CSE 373

Review of Java

slides created by Marty Stepp
also based on course materials by Stuart Reges

http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Summary

• These slides contain material about objects, classes, and object-oriented programming in Java.

• We won't be covering these slides in lecture, but they contain material you are expected to remember from CSE 142 and 143.

• For additional review material, consult Ch. 1-6 of *Core Java*.
Primitives vs. objects; value and reference semantics
A swap method?

- Does the following swap method work? Why or why not?

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```
Value semantics

- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```java
int x = 5;
int y = x;  // x = 5, y = 5
y = 17;     // x = 5, y = 17
x = 8;      // x = 8, y = 17
```
Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

```java
int[] a1 = {4, 15, 8};
int[] a2 = a1;  // refer to same array as a1
a2[0] = 7;
System.out.println(Arrays.toString(a1));  // [7, 15, 8]
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ a1 \rightarrow \begin{array}{c}
\text{index} \\
\hline
0 & 1 & 2 \\
\end{array} \begin{array}{c}
\text{value} \\
\hline
7 & 15 & 8 \\
\end{array} \rightarrow a2 \]
Arrays and objects use reference semantics. Why?

- efficiency. Copying large objects slows down a program.
- sharing. It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1; // same window
panel2.setBackground(Color.CYAN);
```
Objects as parameters

• When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  ▪ If the parameter is modified, it *will* affect the original object.

```java
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```
Arrays as parameters

- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```java
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Output:
  
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>252</td>
<td>334</td>
<td>190</td>
</tr>
</tbody>
</table>

```java
doubler
```

- Output:
  
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>252</td>
<td>334</td>
<td>190</td>
</tr>
</tbody>
</table>
Arrays pass by reference

- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```java
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Output:
  
  
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>252</td>
<td>334</td>
<td>190</td>
</tr>
</tbody>
</table>

  
  a

  iq

  
  [252, 334, 190]
Classes and Objects
Objects

- **object**: An entity that encapsulates data and behavior.
  - *data*: variables inside the object
  - *behavior*: methods inside the object

- You interact with the methods; the data is hidden in the object.

- Constructing (creating) an object:
  
  ```
  Type objectName = new Type(parameters);
  ```

- Calling an object's method:
  
  ```
  objectName . methodName (parameters);
  ```
Classes

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details. Objects and classes provide abstraction in programming.
Blueprint analogy

iPod blueprint

**state:**
current song
volume
battery life

**behavior:**
power on/off
change station/song
change volume
choose random song

---

**iPod #1**

**state:**
song = "1,000,000 Miles"
volume = 17
battery life = 2.5 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song

---

**iPod #2**

**state:**
song = "Letting You"
volume = 9
battery life = 3.41 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song

---

**iPod #3**

**state:**
song = "Discipline"
volume = 24
battery life = 1.8 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song
import java.awt.*;
...
Point p1 = new Point(5, -2);
Point p2 = new Point(); // origin (0, 0)

• Data:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point's x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point's y-coordinate</td>
</tr>
</tbody>
</table>

• Methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setLocation(x, y)</td>
<td>sets the point's x and y to the given values</td>
</tr>
<tr>
<td>translate(dx, dy)</td>
<td>adjusts the point's x and y by the given amounts</td>
</tr>
<tr>
<td>distance(p)</td>
<td>how far away the point is from point p</td>
</tr>
</tbody>
</table>
The class (blueprint) describes how to create objects.

Each object contains its own data and methods.

- The methods operate on that object's data.
Clients of objects

- **client program**: A program that uses objects.
  - **Example**: `Bomb` is a client of `DrawingPanel` and `Graphics`.

```java
Bomb.java (client program)
public class Bomb {
    public static void main(String[] args) {
        new DrawingPanel(...)
        new DrawingPanel(...)
        ...
    }
}
```

```java
DrawingPanel.java (class)
public class DrawingPanel {
    ...
}
```
Fields

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.

- Declaration syntax:

  ```java
  private type name;
  ```

  - Example:

    ```java
    public class Point {
      private int x;
      private int y;

      ...
    }
    ```
Encapsulation

- **encapsulation**: Hiding implementation details from clients.
  - Encapsulation enforces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data
Benefits of encapsulation

• Abstraction between object and clients

• Protects object from unwanted access
  ▪ Example: Can't fraudulently increase an Account's balance.

• Can change the class implementation later
  ▪ Example: Point could be rewritten in polar coordinates \((r, \theta)\) with the same methods.

• Can constrain objects' state (invariants)
  ▪ Example: Only allow Accounts with non-negative balance.
  ▪ Example: Only allow Dates with a month from 1-12.
Instance methods

- **instance method (or object method):** Exists inside each object of a class and gives behavior to each object.

```java
public type name(parameters) {
    statements;
}
```

- same syntax as static methods, but without **static** keyword

Example:

```java
public void translate(int dx, int dy) {
    x += dx;
    y += dy;
}
```
The implicit parameter

- **implicit parameter:**
The object on which an instance method is being called.

