
  - Object clone()
  - int hashCode()
The `toString` Method

- Returns a string representation of the object
- Useful for debugging
- Example: `Rectangle.toString` returns something like
  - `java.awt.Rectangle[x=5,y=10,width=20,height=30]`
- `toString` used by concatenation operator
  - `aString + anObject`
  - means
  - `aString + anObject.toString()`
- `Object.toString` prints class name and object address
- `System.out.println(System.out)`
  - yields
  - `java.io.PrintStream@d2460bf`
  - Implementor of `PrintStream` didn't override `toString`: 
Overriding the toString Method

• Format all fields:
  public class Employee
  {
      public String toString()
      {
          return getClass().getName()
          + "[name=\" + name
          + ",salary=\" + salary
          + "]\";
      }
      ...
  }

• Typical string:
• Employee[name=Harry Hacker,salary=35000]
Overriding toString in Subclass

• Format superclass first
  
  public class Manager extends Employee
  {
    public String toString()
    {
      return super.toString()
        + "[department=" + department + "]";
    }
    ...
  }

• Typical string
  
  Manager[name=Dolly Dollar,salary=100000][department=Finance]

• Note that superclass reports actual class name
The equals Method

- equals tests for equal contents
- == tests for equal location
- Used in many standard library methods
- Example: ArrayList.indexOf

```java
/**
 * Searches for the first occurrence of the given argument,
 * testing for equality using the equals method.
 * @param elem an object.
 * @return the index of the first occurrence of the argument in this list; returns -1 if
 * the object is not found.
 */
public int indexOf(Object elem)
{
    if (elem == null)
    {
        for (int i = 0; i < size; i++)
            if (elementData[i] == null) return i;
    }
    else
    {
        for (int i = 0; i < size; i++)
            if (elem.equals(elementData[i])) return i;
    }
    return -1;
}
```
Overriding the equals Method

- Notion of equality depends on class
- Common definition: compare all fields
  ```java
  public class Employee
  {
      public boolean equals(Object otherObject)
      { // not complete--see below
          Employee other = (Employee)otherObject;
          return name.equals(other.name)
              && salary == other.salary;
      }
  }
  ```
- Must cast the Object parameter to subclass
- Use == for primitive types, equals for object fields
Overriding equals in Subclass

• Call equals on superclass

```java
public class Manager {
    public boolean equals(Object otherObject) {
        Manager other = (Manager) otherObject;
        return super.equals(other) && department.equals(other.department);
    }
}
```
Not all equals Methods are Simple

• Two sets are equal if they have the same elements in some order
• public boolean equals(Object o)
  • {
    •     if (o == this) return true;
    •     if (!(o instanceof Set)) return false;
    •     Collection c = (Collection) o;
    •     if (c.size() != size()) return false;
    •     return containsAll(c);
  • }
The Object.equals Method

- Object.equals tests for identity:

```java
public class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
    ...
}
```

- Override equals if you don't want to inherit that behavior
Requirements for equals Method

- reflexive: \( x.\text{equals}(x) \)
- symmetric: \( x.\text{equals}(y) \) if and only if \( y.\text{equals}(x) \)
- transitive: if \( x.\text{equals}(y) \) and \( y.\text{equals}(z) \), then \( x.\text{equals}(z) \)
- \( x.\text{equals}(null) \) must return false
- The first three requirements represent an equivalence relation.
Fixing Employee.equals

- Violates two rules
- Add test for null:
  
  ```java
  if (otherObject == null) return false
  ```
- What happens if otherObject not an Employee
  - Should return false (because of symmetry)
- Common error: use of instanceof
  
  ```java
  if (!otherObject instanceof Employee) return false;
  // don't do this for non-final classes
  ```
- Violates symmetry: Suppose e, m have same name, salary
  
  ```java
  e.equals(m) is true (because m instanceof Employee)
  m.equals(e) is false (because e isn't an instance of Manager)
  ```
- Remedy: Test for class equality
  
  ```java
  if (getClass() != otherObject.getClass()) return false;
  ```
The Perfect equals Method

• Start with these three tests:

    public boolean equals(Object otherObject)
    {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        ...
    }

• First test is an optimization
The Perfect equals Method

• Because a subclass has no access to the superclass state, its equals method must invoke the superclass version:

```java
public class Manager extends Employee {
    public Boolean equals (Object otherObject) {
        if (!super.equals(otherObject)) return false;
        Manager other = (Manager) otherObject;
        return bonus == other.bonus
    }
    ...
}
```
Consequences of the Symmetry Condition for Equality Testing

• Using the instanceof test in the equals method may violate the symmetry requirement.
• Suppose we have an Employee object e and a Manager object m that happen to have the same name and salary.
  
