Introduction to Java

Hal Perkins

Goals

- Survey of major Java language and library features
- Orientation – not comprehensive
  - No way anyone actually understands all of the libraries(!)
  - Part of the job of learning a new language/environment is to learn how to find information
- Ask lots of questions!

Overview

- A bit of history
- Classes and objects
- Core Java language
- Collection classes
- Class relationships – inheritance and interfaces
- Packages & scope
- Exception handling
- GUI basics (AWT & Swing)
- Threads

References (1)

- Way too many to count. Here are a couple of useful places to start (i.e., I’ve found them useful)
- From Sun
  - Java SDK and documentation (java.sun.com)
    (Good “how to do it” topic orientation)
  - The Java Programming Language by Arnold, Gosling, and Holmes (A-W, 3rd edition)
    (Language and container classes primarily)

References (2)

- Overview of Object-Oriented Programming
  - Understanding Object-Oriented Programming with Java by Tim Budd (Addison-Wesley)
- Longer tutorial on language and libraries
  - Learning Java by Niemeyer & Knudsen (O’Reilly)
- Look-it-up references
  - Java in a Nutshell (core language and libraries)
  - Java Foundation Classes in a Nutshell (AWT, Swing)
  - Java Examples in a Nutshell all by David Flanagan (O’Reilly)

Some History

- 1993 Oak project at Sun
- 1995 Oak becomes Java; web happens
- 1996 Java 1.0 available
- 1997 (March) Java 1.1 - some language changes, much larger library, new event handling model
- 1997 (September) Java 1.2 beta – huge increase in libraries including Swing, new collection classes, J2EE
- 1998 (October) Java 1.2 final (Java2!)
- 2000 (April) Java 1.3 final
- early 2002 Java 1.4 final (assert)
- 2002-2003 Java 1.5 (parameterized types?)
Introduction to Java

Design Goals

- Support secure, high-performance, robust applications running as-is on multiple platforms and over networks
- "Architecture-neutral", portable, allow dynamic updates and adapt to new environments
- Look enough like C++ for programmer comfort
- Support object-oriented programming
- Support concurrency (multithreading)
- Simplicity

Hello World in Java

```java
public class HelloWorld {
    public static void main (String [] args) {
        System.out.println("Hello World");
    }
}
```

It’s all about objects

- Java is a purely object-oriented language (well, almost)
- Fundamental unit of a program is a class
- Instances of classes are objects
  - May be helpful to think of objects receiving messages and replying to them instead of calling methods and returning values
- Java includes an incredibly rich set of libraries

Classes

- Everything in Java is a member of some class
  - No external (global) functions or variables
  - Classes may contain methods and data members
  - Class members may be
    - non-static: one copy for each instance of the class (one copy per object)
    - static: single copy associated with the class, not with any specific instances.

Hello World Revisited

```java
public class HelloWorld {
    public static void main (String [] args) {
        System.out.println("Hello World");
    }
}
```

Command Line Arguments

(If you like this sort of thing – useful for things like file names)

```java
public class PrintArgs {
    public static void main (String [] args) {
        for (int k=0; k < args.length; k++)
            System.out.print(args[k] + " ");
        System.out.println();
    }
}
```

%javac PrintArgs.java
%java PrintArgs Testing one, two, three
Testing one, two, three
**Primitive Data Types**
- 2's complement signed integer
- int (32 bits), byte (8), short (16), long (64)
- int constants are normally type int
- IEEE floating point
- double (64 bits), float (32)
- floating constants are normally type double
- Unicode characters: char (16 bits)
- Logical: boolean
  - constants are true, false
  - not ints
- None of these are “implementation-defined” or “implementation-dependent”

**Arithmetic and assignment**
- Almost same as C/C++
- int k = 17; boolean maybe; double x=42.0
- k = 2 * k; maybe = k > 17;
- Declaration initializers are optional. If omitted,
  - Fields in class instances initialized to 0, false, null.
  - Local vars in methods not initialized by default; compiler
    complains if use before initialization is possible
  - Automatic coercion if no information lost
  - double y = k + 6;
- Explicit cast required to indicate possible information
  - loss is intended
  - k = (int) (x * 1.3 / (x-2.0))

**Basic statements (1)**
- if, while, for, and switch work as in C/C++
- if (x < y) {
  tmp=x; x=y; y=tmp;
} else {
  x=0;
}
- while (k < n && a[k] != x) {
  k++;
}
- Use {} to create compound statements
- Creates a new scope
- Style point – always use these

**Basic statements (2)**
- Logical && and || are short-circuit
- switch requires explicit break if fall-through to
  next case is not desired; if default case is not
  provided and no case label matches, execution
  silently proceeds with next statement.