  - If we have a `Point` object `p1` and call `p1.translate(5, 3);`
    the object referred to by `p1` is the implicit parameter.

  - If we have a `Point` object `p2` and call `p2.translate(4, 1);`
    the object referred to by `p2` is the implicit parameter.

  - The instance method can refer to that object's fields.
    - We say that it executes in the context of a particular object.
    - `translate` can refer to the `x` and `y` of the object it was called on.
Categories of methods

• **accessor**: A method that lets clients examine object state.
  - Examples: `distance`, `distanceFromOrigin`
  - often has a non-`void` return type

• **mutator**: A method that modifies an object's state.
  - Examples: `setLocation`, `translate`

• **helper**: Assists some other method in performing its task.
  - often declared as private so outside clients cannot call it
The `toString` method
tells Java how to convert an object into a `String` for printing

```java
public String toString() {
    code that returns a String representing this object;
}
```

- Method name, return, and parameters must match exactly.

- Example:
  ```java
  // Returns a String representing this Point.
  public String toString() {
      return "(" + x + ", " + y + ")";
  }
  ```
Constructors

- **constructor**: Initializes the state of new objects.

  ```java
  public type(parameters) {
    statements;
  }
  ```

  - runs when the client uses the `new` keyword
  - no return type is specified; implicitly "returns" the new object

  ```java
  public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
      x = initialX;
      y = initialY;
    }
  }
  ```
Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

**Example:** A `Point` constructor with no parameters that initializes the point to (0, 0).

```java
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```
The keyword **this**

- **this**: Refers to the implicit parameter inside your class. *(a variable that stores the object on which a method is called)*

  - Refer to a field: `this.field`
  - Call a method: `this.method(parameters);`
  - One constructor `this(parameters);` can call another:
Calling another constructor

```java
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor
Comparing objects for equality and ordering
Comparing objects

• The `==` operator does not work well with objects.
  
  `==` compares references to objects, not their state.
  It only produces `true` when you compare an object to itself.

Point `p1 = new Point(5, 3);`
Point `p2 = new Point(5, 3);`
Point `p3 = p2;`

```java
// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true
```

![Diagram showing Point objects and their comparison](image-url)
The equals method

• The `equals` method compares the state of objects.

```java
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

• But if you write a class, its `equals` method behaves like `==`

```java
if (p1.equals(p2)) { // false :-(
    System.out.println("equal");
}
```

- This is the default behavior we receive from class `Object`.
- Java doesn't understand how to compare new classes by default.
The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:
    ```java
    public int compareTo(String other)
    ```

- A call of `A.compareTo(B)` will return:
  - a value < 0 if `A` comes "before" `B` in the ordering,
  - a value > 0 if `A` comes "after" `B` in the ordering,
  - or 0 if `A` and `B` are considered "equal" in the ordering.
Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```java
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) { // true
    ...
}
```

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (a &lt; b)</code> {</td>
<td><code>if (a.compareTo(b) &lt; 0)</code> {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>if (a &lt;= b)</code> {</td>
<td><code>if (a.compareTo(b) &lt;= 0)</code> {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>if (a == b)</code></td>
<td><code>if (a.compareTo(b) == 0)</code> {</td>
</tr>
<tr>
<td>{ ...</td>
<td>...</td>
</tr>
<tr>
<td><code>if (a != b)</code></td>
<td><code>if (a.compareTo(b) != 0)</code> {</td>
</tr>
<tr>
<td>{ ...</td>
<td>...</td>
</tr>
<tr>
<td><code>if (a &gt;= b)</code></td>
<td><code>if (a.compareTo(b) &gt;= 0)</code> {</td>
</tr>
<tr>
<td>{ ...</td>
<td>...</td>
</tr>
<tr>
<td><code>if (a &gt; b)</code></td>
<td><code>if (a.compareTo(b) &gt; 0)</code> {</td>
</tr>
<tr>
<td>{ ...</td>
<td>...</td>
</tr>
</tbody>
</table>
compareTo and collections

• You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```java
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan"); // 3
```

• Java's `TreeSet/Map` use `compareTo` internally for ordering.

```java
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```
Comparable (10.2)

public interface Comparable<E> {
    public int compareTo(E other);
}

• A class can implement the Comparable interface to define a natural ordering function for its objects.

• A call to your compareTo method should return:
  a value < 0 if this object comes "before" the other object,
  a value > 0 if this object comes "after" the other object,
  or 0 if this object is considered "equal" to the other.