  ```java
  public class Employee {
      public boolean equals(Object otherObject) {
          if (! (otherObject instanceof Employee)) return false;
          . . . // Don’t
      }
  }
  
  public class Manager extends Employee {
      public boolean equals(Object otherObject) {
          if (! (otherObject instanceof Manager)) return false;
          . . . // Don’t
      }
  }
  
  e.equals(m) would be true, but m.equals(e) would be false.
  • Use getClass instead of instanceof.
  • Using instanceof in the Set equality test was okay because Sets don’t have subclasses.
Hash Tables

- Hash tables allow us to find an object very quickly.
- In a list, we find an object by sequentially accessing each location looking for a match. This is an $O(n)$ operation.
- If we use a binary search algorithm we can reduce the running time to $O(\log n)$.
- A good hash table yields a run time of $O(1)$.
- Hashing requires a hash algorithm that takes some of the contents of an object and turning it into a number that is used as an index into a table.
- In the best case each hashcode output gives a unique index number.
- Unfortunately multiple objects can return the same hashcode, resulting in a collision.
- Since the range of possible numbers is generally larger than the available space for an array, we use the mod operator to constrain the range of the hashcode output. This can also lead to collisions.
Hash Tables

- There are two ways of handling collisions:
  - Open hashing
  - Bucketed hashing

- In open hashing, if you are looking for an item at its hashed location you have to check that you have indeed found the correct item. Some other item could be in that slot.
  - If the item you found is not the correct item you have to look in another spot.
  - Looking in the adjacent spot is not a good idea because it leads to clustering of objects in the table.
  - A second hash function can be used after a collision, and a third function after that.
Hash Tables

• In bucketed hashing each location in the table can hold an item, and also a pointer or reference to a linked list of items that hash to the same location.

• If the item in the table is not the correct one you can traverse the linked list looking for the item in the list.

• A good hash table finds the item or determines that it is not in the list in 1 or 2 tries.

• If there are many collisions it’s time to create a bigger table or better hash algorithm.
Hashing

• hashCode method is used in HashMap, HashSet.
• The Object class has a hashCode method that hashes the object’s address. This can be used to store objects and find them quickly.
• The hashCode needs to be consistent with the equals method so that:
  • If x.equals(y), then x.hashCode() == y.hashCode().
• The default hashCode computes an int from an object’s address.
• Example: hash code of String
  ```java
  int h = 0;
  for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
  ```
  • Hash code of "eat" is 100184
  • Hash code of "tea" is 114704
Hashing
- The default `hashCode()` is NOT compatible with a redefined `equals()`
- The default `equals` compares two objects’ hashed addresses.
- If `x.equals(y)` then `x.hashCode() == y.hashCode()`
- Our `equals` method compares the contents of several fields.
- Remedy: Hash all fields and combine codes:
  ```java
  public class Employee {
    public int hashCode() {
      return name.hashCode() + new Double(salary).hashCode();
    }
  }
  ```
- Note the wrapping of the double value of `salary`.
- Now equal `Employee` objects will yield the same hash code.
Hashing
- A look at the hashcode implementation of the AbstractSet class:

```java
class AbstractSet {
    public int hashCode()
    {
        int h = 0;
        Iterator i = iterator();
        while (i.hasNext())
        {
            Object obj = i.next();
            if (obj != null) h += obj.hashCode();
        }
        return h;
    }
    ...
}
```

- Note that the sum remains the same even if the order of the elements varies.
Shallow and Deep Copies

- Assignment (copy = e) makes shallow copy
- Clone to make deep copy
- Employee cloned = (Employee)e.clone();
Cloning

• Object.clone makes new object and copies all fields
• Cloning is subtle
• Object.clone is protected
• Subclass must redefine clone to be public

public class Employee
{
    public Object clone()
    {
        return super.clone(); // not complete
    }
    ...
}
The Cloneable Interface

• Object.clone is nervous about cloning
• Will only clone objects that implement Cloneable interface

```java
public interface Cloneable {
    
}
```

• Interface has no methods!
• Tagging interface--used in test
• if x implements Cloneable
• Object.clone throws CloneNotSupportedException
• A checked exception
public class Employee
    implements Cloneable
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            return null; // won't happen
        }
    }
    ...
Shallow Cloning

- clone makes a shallow copy
- Instance fields aren't cloned
Deep Cloning

- Why doesn't clone make a deep copy?
- Wouldn't work for cyclic data structures
- Not a problem for immutable fields
- You must clone mutable fields