**Class Definitions**
- Basic use is to define template for instances
  /** Simple, tiny example class
   * @author Al Gaulle
   * @version 6068 */
  public class Blob {
    private int val; // Blob state
    /* construct new Blob with given initial value */
    public Blob(int val) {
      this.val = val;
    }
  }
- /** ... */ comments are JavaDoc comments; JavaDoc processor
  generates API docs (html) automatically from this information

**Class Definitions (continued)**
- /** Set the value of this blob
   * @param val new value for this blob */
  public void setVal(int val) { this.val = val; }
- /** Access this Blob's value
   * @return current value of this blob */
  public int getVal() { return val; }
- /** yield string representation of this Blob */
  public String toString() {
    return Blob: val = " + val ;
  }
- toString() automatically used to cast object to String
  when used in context that requires it
- System.out.println(theBlob);
Constructors

- Constructor(s) can be provided to initialize objects when they are created. Constructors can be overloaded and can delegate to other constructors.

```java
class Blob {
    private int val;
    /** construct Blob with given initial value */
    Blob (int initial) { val = initial; }
    /** construct Blob with default initial value */
    Blob () { this(17); }
    ...
}
```

Instance Creation and References

- Except for primitive types (int, double, boolean, char, etc) all variables are references. Objects are only created by explicit allocation on the heap (with new).

```java
Blob bob; // no blob allocated yet
bob = new Blob(); // Blob allocated here
bob.setVal(42);
int k = bob.getVal();
System.out.println("bob is "+bob);
```

References and Methods

- Dot notation is used to select methods and fields; implicit dereference (no -> as in C/C++).
- No pointer arithmetic; no & operator to generate the address of arbitrary variable; can't create pointers from random bits.
- "Java has no pointers"
- All method parameters are call-by-value (copy of primitive value or object reference)
- Methods can be overloaded (different methods with same name but different number or types of parameters).

Object References

- A variable declared as class X has type "reference to X". No object is created by such a declaration.
- Declaration and object creation can be combined.
- The constant null belongs to all reference types and refers to nothing.
- If reference r is null, then selecting a field or method from r (r.fieldname) throws a NullPointerException.
- Storage occupied by an object is dynamically reclaimed when the object is no longer accessible (automatic garbage collection).

Visibility

- Class members can be preceded by a qualifier to indicate accessibility
  - public - accessible anywhere the class can be accessed
  - private - only accessible inside the class
  - If nothing is specified, the field can be referenced anywhere in the same package (more later).
  - protected - same as package visibility, and also visible in classes that extend this class.

Static Methods and Fields

- static class members are most commonly used for data and methods that are not naturally associated with a specific class instance.

```java
class Math { // standard Java Math class
    static double sqrt(double x) { ... }
    static double sin(double x) { ... }
}
```

```java
distance = Math.sqrt(x*x + y*y);
```
Symbolic Constants

- A class member may be qualified as final.
  - For data, it means the variable must be initialized when declared and cannot be changed after that.
  - For methods, it means the method cannot be overridden in a derived class.
  - The compiler can take advantage of this to inline the constant value or method code.

```java
class Math {
    // standard Java Math class
    static final double PI = 3.1415926535;
    static final double E = 2.71828182845;
}
```

```
area = Math.PI * r * r;
```

Arrays

- Arrays are dynamically allocated. Declaring an array variable only creates a reference variable; it does not actually allocate the array.

```java
double[] a;
a = new double[6];
for (int k = 0; k < 6; k++)
a[k] = 2 * k;
```

Array Notes

- Arrays are 0-origin, as in C/C++
- Arrays are also objects, with one constant member
  - If a is an array, a.length is its length
  - An IndexOutOfBoundsException is thrown if a subscript is < 0 or >= the array length.
- The brackets indicating an array type may also appear after the variable name, as in C/C++
  ```java
  int a[] = new int[100];
  ```

2-D Arrays

- A 2-D array is really a 1-D array of references to 1-D array rows. The allocation

```java
double[][] matrix = new double[10][20];
```

is really shorthand for

```java
double[][] matrix = new double[10][ ];
for (int k = 0; k < 10; k++)
matrix[k] = new double[20];
```

- Array elements are accessed in the usual way

```java
for (int r = 0; r < 10; r++)
for (int c = 0; c < 20; c++)
matrix[r][c] = 0.0;
```