• If you want multiple orderings, use a Comparator instead (see Ch. 13.1)
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;  // same x, smaller y
        } else if (y > other.y) {
            return 1;  // same x, larger y
        } else {
            return 0;  // same x and same y
        }
    }
}
subtraction trick - Subtracting related numeric values produces the right result for what you want compareTo to return:

```java
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;  // different x
    } else {
        return y - other.y;  // same x; compare y
    }
}
```

The idea:
- if \( x > \text{other}.x \), then \( x - \text{other}.x > 0 \)
- if \( x < \text{other}.x \), then \( x - \text{other}.x < 0 \)
- if \( x == \text{other}.x \), then \( x - \text{other}.x == 0 \)

\[\text{NOTE: This trick doesn't work for doubles (but see } \text{Math}.\text{signum)}\]
compareTo tricks 2

• *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```java
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

• *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```java
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```
Inheritance
Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - A way to share/reuse code between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - Gets a copy of every field and method from superclass
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.
Inheritance syntax

```
public class name extends superclass {

  Example:

  public class Lawyer extends Employee {
    ...
  }

  • By extending Employee, each Lawyer object now:
    ▪ receives a copy of each method from Employee automatically
    ▪ can be treated as an Employee by client code

  • Lawyer can also replace ("override") behavior from Employee.
```
Overriding Methods

• `override`: To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```java
public class Lawyer extends Employee {
    // overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```
The super keyword

- A subclass can call its parent's method/constructor:

  ```java
  super.method(parameters) // method
  super(parameters); // constructor
  ```

```java
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```
Subclasses and fields

```java
public class Employee {
    private double salary;
    ...
}

public class Lawyer extends Employee {
    ...
    public void giveRaise(double amount) {
        salary += amount; // error; salary is private
    }
}
```

• Inherited private fields/methods cannot be directly accessed by subclasses. *(The subclass has the field, but it can't touch it.)*
  - How can we allow a subclass to access/modify these fields?
Protected fields/methods

```java
protected type name; // field
protected type name(type name, ..., type name) {
    statement(s); // method
}
```

- a **protected field** or **method** can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same "package" (discussed later)
  - useful for allowing selective access to inner class implementation

```java
public class Employee {
    protected double salary;
    ...
}
```
Inheritance and constructors

• If we add a constructor to the Employee class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
  symbol : constructor Employee()
  location: class Employee
public class Lawyer extends Employee {
  ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
Inheritance and constructors

• Constructors are not inherited.
  ▪ Subclasses don't inherit the `Employee(int)` constructor.
  ▪ Subclasses receive a default constructor that contains:

    ```java
    public Lawyer() {
        super(); // calls Employee() constructor
    }
    ```

• But our `Employee(int)` replaces the default `Employee()`.
  ▪ The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.
Calling superclass constructor

\texttt{super(parameters);} \\

- Example:
  \begin{verbatim}
  public class Lawyer extends Employee {
      public Lawyer(int years) {
          \texttt{super(years);}  // calls Employee c'tor
      }
      ...
  }
  \end{verbatim}

- The \texttt{super} call must be the first statement in the constructor.
Polymorphism
Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.
Coding with polymorphism

• A variable of type $T$ can hold an object of any subclass of $T$.

  \[
  \text{Employee } ed = \text{ new Lawyer();}
  \]

  ▪ You can call any methods from the \text{Employee} class on \text{ed}.

• When a method is called on \text{ed}, it behaves as a \text{Lawyer}.

  \[
  \text{System.out.println(} \text{ed.getSalary()} \text{);} \quad \text{// 50000.0}
  \]

  \[
  \text{System.out.println(} \text{ed.getVacationForm()} \text{);} \quad \text{// pink}
  \]
Polymorphic parameters

• You can pass any subtype of a parameter's type.

```java
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee e) {
    System.out.println("pay : "+e.getSalary());
    System.out.println("vdays: "+e.getVacationDays());
    System.out.println("vform: "+e.getVacationForm());
    System.out.println();
}

OUTPUT:
pay : 50000.0   pay : 50000.0
vdays: 15       vdays: 10
vform: pink     vform: yellow
```
Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```java
public static void main(String[] args) {
    Employee[] e = {new Lawyer(), new Secretary(),
                   new Marketer(), new LegalSecretary()};

    for (int i = 0; i < e.length; i++) {
        System.out.println("pay : "+ e[i].getSalary());
        System.out.println("vdays: "+ i].getVacationDays());
    }
}
```

Output:

<table>
<thead>
<tr>
<th>pay</th>
<th>vdays</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000.0</td>
<td>15</td>
</tr>
<tr>
<td>60000.0</td>
<td>10</td>
</tr>
<tr>
<td>50000.0</td>
<td>10</td>
</tr>
<tr>
<td>55000.0</td>
<td>10</td>
</tr>
</tbody>
</table>
Casting references

- A variable can only call that type's methods, not a subtype's.

```java
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; in Employee
ed.sue();  // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue`.

- To use `Lawyer` methods on `ed`, we can type-cast it.