```java
class Employee
```
```java
implements Cloneable
```
```java{
    public Object clone()
    {
        try
        {
            Employee cloned = (Employee) super.clone();
            cloned.hireDate = (Date) hiredate.clone();
            return cloned;
        }
        catch(CloneNotSupportedException e)
        {
            return null; // won't happen
        }
    }
}
```
Deep Cloning

Diagram:

```java
Employee e = new Employee()
    .setName(name)
    .setSalary(salary)
    .setHireDate(hireDate);

Employee cloned = e
    .setName(name)
    .setSalary(salary)
    .setHireDate(hireDate);
```

```
 String
    .setSomeValue()

 Date
    .setSomeValue()
```
Cloning and Inheritance

- Object.clone is paranoid
- clone is protected
- clone only clones Cloneable objects
- clone throws checked exception
- You don't have that luxury
- Manager.clone must be defined if Manager adds mutable fields
- Rule of thumb: if you extend a class that defines clone, redefine clone
- Lesson to learn: Tagging interfaces are inherited. Use them only to tag properties that inherit
Serialization

• Save collection of objects to stream
  Employee[] staff = new Employee[2];
  staff.add(new Employee(...));
  staff.add(new Employee(...));

• Construct ObjectOutputStream:
  ObjectOutputStream out
    = new ObjectOutputStream(
      new FileOutputStream("staff.dat"));

• Save the array and close the stream
  out.writeObject(staff);
  out.close();
Serialization

• The array and all of its objects and their dependent objects are saved
• Employee doesn't have to define any method
• Needs to implement the Serializable interface
• Another tagging interface with no methods
How Serialization Works

- Each newly encountered object is saved
- Each object gets a serial number in the stream
- No object is saved twice
- Reference to already encountered object saved as "reference to #"
Serializing Unserializable Classes

- Some classes are not serializable
- Example: Ellipse2D.Double
- How can we serialize Car?
- Suppress default serialization to avoid exception
- Mark with transient:
  - private transient Ellipse2D frontTire;
- Supply private (!) methods
  - private void writeObject(ObjectOutputStream out)
  - private void readObject(ObjectInputStream in)
- In these methods
  - Call writeDefaultObject/readDefaultObject
- Manually save other data
A Serializable Car

import java.awt.*;
import java.awt.geom.*;
import java.io.*;

/** A serializable car shape. */
public class Car implements Serializable {

/** Constructs a car. */
public Car(int x, int y, int width) {
    body = new Rectangle(x, y + width / 6,
    width - 1, width / 6);
    roof = new Rectangle(x + width / 3, y,
    width / 3, width / 6);
    frontTire = new Ellipse2D.Double(x + width / 6, y + width / 3,
    width / 6, width / 6);
    rearTire = new Ellipse2D.Double(x + width * 2 / 3, y + width / 3,
    width / 6, width / 6);
}
private void writeObject(ObjectOutputStream out) throws IOException
{
    out.defaultWriteObject();
    writeRectangularShape(out, frontTire);
    writeRectangularShape(out, rearTire);
}

/**
 * A helper method to write a rectangular shape.
 * @param out the stream onto which to write the shape
 * @param s the shape to write
 */
private static void writeRectangularShape(ObjectOutputStream out, RectangularShape s) throws IOException
{
    out.writeDouble(s.getX());
    out.writeDouble(s.getY());
    out.writeDouble(s.getWidth());
    out.writeDouble(s.getHeight());
}
A Serializable Car

051: private void readObject(ObjectInputStream in)
052: throws IOException, ClassNotFoundException
053: {
054:     in.defaultReadObject();
055:     frontTire = new Ellipse2D.Double();
056:     readRectangularShape(in, frontTire);
057:     rearTire = new Ellipse2D.Double();
058:     readRectangularShape(in, rearTire);
059: }
/**
 * A helper method to read a rectangular shape.
 * @param in the stream from which to read the shape
 * @param s the shape to read. The method sets the frame
 * of this rectangular shape.
 */

private static void readRectangularShape(ObjectInputStream in,
        RectangularShape s)
        throws IOException
{
    double x = in.readDouble();
    double y = in.readDouble();
    double width = in.readDouble();
    double height = in.readDouble();
    s.setFrame(x, y, width, height);
}
A Serializable Car

078: /**
079:   * Draws the car.
080:   * @param g2 the graphics context
081:   */
082: public void draw(Graphics2D g2)
083: {
084:     g2.draw(body);
085:     g2.draw(roof);
086:     g2.draw(frontTire);
087:     g2.draw(rearTire);
088: }
089: public String toString()
090: {
091:     return getClass().getName()
092:         + "[body=" + body
093:         + ",roof=" + roof
094:         + ",frontTire=" + formatRectangularShape(frontTire)
095:         + ",rearTire=" + formatRectangularShape(rearTire)
096:         + "]";
097: }
098: }
A Serializable Car

/**
 * A helper method to format a rectangular shape.
 * @param s the shape to format
 * @return a formatted representation of the given shape
 */
private static String formatRectangularShape(RectangularShape s)
{
    return RectangularShape.class.getName()
        + "[x=" + s.getX()
        + ",y=" + s.getY()
        + ",width=" + s.getWidth()
        + ",height=" + s.getHeight()
        + "]";
}

private Rectangle body;
private Rectangle roof;
private transient Ellipse2D.Double frontTire;
private transient Ellipse2D.Double rearTire;
Reflection

• Ability of running program to find out about its objects and classes

• Class object reveals
  • superclass
  • interfaces
  • package
  • names and types of fields
  • names, parameter types, return types of methods
  • parameter types of constructors
Reflection

- Class getSuperclass()
- Class[] getInterfaces()
- Package getPackage()
- Field[] getDeclaredFields()
- Constructor[] getDeclaredConstructors()
- Method[] getDeclaredMethods()
Enumerating Fields

• Print the names of all static fields of the Math class:

```java
Field[] fields = Math.class.getDeclaredFields();
for (Field f : fields)
    if (Modifier.isStatic(f.getModifiers()))
        System.out.println(f.getName());
```
Enumerating Constructors

• Print the names and parameter types of all Rectangle constructors:

```java
for (Constructor c : cons) {
    Class[] params = cc.getParameterTypes();
    System.out.print("Rectangle(");
    boolean first = true;
    for (Class p : params) {
        if (first) first = false; else System.out.print(", ");
        System.out.print(p.getName());
    }
    System.out.println(")");
}
```

• Yields:

Rectangle()
Rectangle(java.awt.Rectangle)
Rectangle(int, int, int, int)
Rectangle(int, int)
Rectangle(java.awt.Point, java.awt.Dimension)
Rectangle(java.awt.Point)
Rectangle(java.awt.Dimension)
Getting A Single Method Descriptor

• Supply method name
• Supply array of parameter types
• Example: Get Rectangle.contains(int, int):
  Method m = Rectangle.class.getDeclaredMethod("contains", int.class, int.class);
• Example: Get default Rectangle constructor:
  Constructor c = Rectangle.class.getDeclaredConstructor();
• getDeclaredMethod, getDeclaredConstructor are varargs methods
Invoking a Method

- Supply implicit parameter (null for static methods)
- Supply array of explicit parameter values
- Wrap primitive types
- Unwrap primitive return value
- Example: Call System.out.println("Hello, World") the hard way.
  
  ```java
  Method m = PrintStream.class.getDeclaredMethod("println", String.class);
  m.invoke(System.out, "Hello, World!");
  ```

- `invoke` is a varargs method
Inspecting Objects

- Can obtain object contents at runtime
- Useful for generic debugging tools
- Need to gain access to private fields

```java
Class c = obj.getClass();
Field f = c.getDeclaredField(name);
f.setAccessible(true);
Throws exception if security manager disallows access
Access field value:
Object value = f.get(obj);
f.set(obj, value);
```
- Use wrappers for primitive types
Inspecting Objects

- Example: Peek inside string tokenizer:

```java
01: import java.lang.reflect.*;
02: import java.util.*;
03:
04: /**
05:    This program shows how to use reflection to print
06:    the names and values of all nonstatic fields of an object.
07: */
08: public class FieldTester
09: {
10:    public static void main(String[] args)
11:          throws IllegalAccessException
12:    {
13:        Random r = new Random();
14:        System.out.print(spyFields(r));
15:        r.nextInt();
16:        System.out.println("\nAfter calling nextInt:\n");
17:        System.out.print(spyFields(r));
18:    }
```
public static String spyFields(Object obj) throws IllegalAccessException {
    StringBuffer buffer = new StringBuffer();
    Field[] fields = obj.getClass().getDeclaredFields();
    for (Field f : fields) {
        if (!Modifier.isStatic(f.getModifiers())) {
            f.setAccessible(true);
            Object value = f.get(obj);
            buffer.append(f.getType().getName());
            buffer.append(" ");
            buffer.append(f.getName());
            buffer.append("=");
            buffer.append(" " + value);
            buffer.append("\n");
        }
    }
    return buffer.toString();
}
Inspecting Objects

• Program output:

    int currentPosition=0
    int newPosition=-1
    int maxPosition=13
    java.lang.String str="Hello, World!"
    java.lang.String delimiters="",
    boolean retDelims=false
    boolean delimsChanged=false
    char maxDelimChar="",
    ---
    int currentPosition=5
    ...
Inspecting Array Elements

• Use static methods of Array class
  • Object value = Array.get(a, i);
  • Array.set(a, i, value);
  • int n = Array.getLength(a);
• Construct new array:
  • Object a = Array.newInstance(type, length);
Generic Types

- A generic type has one or more type variables
- Type variables are instantiated with class or interface types
- Cannot use primitive types, e.g. no `ArrayList<int>`
- When defining generic classes, use type variables in definition:

```java
public class ArrayList<E>
{
    public E get(int i) { . . . }
    public E set(int i, E newValue) { . . . }
    . . .
    private E[] elementData;
}
```

- NOTE: If S a subtype of T, `ArrayList<S>` is not a subtype of `ArrayList<T>`.
Generic Methods

• Generic method = method with type parameter(s)
  public class Utils
  {
    public static <E> void fill(ArrayList<E> a, E value, int count)
    {
      for (int i = 0; i < count; i++)
        a.add(value);
    }
  }

• A generic method in an ordinary (non-generic) class
• Type parameters are inferred in call
• ArrayList<String> ids = new ArrayList<String>();
• Utils.fill(ids, "default", 10); // calls Utils.<String>fill
Type Bounds

• Type variables can be constrained with type bounds
• Constraints can make a method more useful
• The following method is limited:
  ```java
  public static <E> void append(ArrayList<E> a, ArrayList<E> b, int count)
  {
      for (int i = 0; i < count && i < b.size(); i++)
          a.add(b.get(i));
  }
  ```
• Cannot append an ArrayList<Rectangle> to an ArrayList<Shape>
Type Bounds

• Overcome limitation with type bound:
• public static <E, F extends E> void append(
    ArrayList<E> a, ArrayList<F> b, int count)
  {
    for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
  }
• extends means "subtype", i.e. extends or implements
• Can specify multiple bounds:
• E extends Cloneable & Serializable
Wildcards

• Definition of append never uses type F. Can simplify with wildcard:

```java
public static <E> void append(
    ArrayList<E> a, ArrayList<? extends E> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```
Wildcards

- Wildcards restrict methods that can be called:
- `ArrayList<? Extends E>.set` method has the form
  - `? extends E add(? extends E newElement)`
  - You cannot call this method!
- No value matches `? extends E` because `?` is unknown
- Ok to call `get`:
  - `? extends E get(int i)`
  - Can assign return value to an element of type `E`
Wildcards