Arrays of Objects

- If the array elements have an object type, the objects must be created individually.

```java
Blob[] list;
list = new Blob[10];
for (int k = 0; k < 10; k++)
list[k] = new Blob();
```

Strings

- A character string “abc” is an instance of class String, and is a read-only constant.
- Strings are objects; they are not arrays of chars.
- There is no visible ‘\0’ byte at the end
- If s is a string, s.length() is its length, and s.charAt(k) is the character in position k.
- Class String includes many useful string processing functions (search, substring, ...).
- + concatenates strings (“hello” “there”)
Derived Classes

- A class definition may extend (be derived from) a single parent class (single inheritance).
  ```java
class Point {
    private int h, v; // instance vars
    public Point(int x, int y) { h = x; v = y; } // constructor
}

class ColorPoint extends Point {
    private Color c; // additional instance var
    public ColorPoint(int x, int y, Color c) { super(x, y); this.c = c; }
}
```

Derived Classes (cont.)

- All of the usual object-oriented notions are supported, including inheritance of fields and methods from superclasses and overriding.
- Inside a method, `this` refers to the current object; `super` refers to the current object viewed as an instance of the parent class.
- There is a single class `Object` at the root of the class hierarchy.
  - If a class declaration does not explicitly extend some class, it implicitly extends `Object`.

Abstract Classes

- An abstract class is one that contains an abstract method or is declared to be abstract
  ```java
  abstract class ExtendMe {
    ...
    public abstract mustOverride(...);
  }
  
  A final class may not be extended further.
  
  Pop quiz: can a class be both final and abstract?
```

Wrapper Classes for Basic Types

- For each basic type (int, double, etc.) there is a corresponding class (Integer, Double, etc.) that is an object version of that type.
- `Integer(17)` is an object representation of the int `17`.
- Particularly useful with container classes that can only hold objects (ArrayList, HashTable, etc.)
- Wrapper classes also contain many useful utility functions and constants.
  ```java
  if (k < (Integer.MAX_VALUE/10)) ...
  if (Character.isLowerCase(ch)) ...
  ```

Interfaces

- Interfaces allow specification of constants and methods independently of the class hierarchy.
- Interfaces may extend other interfaces, but since they are pure specification, no implementation is inherited.
  ```java
  interface AbsType {
    static final int one = 1;
    static final int two = 2;
    void f(int a, int b);
    double g();
  }
  ```

Interfaces (cont)

- A class may implement as many interfaces as desired.
- Full implementation of all methods in the interface must be provided by the class or inherited from a parent class. Nothing is inherited from the interface.
- Gives most of the useful effects of multiple inheritance
  - Allows otherwise unrelated classes to implement common behavior.
  - Some interfaces are “markers” - identify classes that can be used in certain contexts:
    - Widely used for event handling in the Java user interface (MouseEventListener, ActionListener, many others)
Interfaces and Abstract Types

- Both define a new type
- In real systems, any important type should be defined by an interface
  - Specifies the type without tying to an implementation
- Often, should provide a model implementation of the interface in an abstract or concrete class
- Programmer has choice of implementing the interface or using (maybe extending) the abstract class

Container Classes

- The Java container classes are a good example of the use of interfaces and classes
- Example: interface List – ordered list of objects
  - operations: add(obj), size(), get(k), set(k, obj), many, many more
- Implementations
  - ArrayList – ordered list with O(1) access to elements
  - LinkedList – ordered list implemented with doubly-linked list
- Other kinds of collections: set, map (table), etc.

Iterators

- This generalizes the notion of
  
  for (int k = 0; k < a.size; k++) { process a[k] }

- Collections provide an iterator() method, which yields an object that provides element-by-element access to items in the collection
  
  ArrayList theList = new ArrayList();
  //... code to fill theList omitted
  Iterator it = theList.iterator();
  while (it.hasNext()) {
    Object o = it.next();
    process o (may need to cast to specific element type)
  }

Object Compare and Copy

- Default assignment and comparison only copies or compares references (shallow operations)
  
  Blob b = new Blob();
  Blob c = new Blob();
  if (b==c) {
    System.out.println("Something wrong");
  }
  c = b;
  b.setVal(100);
  System.out.println( c.getVal() );

