Types, Enums and Repetition

This lecture will

- Introduce the rest of the basic types
- Introduce **Enum**s
- Explain loops in Java
- Introduce the **`while`** and **`do`** statements for conditional loops and the **`for`** statement for counting loops
- Tell you how to escape from an infinite loop
- Show how loops can be nested.
- Explain how to select an appropriate loop construct for a given problem.

Types

- So far we have covered four basic types in Java
  - int
  - double
  - boolean
  - char
- There are four more, all of them numeric
  - byte
  - short
  - long
  - float

Numeric Types – the full set

Type	Contains	Minimum value	Maximum value
`byte`	signed integer	-128	127
`short`	signed integer	-32768	32767
`int`	signed integer	-2147483648	2147483647
`long`	signed integer	-9223372036854775808	9223372036854775807
`float`	floating point	-1.40239846E-45	+3.40282347E+38
`double`	floating point	-10^{308}	$+10^{308}$

- For practical purposes you can normally stick to **`int`** and **`double`**

Integer literal values

- If a number has no decimal point it is assumed to be an **`int`** unless Java has a reason to assume otherwise

```
int larger = 42;
short smaller = 42;
byte smallest = larger; // ERROR
byte otherSmallest = (byte)larger;
long largest = smallest;
```

- You can assign a smaller value to a larger but not the other way around unless you use a cast

Real literal values

- If a number has a decimal point it is a **double**
- You can also use exponential format for doubles

```
double normal = 42000000.0;
double exponential = 4.2E7; //4.2 * 10^7
```

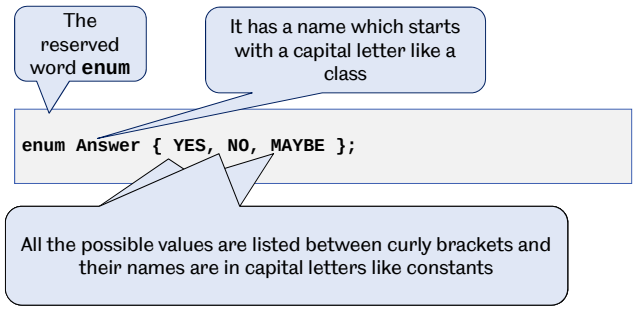
- In the unlikely event that you wanted to use a **float** literal you have to add an **f**
- These are all **float** values
  - **4.2f**
  - **42f**
  - **4.2E10f**

Enumerations

- The **boolean** type has two possible values
- The next smallest basic type is **byte** with 256 values
- Sometimes we want something in between
- For example the answer to "Will it rain tomorrow?" may not be a straight Yes/No. We also need Maybe
- It would be nice to be able to declare our own types with our own, limited, set of values

```
enum Answer { YES, NO, MAYBE };
```

The enum declaration



- It goes before
  public static void main (String [] args){

Reasons to use an enum

- There are a (relatively) small number of possible values
- You know what they are in advance
- For example

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                            FRIDAY, SATURDAY, SUNDAY };
```

```
enum Suit {  HEARTS, CLUBS, DIAMONDS, SPADES };
```

```
enum Role {  MANAGER, CLERICAL, TECHNICAL, CATERING };
```

Using an enum – Declarations

```
import sheffield.*;

public class EnumMaybe {

     enum Answer { YES, NO, MAYBE };

     public static void main (String [] args) {

          Answer rainTomorrow;

        ……..

     }
}
```

Declaring the **enum** type

Declaring a variable of the **enum** type

Using an enum – Assigning a value

```
Answer rainTomorrow;
EasyReader keyboard = new EasyReader();

if  ( keyboard.readBoolean(
        "Will it be dry tomorrow? ") )
   if  (keyboard.readBoolean(
        "Are you sure it won't rain tomorrow? "))
          rainTomorrow = Answer.NO;
   else
          rainTomorrow = Answer.MAYBE;
else
   if  ( keyboard.readBoolean(
                "So it will rain tomorrow? ") )
          rainTomorrow = Answer.YES;
   else
          rainTomorrow = Answer.MAYBE;
……
```

Using an enum – Using the values