- Wildcards can be bounded in opposite direction
- ? super F matches any supertype of F

```java
public static <F> void append(
    ArrayList<? super F> a, ArrayList<F> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```

- Safe to call ArrayList<? super F>.add:
- boolean add(? super F newElement)
- Can pass any element of type F (but not a supertype!)
Wildcards

• Typical example--start with
  public static <E extends Comparable<E>> E getMax(ArrayList<E> a)
  {
    E max = a.get(0);
    for (int i = 1; i < a.size(); i++)
      if (a.get(i).compareTo(max) > 0) max = a.get(i);
    return max;
  }

• E extends Comparable<E> so that we can call compareTo
• Too restrictive--can't call with ArrayList<GregorianCalendar>
• GregorianCalendar does not implement Comparable<GregorianCalendar>, only Comparable<Calendar>
• Wildcards to the rescue:
• public static <E extends Comparable<? super E>> E getMax(ArrayList<E> a)
Type Erasure

• Virtual machine does not know about generic types
• Type variables are erased--replaced by type bound or Object if unbounded
• Ex. ArrayList\<E\> becomes
  
  ```java
  public class ArrayList
  {
      public Object get(int i) { . . . }
      public Object set(int i, Object newValue) { . . . }
      . . .
      private Object[] elementData;
  }
  ```

• Ex. getMax becomes
  
  ```java
  public static Comparable getMax(ArrayList a)
      // E extends Comparable<? super E> erased to Comparable
  ```

• Erasure necessary to interoperate with legacy (pre-JDK 5.0) code
Limitations of Generics

• Cannot replace type variables with primitive types
• Cannot construct new objects of generic type
• `a.add(new E()); // Error--would erase to new Object()
• Workaround: Use class literals
  
  public static <E> void fillWithDefaults(ArrayList<E>,
  Class<? extends E> cl, int count)
  throws InstantiationException, IllegalAccessException
  {
    for (int i = 0; i < count; i++)
      a.add(cl.newInstance());
  }
• Call as `fillWithDefaults(a, Rectangle.class, count)`
Limitations of Generics

- Cannot form arrays of parameterized types
- Comparable<E>[]} is illegal. Remedy: ArrayList<Comparable<E>>
- Cannot reference type parameters in a static context (static fields, methods, inner classes)
- Cannot throw or catch generic types
- Cannot have type clashes after erasure. Ex. GregorianCalendar cannot implement Comparable<GregorianCalendar> since it already implements Comparable<Calendar>, and both erase to Comparable
Components

• More functionality than a single class
• Reuse and customize in multiple contexts
• "Plug components together" to form applications
• Successful model: Visual Basic controls
  • calendar
  • graph
  • database
  • link to robot or instrument
• Components composed into program inside builder environment
A Builder Environment
Java Beans

- Java component model
- Bean has
  - methods (just like classes)
  - properties
  - events
Java Beans
Java Beans

- A calendar bean:

- A property sheet:
The Façade Class

• Bean usually composed of multiple classes
• One class nominated as facade class
• Clients use only facade class methods
**The Façade Pattern**

**Context**
- A subsystem consists of multiple classes, making it complicated for clients to use
- Implementor may want to change subsystem classes
- Want to give a coherent entry point

**Solution**
- Define a facade class that exposes all capabilities of the subsystem as methods
- The facade methods delegate requests to the subsystem classes
- The subsystem classes do not know about the facade class
The Façade Pattern
The Façade Pattern

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (Beans)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Builder tool</td>
</tr>
<tr>
<td>Façade</td>
<td>Main bean class with which the tool interacts</td>
</tr>
<tr>
<td>SubsystemClass</td>
<td>Class used to implement bean functionality</td>
</tr>
</tbody>
</table>
Bean Properties

• Property = value that you can get and/or set
• Most properties are get-and-set
• Can also have get-only and set-only
• Property not the same as instance field
• Setter can set fields, then call repaint
• Getter can query database
Property Syntax

- Not Java :-(
- C#, JavaScript, Visual Basic
- b.propertyName = value
- calls setter
- variable = b.propertyName
- calls getter
Java Naming Conventions

- property = pair of methods
- public X getPropertyName()
- public void setPropertyName(X newValue)
- Replace propertyName with actual name
  - (e.g. getColor/setColor)
- Exception for boolean properties:
  - public boolean isPropertyName()
- Decapitalization hokus-pokus:
  - getColor -> color
  - getURL -> URL
Editing Beans in a Builder Tool

- Use wizard to make empty frame
Editing Beans in a Builder Tool

• Add button to frame, edit button with property sheet
Packaging a Bean

• Compile bean classes:

001: import java.awt.*;
002: import java.awt.geom.*;
003: import javax.swing.*;
004:
005: /**
006:   * A component that draws a car shape.
007: */
008: public class CarBean extends JComponent
009: {
010:   /**
011:    * Constructs a default car bean.
012:   */
013:   public CarBean()
014:   {
015:     x = 0;
016:     y = 0;
017:     width = DEFAULT_CAR_WIDTH;
018:     height = DEFAULT_CAR_HEIGHT;
019:   }
/**
 * Sets the x property.
 * @param newValue the new x position
 */
public void setX(int newValue)
{
    x = newValue;
    repaint();
}

/**
 * Gets the x property.
 * @return the x position
 */
public int getX()
{
    return x;
}
Packaging a Bean

• Create manifest file: CarBean.mf
  Manifest-Version: 1.0
  Name: CarBean.class
  Java-Bean: True

• Run the JAR tool:
  jar cvfm CarBean.jar CarBean.mf *.class

• Import JAR file into builder environment
Composing Beans

- Make new frame
- Add car bean, slider to frame
- Edit stateChanged event of slider
- Add handler code
  ```java
carBean1.setX(jSlider1.getValue());
```
- Compile and run
- Move slider: the car moves
Composing Beans
End