```java
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();  // ok

((Lawyer) ed).sue();  // shorter version
```
More about casting

• The code crashes if you cast an object too far down the tree.

```java
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi"); // ok
((LegalSecretary) eric).fileLegalBriefs(); // error
// (Secretary doesn't know how to file briefs)
```

• You can cast only up and down the tree, not sideways.

```java
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi"); // error
```

• Casting doesn't actually change the object's behavior. It just gets the code to compile/run.

```java
((Employee) linda).getVacationForm(); // pink
```
Interfaces
Shapes example

• Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.

• Certain operations are common to all shapes:
  ▪ perimeter: distance around the outside of the shape
  ▪ area: amount of 2D space occupied by the shape

  ▪ Every shape has these, but each computes them differently.
Shape area and perimeter

- Circle (as defined by radius $r$):
  - area = $\pi r^2$
  - perimeter = $2 \pi r$

- Rectangle (as defined by width $w$ and height $h$):
  - area = $wh$
  - perimeter = $2w + 2h$

- Triangle (as defined by side lengths $a$, $b$, and $c$)
  - area = √($s (s - a) (s - b) (s - c)$)
    where $s = \frac{1}{2} (a + b + c)$
  - perimeter = $a + b + c$
Common behavior

- Suppose we have 3 classes Circle, Rectangle, Triangle.
  - Each has the methods perimeter and area.

- We'd like our client code to be able to treat different kinds of shapes in the same way:
  - Write a method that prints any shape's area and perimeter.
  - Create an array to hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
  - Make a DrawingPanel display many shapes on screen.
Interfaces

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A *Lawyer* can be treated as an *Employee* and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A *Rectangle* object can be treated as a *Shape* but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}

Example:
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}

- Saved as Shape.java

- abstract method: A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.
Implementing an interface

```java
public class name implements interface {
    ...
}
```

- A class can declare that it "implements" an interface.
  - The class promises to contain each method in that interface.
    (Otherwise it will fail to compile.)

- Example:
  ```java
  public class Bicycle implements Vehicle {
    ...
  }
  ```
public class Banana implements Shape {
    // haha, no methods! pwned
}

- If we write a class that claims to be a Shape but doesn't implement area and perimeter methods, it will not compile.

    Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
public class Banana implements Shape {
    ^
Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
  - they allow **polymorphism**
    (the same code can work with different types of objects)

```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
}

... 
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```
Abstract Classes
List classes example

• Suppose we have implemented the following two list classes:

  - ArrayList

    | index | 0 | 1 | 2 |
    |-------|---|---|---|
    | value | 42 | -3 | 17 |

  - LinkedList

    - We have a List interface to indicate that both implement a List ADT.
    - Problem:
      • Some of their methods are implemented the same way (redundancy).
Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
  - `add(value)`
  - `contains`
  - `isEmpty`

- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?
Abstract classes (9.6)

- **abstract class**: A hybrid between an interface and a class.
  - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)

- What goes in an abstract class?
  - implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - declare generic behaviors that subclasses implement (interface role)
Abstract class syntax

```java
// declaring an abstract class
public abstract class name {
  ...

  // declaring an abstract method
  // (any subclass must implement it)
  public abstract type name(parameters);
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type
Abstract and interfaces

• Normal classes that claim to implement an interface must implement all methods of that interface:

```java
public class Empty implements List {} // error
```

• Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```java
public abstract class Empty implements List {} // ok
public class Child extends Empty {} // error
```
An abstract list class

// Superclass with common code for a list of integers.
public abstract class AbstractList implements List {
    public void add(int value) {
        add(size(), value);
    }

    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
}

public class ArrayList extends AbstractList { ...

public class LinkedList extends AbstractList { ...

Abstract class vs. Interface

• Why do both interfaces and abstract classes exist in Java?
  ▪ An abstract class can do everything an interface can do and more.
  ▪ So why would someone ever use an interface?

• Answer: Java has single inheritance.
  ▪ can extend only one superclass
  ▪ can implement many interfaces
  ▪ Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.
Inner Classes
Inner classes

• **inner class**: A class defined inside of another class.
  - can be created as *static* or non-static
  - we will focus on standard non-static ("nested") inner classes

• usefulness:
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object
// outer (enclosing) class
public class name {
    ...
    
    // inner (nested) class
    private class name {
        ...
    }
}

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as `OuterClassName.this`
Example: Array list iterator

```java
public class ArrayList extends AbstractList {
    ...
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<Integer> {
        private int index;  // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {
            return index < size();
        }

        public E next() {
            index++; return get(index - 1);
        }

        public void remove() {
            ArrayList.this.remove(index - 1); index--;
        }
    }
}
```