```
import sheffield.*;
public class EnumMaybe {

   enum Answer { YES, NO, MAYBE };

   public static void main (String [] args) {
     EasyReader keyboard = new EasyReader();
     Answer rainTomorrow;
      ……

     if  (  rainTomorrow != Answer.NO  )
            System.out.println("Take an umbrella");

}
```

You can only use == and !=

Using an enum with a switch

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
        FRIDAY, SATURDAY, SUNDAY };
public static void main (String [] args) {
     Day today, nextWorkingDay;
     ...
     switch ( today ) {
        case MONDAY :
           nextWorkingDay = Day.TUESDAY; break;
        case TUESDAY :
           nextWorkingDay = Day.WEDNESDAY; break;
        case WEDNESDAY :
           nextWorkingDay = Day.THURSDAY; break;
        case THURSDAY :
           nextWorkingDay = Day.FRIDAY; break;
        default : nextWorkingDay = Day.MONDAY;
     }
     ...
```

You don't need the **Day.** before the enum value if it is after **case**

The rain problem in full

```
import sheffield.*;
public class EnumMaybe {
   enum Answer { YES, NO, MAYBE };
   public static void main (String [] args) {
      EasyReader keyboard = new EasyReader();
      Answer rainTomorrow;
       if  ( keyboard.readBoolean("Will it be dry tomorrow? (Y/N) ") )
           if  (  keyboard.readBoolean(
                        "Are you sure it won't rain tomorrow? (Y/N) "))
               rainTomorrow = Answer.NO;
           else
               rainTomorrow = Answer.MAYBE;
       else
           if  (keyboard.readBoolean("So it will rain tomorrow?(Y/N) "))
               rainTomorrow = Answer.YES;
           else
               rainTomorrow = Answer.MAYBE;
      switch (rainTomorrow) {
         case MAYBE :
         case YES :
            System.out.println("Take an umbrella");   break;
         case NO :
            System.out.println("Take sun screen");
      }
   }
}
```

enums fit naturally with **switch** statements

Repetition

- We often need to repeat a statement or sequence of statements (called the **loop body**) a number of times
- Java provides two kinds of control structure for repetition; **counting** loops and **conditional** loops
- Counting loops repeat a set number of times; we know how many times before we start the loop:

  do the following five times
  add a spoonful of sugar to the cake mixture
  mix well

- In a conditional loop, we do not know how many times to repeat before we start the loop; we decide when to stop by using a test

Two kinds of conditional loop

- The test in a conditional loop may appear at the end:

  do
  add a spoonful of sugar to the cake mixture
  mix well
  while the mixture does not taste sweet enough

- The test may also appear at the start of a loop:

  while the mixture does not taste sweet enough
  add a spoonful of sugar to the cake mixture
  mix well

- In a 'do' loop we always do the loop body once, whereas it may not be executed at all in the 'while' loop

The do statement

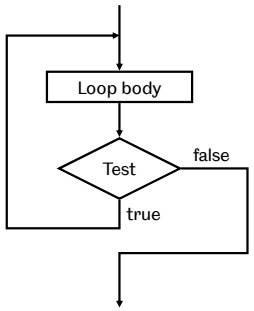
- The **do** statement is a conditional loop, with the test at the **end** of the loop.

```
do
        Loop_body
while (  Boolean_expression  );
```

  The loop body is always executed at least once. Repetition continues until the expression evaluates to **false**

- The **do** loop is often useful when handling user responses to a question

Flow diagram of the do loop



Example – asking for a multiple of 7

```
EasyReader keyboard = new EasyReader();
int number;

do {
   number = keyboard.readInt(
          "Please type in a multiple of 7: ");
} while  ( number % 7 != 0 );

System.out.println("Thank you");
```

Tests to see if it is divisible by 7 because if it is the remainder, number % 7, will be 0

```
Please type in a multiple of 7: 4
Please type in a multiple of 7: 27
Please type in a multiple of 7: 938
Thank you
```

