Announcements

• The T.A.’s office hours will be posted on the web.
  – Monday - Wednesday: 12:00 - 13:30
  – McConnell 234 (Compilers Lab)

• Lectures from 1 to 8 have been posted on the web. However, lectures I have not yet given are subject to change.

• The tutorial on Java GUI will be given Thursday, January 22th at 18:15. The room will be announced shortly.
• O.O. Programming allows programmers to shift responsibility.

• Java has a rich set of abstraction building blocks:
  – Abstract classes (concrete)
  – Interfaces
  – Overloading
  – Overriding

• Design patterns are built from basic constructs.
Java programs consist of classes and interfaces.

- **Classes**
  - Define collections of procedures
  - Define new data types

- **Interfaces**
  - Define new data types / parts of data types
Objects & Variables

• All data is accessed by means of variables.
• Local variables (of methods) reside on run-time stack.
• Each variable has a type declaration.
  – Primitive types: values

    3 false c

  – All other types: objects
    – References to object on heap.
  – Predefined types in package java.lang (implicit import java.lang).
Objects & Variables (cont.)

- Primitive Variable

```java
int i = 6;
```

- Uninitialized Primitive Variable

```java
int j;
```

- Array of 5 primitives

```java
int [] a = {1,3,5,7,9};
```

- Empty Array of 3 primitives

```java
int [] b = new int[3];
```
• Reference to String Object

    String t; or String t = null;

• String object

    String s = new String("abcde");
Assignment (The $=$ symbol)

- Every object has an identity that is distinct from any other object.
- Assignment: copies values (primitive) or references.

```java
j = i; // copy value

b = a; // copy reference

t = s; // copy reference
```

- Reference assignment makes variables share objects.
- The symbol $==$ checks if two variables contain the same value (or reference).
- If objects become unreachable, storage will be reclaimed by the garbage collector.
Mutability

• The state of a *mutable* object can change.
  – Example: Arrays are mutable

  a: {1,3,5,7,9}

  a[2] = 9;

  a: {1,9,5,7,9}

• The state of *immutable* objects never changes.
  – Example: Strings are immutable

  t: String object of value "abcde"

  t = t + 'f'
Mutability

\[ t: \text{New string object of value "abcdef"} \]

- In other words, a new string object is created and referenced by \( t \).
- The old string object is discarded and will eventually be garbage collected.
Method Call Semantics

• Let us take the example:

\[ \text{myBook.readChapter}(x, y, \ldots); \]

• First, we evaluate \textit{myBook} for the class of the object whose method is being called (using dispatch).

• Then, we evaluate the expressions \(x, y, \ldots\) for actual parameter values.

• Then, we create an activation record on the run time stack containing:
  – formal parameters
  – local variables

• Then, we transfer control to first statement of target method.

• If \textit{myBook} is null, we get a \textit{NullPointerException}. 
Type Checking

• Java is *Strongly Typed*
  – The compiler checks that every assignment and every method call is type correct.
  – Variable declarations give type of variables.
  – Method headers define signatures: the set of argument and result types.

• Java is *type-safe*
  – Declarations and headers allow the compiler to determine the *apparent* type of any expression.
  – All array accesses are checked to be within bounds.

• Type mismatches cannot occur at run time (unlike C,C++ with union types & explicit deallocation).
Type Substitutability

- If $S$ is a subtype (subclass) of $T$, then objects of type $S$ are usable anywhere where $T$ is usable.
  - $S$ has all methods that $T$ has (enforced by compiler).
  - The methods in $S$ must behave the same way as the methods in $T$ (un-enforceable).

- All types are subtypes of $Object$ and understand:
  - boolean equals ($Object$ $o$)
  - String toString ()

- The actual type of an object (defined by creation) is guaranteed to be a subtype of the apparent type of the variable to which the object is assigned.

```java
Object o1 = "abc"; // String

Object o2 = {1,2,3}; // Array
```
Type Substitutability
Type Checking

- Compiler always works with apparent types:

  ```java
  Object o1 = "abc"; // actual type String

  Object o2 = {1,2,3}; // Array
  ```

- Therefore:

  ```java
  if (o1.equals("abc")) // legal

  if (o2.equals("abc")) // legal

  if (o1.length()) // illegal

  String s = o1; // illegal
  ```
Type Checking

• You can get around this by type-casting:

```java
if (((String) o1.length())) // legal

String s = (String) o1; // legal
```

• Is safe because type-check occurs at run time (not like C).
Type Conversion

- Type casting changes the apparent type of an expression, but does \emph{not compute or modify} values.