Defining Compare and Copy

- Intended meaning of a.equals(b) is that a and b are "equal" in sense appropriate for the class of a and b.
  - Tricky semantics if class is extended and fields are added/overridden
- b.clone() should create a new "copy" of b and return a reference to it.
- All classes inherit equals and clone from Object
  - Default versions do a shallow compare/copy
  - Override if a different compare/copy is desired
- To override clone, a class must also extend the Cloneable interface (this is purely a marker interface, has no methods or constants)

Exceptions

- Java has an extensive exception handling mechanism. Basic idea
  
  try {
    this Might Explode(x,y,z);
  } catch (Exception e) {
    <deal with the problem>
  }

- To generate an exception, execute
  
  throw new anExceptionClass(parameters);
  to cause the call chain to unwind until a catch clause that matches the thrown object is found.
Exceptions (cont)

- Multiple catch clauses can be used to selectively handle exceptions

```java
try {
    tryToReadData(x, y, z);
} catch (IOException e) {
    <deal with I/O problem>
} catch (Exception e) {
    <deal with other exceptions>
}
```

- If a method does something that might generate an exception, it must either handle it, or declare that it might throw that exception (throws clause).

Exceptions (cont)

- Classes of exceptions
  - Checked: things like IOException that result if an operation does not complete successfully
  - Unchecked: things that indicate programming errors or system failure (IndexOutOfBoundsException, NullPointerException)

- If a method does something that might generate a checked exception, it must either handle it, or declare that it might throw that exception (throws clause).

Packages

- Packages provide a way to partition the global class namespace.
- A class is placed in a package by including at the beginning of the class source file
- A class in another package can use items from a package by explicitly qualifying the item name
- A class in another package can use items from a package by importing names from the package

```java
package widget;

Blob b = new widget.Blob();
```

```java
import widget.*;

Blob b = new Blob();
```

- Package names are grouped into hierarchies by using package names with embedded dots

```java
java.lang, java.util, java.awt
```

- Import is not transitive (unlike C/C++ #include)
- Import only opens scope of given package, not subpackages
- If a class definition does not include a package statement, that class is part of a default anonymous package.
- Useful for small projects (like homework assignments)
- Good simplification – particularly because some programming environments require that the source code directory structure reflects the subpackage structure

Some Standard Library Packages

- `java.lang` – core classes (Math, String, System, Integer, Character, etc.)
- `java.util` – collections, date/time, random numbers
- `java.io` – input/output streams, files
- `java.net` – network I/O, sockets, URLs
- `java.awt` – basic (original) graphical user interface
- `java.awt.event` – GUI event handling
- `javax.swing` – sophisticated newer GUI built on top of AWT

Streams

- Stream = flow of data (bytes or characters)
- Can be associated with files, communication links, keyboard/screen/printer
- Many stream classes; most are designed to be used as wrappers that accept data and transform or filter it before passing it along
- Java 1.0: Byte streams with a few wrappers to handle ASCII text
- Java 1.1: Added text streams to handle Unicode properly
Stream Abstract Classes
- Byte streams: InputStream, OutputStream
- Character streams: Reader, Writer
- All Java stream classes are extensions of these (directly or indirectly)
- There are wrapper classes to convert between these
  - Historical note: console I/O streams (System.in, System.out, System.err) existed in Java 1.0, so these are InputStreams and OutputStreams, even though they really should be Readers and Writers

Basic Reader/Writer Operations
- Reader
  - int read(); // next Unicode character or -1 if EOF
  - int read(char[] cbuff); // read up to array capacity
  - All can throw IOExceptions
- Writer
  - int write(int c); // write character
  - int write(char[] cbuff); // write array of characters

FileStreams for Text
- Basic Classes: FileReader, FileWriter
- Several constructors
  - Open file with filename
  - Open file with File object

Low-Level File Copy
```java
class TediousCopy {
  public static void main(String[] args)
      throws IOException {
    FileReader inFile = new FileReader("input.txt");
    FileWriter outFile = new FileWriter("copy.txt");
    int ch; // current character
    ch = inFile.read();
    while(ch != -1) {
      outFile.write(ch);
      ch = inFile.read();
    }
    inFile.close();
    outFile.close();
  }
}
```

Buffered Input and Output
- Wrapper classes – data read from or written to basic source/sink stream objects; the wrapper objects transform the stream
- Classes available to handle newlines transparently
- BufferedReader – method ReadLine() – Returns string with next line of input, or null if EOF
- PrintWriter – methods print and println
  - Overloaded for primitive types and String
  - println emits end-of-line appropriate for host system after data written

