Today …

• More on basic language constructs
  – primitive types, variables, operators, control flow

• Reading Assignments
  – Core: Ch. 3: 35 – 49, 51 – 52, 71 - 92
  – Style: Ch. 3 – 4

• Homework Assignment #1
  – Due: Thursday, January 28th (2 weeks)
Announcement

• We have a new room!
  – Paccar 210
  – Across from the Engineering office

• Next Tuesday we’ll meet in this room

• This is a lab classroom
  – Computers on each desk
  – Java not installed in the lab :( 
  – But, will try to get it installed w/ an editor

Challenge question

• What is the signature for a Java `main` method?

Answer: `public static void main string bracket bracket args`
(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

    public class MyFirstApp {
        public static void main(String[] args) {
            System.out.println("Java Rules!");
        }
    }

We will talk more about classes later.
For now you just need to know the basic syntax to start writing your own programs.

We are defining a class called "MyFirstApp"
There must be one of these in each file
(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}

MyFirstApp has a single method (in this case the special "main" method)

the return type of the method (void means no return value)
(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}

The formal arguments to the method (an array of String objects called ‘args’)

we will cover this later (like static member functions in C++)
(Beginning) Anatomy of a Class

- Every Java application has to have:
  - At least one class
  - At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

The Java System class (can do lots of useful stuff)

A (static) field of the System class
An object denoting the standard out (we’ll go over this later)
(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

A method call on the `out` object
(print a line to the terminal/console)

---

(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

The actual argument to the method
(the string to be printed in this case)
(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

Statements end in semicolons

(Beginning) Anatomy of a Class

• Every Java application has to have:
  – At least one class
  – At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

Braces define blocks of code
(Beginning) Anatomy of a Class

- Every Java application has to have:
  - At least one class
  - At least one main method (not per class, just in the app)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Java Rules!");
    }
}
```

Writing a class with a main

- In Java, everything goes in a class
  - Edit your source code file … with a .java extension
  - Compile it into a new class file … with a .class extension
  - When you run your program, you are running a class

- The JVM runs a program by loading the class and executing its main method
  - Programs usually have multiple classes (source files)
  - But the program starts from one class
Writing a class with a main

1. Save as:
   MyFirstApp.java

2. Compile:
   javac MyFirstApp.java

3. Run:
   java MyFirstApp

The Java on-line API documentation

The Java API docs are very useful when writing code!

- [http://java.sun.com/javase/6/docs/api/](http://java.sun.com/javase/6/docs/api/)
- or follow the link from the course webpage

- Contains all the classes in the standard libraries, their methods, and comments
- The documentation is built using javadoc
  - We’ll talk more about this program later
Java Data Types

• In Java, all variables must have a declared type

• Java distinguishes between “primitive” and “non-primitive” types
  – Primitive types are basic values (numbers, characters, etc.)
  – Array and reference types are the non-primitives
    • We’ll talk more about these later
Java Data Types

- This distinction means Java is not a “pure” object-oriented language (since it has non-object values)
  - Java does provide object “wrappers” for primitive types
  - Since Java 1.5, automatic “autoboxing” and “unboxing” operations are supported (more on this later)

- Like C++, Java is case sensitive
  - So, e.g., a variable “x” is different than “X”
  - Or “main” is not the same as “Main” or “MaiN”, etc.

Java Primitive Data Types

Integer types

- ranges of Java integer types are not dependent on the architecture

- int
  - Example:
    
    int i = 1999999; // declares var i, sets it to value
  
  - 4 bytes (-2,147,483,648 to 2,147,483,647)

- short
  - short s = 19999;
  
  - 2 bytes (-32,768 to 32,768)
Java Primitive Data Types

- **long**
  - long \( l \) = 5999999999999999999L; // need the L here!

- **byte**
  - byte \( b \) = 99;
  - 1 byte (-128 to 127)

Integer types example

```java
public class IntegerTest {
    public static void main(String[] args) {
        int i = 1999999;
        short s = 19999;
        long l = 5999999999999999999L;
        byte b = 99;
        System.out.println("i = " + i + "s = " + s + "l = " + l + "b = " + b);
    }
}
```

This prints:

```
i = 1999999
s = 19999
l = 5999999999999999999
b = 99
```

What does this print?
Java Primitive Data Types

*Floating point types*

– again, these *are not* dependent on the architecture

- **float**
  – float \( f = 3.14159F; \)
  – 4 bytes (about +/- 3.40282347E+38, 6-7 sig. digits)

- **double**
  – double \( d = 3.14159; \)
  – 8 bytes (much larger than float, 15 sig. digits)

Note if you don’t put “F” (or “f”), defaults to double!

Java Primitive Data Types

*Character and boolean types*

- **char**
  – char \( c = 'a'; \)  // this is not the same as the string “a”
  – Unicode (‘\uXYZ’ for XYZ the unicode number)

- **boolean**
  – boolean \( b = \text{true}; \)  // can also use false
  – You cannot *convert* between boolean values and integers (e.g., 0 is not false)
Classes for primitive types: “Boxing”

• It is sometimes convenient to use primitive values as objects
• Java provides classes for each primitive type

```java
Integer i = new Integer(10); // i refers to an object, not 10
Float f = new Float(4.4F); // why the “F”?
Character c = new Character(’a’);
System.out.print(“i = “ + i.intValue()); // get and print 10
```

– Integer, Float, Character are classes
– and i, f, and c are references to objects of these classes
– We will go into classes and objects later …

Java classes for primitive types

• Recall that
  – Java distinguishes between primitive & non-primitive types
  – Types are checked statically (at compile-time) in Java

• Assume we have two variables defined:
  ```java
  Integer obj = new Integer(5); // object type
  int prim = 10; // primitive type
  ```

• Should this expression compile?
  ```java
  prim = prim – obj; // int – Integer
  ```

This is an example of “unboxing”
Java takes the value out of the object to ensure type safety
Java classes for primitive types

• What about this expression?

```java
obj = prim + obj; // int + Integer
```

This is an example of "autoboxing". Java puts the value into an object to ensure type safety.

• Older versions of Java didn’t support either of these
  – Later we’ll see where this is really useful

• When possible though, you should use the primitive types, not the “boxed” versions of the primitive types

Variables

• Every variable is declared with a type

```java
double salary; // salary is of type double
int vacationDays; // vacationDays is of type int
boolean done; // done is of type boolean
```

• Variables **must be** initialized before they are used!

```java
int vacationDays;
if(vacationDays > 100) // compile error!!!
    System.out.print("You really need a vacation!");
```

VariableInitialization.java:4: variable vacationDays might not have been initialized if(vacationDays > 100)
           ^
1 error
Variable declaration vs. initialization

- This is a variable *declaration*
  ```java```
  int vacationDays;
  ```java```
- This is a variable *initialization* (also an assignment)
  ```java```
  vacationDays = 20;
  ```java```
- These can be *combined* in Java
  ```java```
  int vacationDays = 20;
  ```java```
- Declarations can be *anywhere* in your code
  ```java```
  double salary = 65000.00;
  System.out.println(salary); // print the value, add newline
  int vacationDays = 12;
```

Constant variables

- Java uses the keyword “*final*” to denote constants
  - A final variable cannot be changed after initialization

  ```java```
  final double CM_PER_INCH = 2.5;
  // do some stuff
  CM_PER_INCH = 2.54; // compile error!!!
  ```java```

```
FinalTest.java:4: cannot assign a value to final variable CM_PER_INCH
  CM_PER_INCH = 2.54;
  ^
  1 error
```
Class constants (fields)

- Many classes have constant fields that can be accessed within other classes and methods
- For example:
  
  ```java
  public class Integer {
    public static final int MAX_VALUE = 2147483647;
  }
  ```
- Within our programs, we can access this constant value:
  
  ```java
  public class MyClass {
    public static void main(String[] args) {
      System.out.print("Guess a number between 1 and " +
                      Integer.MAX_VALUE);
    }
  }
  ```

Converting between numeric types

- Lossless value conversions:
  - byte → short
  - short → int
  - int → long
  - int → double
- Allowed conversions that can lose precision:
  - int → float (e.g., 123456789 to 1.23456792*10^8)
  - long → float
  - long → double
Explicit casts

- You have to explicitly force a (potentially) lossy conversion
  - For example
    ```
    double pi = 3.14159
    int y = pi;  // compile error!
    ```
  - Instead, you have to do this
    ```
    double pi = 3.14159;
    int y = (int) pi;
    ```

One way to think about this is that the compiler automatically casts the allowed conversions ... but not the disallowed ones

Operators

- Java supports the normal arithmetic operators
  ```
  +  (plus)
  -  (minus)
  *  (multiply)
  /  (divide)
  %  (modulus)
  ```

- And the various standard shortcuts
  ```
  - x += 4;  // x = x + 4;
  - x++;     // x; x = x + 1
  - ++x;     // x = x + 1; x;
  ```
Operators

- Java also supports the standard boolean operators
  
  ```
  && (logical and)
  || (logical or)
  == (value equal)
  != (value not equal)
  <, >, <=, >=
  ```

  When chaining these together (a && b || c), you should use `parenthesis` to make evaluation order clear to the “reader”

- Plus the bitwise ops
  
  - & (and), | (or), ^ (xor), ~ (not), >>, << (bit shift)

Control Flow

- Standard conditionals: if-then-else and switch

- Standard iteration (loops): do-while, while-do, and for

- A for-each variant (differs from C++)
If-then-else

if (boolean condition) statement

• Example:
  if (yourSales >= target)
      bonus = 100;
  – You can also add braces:
      if (yourSales >= target) {
          bonus = 100;
      }

  What type must condition be? Can it be an int?

For multiple statements in a block, you must use braces
**If-then-else**

\[
\text{if (boolean condition) block1 else block2}
\]

- Example:

  ```
  if (yourSales >= target) {
    bonus = 100;
  }
  else {
    bonus = 0;
  }
  ```

*Do we need the braces here?*

---

**If-then-else**

\[
\text{if (boolean condition) block1 else if block2}
\]

- Example:

  ```
  if (yourSales >= target) {
    bonus = 100;
  }
  else if (yourSales >= (target/2)) {
    bonus = 50;
  }
  ```
**If-then-else**

if (boolean condition) block l else if block2

- Will this compile?
  
  if (yourSales >= target)
  bonus = 100;
  else if (yourSales >= (target/2))
  bonus = 50;

  Yes! These blocks have only one statement each

**If-then-else**

if (boolean condition) block l else if block2 else block2

- Example:
  
  if (yourSales >= target) {
    bonus = 100;
  } else if (yourSales >= (target/2)) {
    bonus = 50;
  } else {
    bonus = 0;
  }

  Any number of else if's are allowed

  How many else's can there be?
Switch Statements

- I personally try never to use these, but they have been known to come in handy …

```java
switch (choice) {
  case x:
    statement1; ...
    break;
  case y:
    statement1; ...
    break;
  default:
    statement1; ...
    break;
}
```

- `choice` is an integer (variable) or enumerated constant (we’ll talk about these next time)
- `x`, `y`, etc., are `int` or `enum` values
- `default` is like the `else` case
- `break` is not required
- It exits the switch statement
- Unboxing also works

Loops

**while (condition) block**

- Example:

```java
while(balance < goal) {
  balance += payment;
  double interest = balance * (interestRate / 100);
  balance += interest;
  years++;
}
System.out.println(years + “ years.”);
```

*What does this loop do? What is printed?*
Loops

do block while (condition)

• Example:
  do {
    balance += payment;
    double interest = balance * (interestRate / 100);
    balance += interest;
    years++;
    // print current balance
    // ask if ready to retire and get input
  } while(input == ‘n’);

What does this loop do?

Loops

for (initialize; condition; modify) block

• Examples:
  for(int i = 10; i > 0; i++) {
    System.out.println(i);  // i only visible in this block
  }
  – Alternatively
    int i;
    for(i = 0; i < 10; i++)
      System.out.println(i);
    System.out.println(i);

Will this compile? What is the value of i?
Loops – “Foreach”

for (variable : collection) block

• This is a “generalized” version of for loop (new in 1.5)
• Example:

```java
int[] theArray = {1, 2, 3, 4, 5}; // this is an array of ints
for(int i : theArray) // read: “for each i in theArray”
    System.out.println(i); // i only visible in this block
```

This is pretty slick!

We’ll also see this again later ...

Breaking out of a loop

• The break statement can be used within a block to break out of the nearest loop
• Example:

```java
while(years <= 100) {
    balance += payment;
    double interest = balance + (interestRate / 100);
    balance += interest;
    if(balance >= goal)
        break;
    years++;
}
```

What does this do here?
Continuing a loop

- The `continue` statement can be used within a block to continue the nearest loop

- Example:

```java
for(int x = 0; some condition; x++) {
    if(some other condition) {
        continue; // skip the useful stuff & keep looping
    }
    // do something useful
}
```