- Type conversion changes a type into another type and typically \emph{computes} the new value.

- Java defines implicit conversions on primitive types:
  - Chars are widened to numeric types:
    
    ```
    char c = 'a';
    int n = c;
    float f = n;
    ```
    
    - int is widened to long
    - long is widened to float
Overloading & conversion

• Method overloading: method with same name but different signature.

static int comp (int, long) // definition 1
static int comp (long, int) // definition 2
static int comp (long, long) // definition 3

• Consider the following declarations:

int x;
long y;
Overloading & conversion

- The actual method called is the most-specific:
  - \texttt{comp (x,y)} : definition 1
  - \texttt{comp (y,y)} : definition 3
  - \texttt{comp (x,x)} : compile-time error because neither definition 1 or 2 is most-specific

- All these rules apply to objects and subtypes.
Method dispatch

• Consider this piece of code:

```java
String t = "ab";

Object o = t + "c"; // concatenation

String r = "abc";

boolean b = o.equals(r);
```

• We want to find out whether b has the value abc.

• String defines equals(object o) to compare character per character.

• However, the standard definition of equals(object o) in Object compares object identity (==).
Method dispatch

- Fortunately, dispatch is based on actual type (of the receiver object), not on apparent type.
- We get the correct result.
Packages

Classes and Interfaces are grouped in Packages.

- To Declare:

  ```
  package myPackage;
  ```

  ```
  public class myClass01 {...
  ```

- To use:

  ```
  ... myPackage.myClass01...
  ```

- or:

  ```
  ```
Packages

import myPackage.*;

...myClass01...
Packages

• Provide encapsulation
  – only public classes, interfaces, methods & fields are visible outside the package
  – all other declarations are only visible within the package

• Provide naming scope
  – prevents naming conflicts between classes and interfaces defined in different packages

• Permits naming hierarchy

  import ourProject.numericalCode.myPackage.*

  import ourProject.numericalCode.*

  import ourProject.*
Packages

Each project team member is responsible for a package.
Java-specific type: Vector

- Vector is a cross between a list (extensible) and an array (index). It’s defined in java.util
  - Elements are of type Object.
  - If you put something in a Vector and take it out later, the apparent type has widened to Object.
  - Vector grows by adding to high end:

    Vector v = new Vector(); // creates empty Vector

    if (v.size() == 0) // true

    v.add("abc"); // increases size by 1 and stores argument

- To access an element, a cast is necessary:
Java-specific type: Vector

String s = (String) v.get(0);

- Other operations on vectors:

  v.remove(0); // removes 1st element (shifts remainder)

  v.set(0,"abcd"); // changes existing element
Stream input/output

• Package java.io provides standard Input and Output (io).

• Input

```java
// read an integer

BufferedReader in =
    new BufferedReader (new InputStreamReader(System.in));

String s = in.readLine();

int i = Integer.parseInt(s);
```

• Output
Stream input/output

// write an integer

System.out.println(i);
Applications

• A java application starts with the main method of a specified class:

```
java myClass a1 a2 ...
```

• Class with a main method:

```
public class myClass {

    public static void main(String [] args) {

        // args[0] == a1

        // args[1] == a2

        // start of program
    }
}
```
Applications
Summary

- Values and objects
- Objects can be shared and mutable
- Java is strongly typed and type-safe
- Java provides automatic storage management
- All objects are subtypes of Object and understand toString() equals()
- Primitive types are converted to other types
- All types can be cast to other types (no computation)
- Packages provide encapsulation and naming scope
- java.util provides Vector
- java.io provides standard input/output
- Executions starts at main() method
Tool of the day: CVS

- CVS is the Concurrent Versions System, an open-source version control system.
  - A version control system allows multiple programmers to work on a project at the same time.
  - It tracks changes and builds a history of those changes.
  - It allows you to merge modification done on files.
  - Works with SSH, so you don’t need a dedicated server to use it. You can even use it on your CS account.
  - More information on CVS is available at:

    http://www.cvshome.org/

- Other version control system exist.
  - Visual SourceSafe, the Microsoft solution, offers tight locking controls.
Tool of the day: CVS

- Subversion, the replacement for CVS, is slowly gaining popularity.