Example: Copy Text Files (1)
```java
// open input file
FileReader inFile;
try {
  inFile = new FileReader("c:\input.txt");
} catch (IOException e) {
  System.err.println("Input file ouch: " + e );
}
BufferedReader in = new BufferedReader(inFile);
```
- Gotcha: need to use command line arguments or JFileChooser or something to avoid system-dependent file names in code
**Example: Copy Text Files (2)**

```java
// open input file
FileWriter outfile;
try {
inFile = new FileWriter("copy.txt");
} catch (IOException e) {
    System.err.println("Output file ouch: " + e);
}
PrintWriter out = new PrintWriter(outfile);
```

**Example: Copy Text Files (3)**

```java
try {
    String line = in.readLine( );
    while (line != null) {
        out.println(line);
        line = in.readLine( );
    }
} catch (IOException e) {
    System.err.println("ouch while copying: " + e);
} finally {
    in.close( );
    out.close( );
}
```

**User Interfaces – AWT and Swing**

- **AWT** – original GUI
  - Heavyweight objects – each AWT object (button, label, window) had corresponding native GUI object
  - Incomplete and awkward to program in places
- **Swing** – new GUI in Java 2 (JDK 1.2)
  - Lightweight components – everything except top-level windows implemented in Java
  - Extends AWT; keeps the Java 1.1 AWT event model
  - Much more complete library

**Components & Containers**

- Every AWT/Swing class ultimately extends `Component`
  - Contains dozens of basic methods
- Some components are containers – can contain other (sub-)components
- Top-level containers: JFrame, JDialog, JApplet
- Mid-level containers: JPanel, scroll panes, tool bars, …
- Basic components: JButton, JLabel, text fields, check boxes, lists, file choosers, …

**A Simple Swing Application**

```java
import java.awt.*;
import javax.swing.*;
// free-standing application w/Window
public class App extends JFrame {
    public void paintComponent(Graphics g) {
        // redraw screen when requested by window manager
    }
    //</main program -- create window etc.
    public static void main(String args[]){
        App frame = new App();
        frame.setVisible(true);
        continue processing
    }
}
```

**Java Application Notes**

- `paintComponent(Graphics g)` is called by the window manager as needed, i.e., asynchronously
  - Graphics parameter is the drawing context object
  - Supports drawing methods
    - `g.setColor(Color.Blue);`
    - `g.drawOval(40,30,100,150);`
  - Component can request redrawing by calling `repaint()`
    - Causes window manager to perform repaint when convenient for underlying windowing system
Event Handling

- User interface components generate events
- Objects (often other components) can register themselves to receive events of interest
- When an event happens, an appropriate method is called in all listeners (all registered objects)
- A listener object must implement the interface corresponding to the events, which means implementing all methods declared in the interface
- Need import java.awt.event.*;

Example: Track Mouse

```java
public class TrackMouse extends JFrame implements MouseMotionListener {
    // instance variables
    int locX = 100; // last mouse location
    int locY = 100;

    // constructor - register this object
    // to receive mouse move events
    public TrackMouse( ) {
        addMouseMotionListener(this);
    }

    // MouseMotionListener methods
    public void MouseMoved( ) { }
    public void MouseDragged(MouseEvent e) {
        locX = e.getX();
        locY = e.getY();
        repaint( );
    }

    // repaint screen
    public void paintComponent(Graphics g) {
        g.drawString("Here!", locX, locY);
    }
}
```

Example: Button

Most user-interface components need to be allocated, added to an appropriate container, and interested objects need to register to receive events.

```java
public class WatchButton extends JFrame implements ActionListener {
    // instance variables
    JButton belly; // the button

    // constructor - create button, add to this Frame
    // and register this object as a listener
    public WatchButton( ) {
        belly = new JButton("press me");
        getContentPane( ).add(belly);
        belly.addActionListener(this);
    }

    // react to button press
    public ActionPerformed(ActionEvent e) {
        if (e.getSource()==belly) {
            respond to button press
        }
    }
}
```

Example: Button (cont)

- The test isn’t strictly necessary if we know that belly is the only button that could generate the event
- Many other UI components (text boxes, dials, …) generate similar events. The event object contains details of the event (source, kind, data values, locations, etc.).
Layout Managers

- A Layout Manager is associated with every Container. The layout manager is responsible for positioning components in the container when the container is redrawn.
- Basic layout manager classes
  - FlowLayout - arranges components from left to right, top to bottom. Nothing Fancy
  - GridLayout - regularly spaced rows and columns
  - BorderLayout - Components can be placed in the Center, North, South, East, or West.
  - Useful trick: to place several controls in one of these places, create a Panel containing the controls, then place the Panel in one of the 5 BorderLayout locations.
- GridBagLayout - General constraint layout.

Layout Manager Example

- In the constructor for a Container
  ```java
  public SomeContainer() extends ... {
      ...  
      /** Construct new container */
      public SomeContainer() {
          JButton c = new JButton("cold");
          JButton w = new JButton("warm");
          setLayout(new BorderLayout());
          add(c, BorderLayout.CENTER);
          add(w, BorderLayout.SOUTH);
          ...  
      }
  }
  ```
- Also need to add listeners for the buttons, etc.

Threads

- Thread = Execution of one sequence of instructions (including function/method calls, conditionals, loops).
- Normal Java program executes in a thread created for main (application) or borrowed from the browser (applets).
- Class Thread can be used to create additional threads that execute concurrently.
- Each new thread is associated with (controlled by) a Thread object.

Single Thread Example

```java
class Foo {
    void run() {
        for (int i=0; i<100; i++)
            System.out.println("foo ");
    }
}
class Bar {
    public static void main(char[] args) {
        Foo foo = new Foo();
        foo.run();
        for (int i=0; i<100; i++)
            System.out.println("bar ");
    }
}
```  
- Prints 100 "foo"s followed by 100 "bar"s

Extending Class Thread

- Class Thread can be extended to create objects that run concurrently in their own thread.
- Execution begins in method run of the new class.
  ```java
  class Foo extends Thread {
      void run() {
          for (int i=0; i<100; i++)
              System.out.println("foo ");
      }
  }
  ```
- Foo.run overrides a (basically) empty method run in class Thread.

Concurrent Execution

- To begin concurrent execution, call method start of a Thread object. This sets up the new thread, then calls the object’s run method.
  ```java
  class Bar {
      public static void main(char[] args) {
          Foo foo = new Foo();
          foo.start();
          for (int i=0; i<100; i++)
              System.out.println("bar ");
      }
  }
  ```  
- Prints 100 "foo"s and 100 "bar"s in some unpredictable order
Uses for Threads

- Asynchronous or nonblocking I/O
  - Continue execution in one thread while waiting for I/O to complete or time out in another.
- Timers
  - Wait for an interval to expire, then cause something to happen (examples: animations; do something if the user doesn’t respond after a reasonable interval. …)
- Process multiple tasks simultaneously
  - Handle GUI in one thread while doing extended calculations in another.
- Parallel algorithms
  - If the JVM supports it, run parts of the computation concurrently on different processors.

Runnable Classes

- There are many situations where we want to execute a computation concurrently, but in a class that’s not a subclass of Thread.
- We still need a Thread object to create and control the thread.
- A thread can begin execution in any class that implements Runnable and contains a run method.

```java
public interface Runnable {
    public abstract void run();
}
```

Runnable Classes (cont.)

```java
public static void main(String[] args) {
    new FooBar().start();
}
```

Using Runnable

- This class executes one of its methods in a separate thread.

```java
class FooBar implements Runnable {
    public void foo() {
        for (int i=0; i<100; i++)
            System.out.println("foo ");
    }
    public void bar() {
        for (int i=0; i<100; i++)
            System.out.println("bar ");
    }
    public void run() {
        foo();
    }
}
```

Using Runnable (cont.)

```java
class C {
    int x, y;
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int sumXY() { return x+y; }
}
```

Synchronization

- Since threads may interleave execution in any order, we may need to control access to objects to ensure only one thread at a time can update related variables.

```java
class C {
    int x, y;
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int sumXY() { return x+y; }
}
```

```java
class C {
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int sumXY() { return x+y; }
}
```

- What happens if one thread executes sumXY while another thread is halfway through executing setXY on the same object?

```java
class C {
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int sumXY() { return x+y; }
}
```

Synchronized methods

- Every object has an associated lock
- We can require threads to acquire the lock before executing one of the object’s methods by declaring the method to be synchronized.
- A synchronized method automatically acquires the object’s lock when it is called. Other threads are blocked until the lock is released automatically when the synchronized method terminates.
synchronized methods

class C {
   int x,y;
   public synchronized void setXY(int x, int y) {
      this.x = x; this.y = y;
   }
   public synchronized int sumXY() { return x+y; }
}

- If some thread is executing setXY or sumXY, no other thread can execute either of those methods until the first thread releases the lock.
- Methods wait and notify are available to temporarily release the lock and regain it as needed.