Example – squaring a list of numbers

```
//   Computes the square of a sequence of numbers

EasyReader keyboard = new EasyReader();

do {
     int number = keyboard.readInt("Enter a number: ");
     System.out.println("The square of your number is "
                + number*number);
} while (  keyboard.readBoolean("Another try? ")  );

System.out.println("Finished");
```

```
Enter a number: 5
The square of your number is 25
Another try? y
Enter a number: 98
The square of your number is 9604
Another try? n
Finished
```

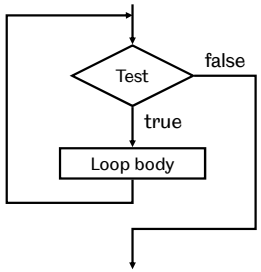
The `while` statement

- The **`while`** statement is a **conditional loop**, with the test at the **start** of the loop

```
while (  Boolean_expression )
       Loop_body;
```

  First, we evaluate the Boolean expression. If it evaluates to **`true`**, we execute the loop body
- After executing the loop body, the Boolean expression is re-evaluated. The loop terminates when it evaluates to **`false`**
- In this loop, unlike the **do** loop, the loop body may never be executed

Flow diagram of the `while` loop



Note that if the test never evaluates to **`false`**, then we have an **infinite loop**

Testing to see if a number is prime

- A number is prime if it is only divisible by 1 or itself
- We are going to test to see if a number is prime by first trying to see if it is divisible by 2. If it is the number is not prime
- If it is not divisible by 2 we try 3 and so on until we have tested all the possible factors up to half the number. If we still haven't found one then the number is prime

Example – looking for primes

Contains a lot of unnecessary brackets

```
int possiblyPrime = keyboard.readInt(
            "Please type in an integer: ");
int testFactor = 2;
while  (  ((possiblyPrime % testFactor) != 0) &&
             (testFactor < (possiblyPrime / 2))  )  {
   testFactor++;
}
if  (  possiblyPrime % testFactor == 0  )
   System.out.println(possiblyPrime+" is not prime "+
            "because it is divisible by "+testFactor);
else
   System.out.println(possiblyPrime+" is prime");
```

```
Please type in an integer: 1234567
1234567 is not prime because it is divisible by 127
```

Infinite loops

- These loops never stop

```
int x = 10;
while (  x == 10  )
   System.out.println("Help!  Get me out of here!");
```

```
char answer = 'y';
do
   System.out.println("It's boring here isn't it?");
while (answer == 'y');
```

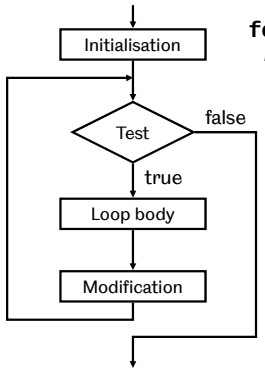
Escaping from infinite loops

- Ideally never get into one
- If the program is simply doing nothing it may be waiting for input and you have forgotten the prompt
- But there is always Ctrl+C

The `for` statement

- The **`for`** statement is a **counting** loop
- The loop uses a **control variable** (counter) to keep track of how many times we have been around the loop
- First the control variable is **Initialised** – the counter is set to its start value
- Then the control variable is **Tested** – the counter is checked to see if we have been around the loop enough times
- If it passes the test the loop body is obeyed
- Then the control variable is **Modified**, the counter is increased or decreased, and tested again

Flow diagram and syntax of the `for` loop



`for` (*Initialisation*; *Test*; *Modification*)
Loop_body;

Examples of the `for` statement

- The following loop displays a message 5 times:

```
int i;
for (i=1; i<=5; i++)
    System.out.println("Hello");
```

- The **`++`** operator increments the value of a variable
- We can place the declaration of **`i`** inside the **`for`** statement:

```
for (int i=1; i<=5; i++)
    System.out.println("Hello");
```

- Now **`i`** can only be accessed within the loop body – this is usually the preferred form

Counting in Java

- In Java we start counting at 0, rather than 1. So we could write an equivalent loop as:

```
for (int i=0; i<5; i++)
    System.out.println("Hello");
```

- Usually we make use of the control variable inside the loop, e.g. to display the integers between 0 and 9:

```
for (int i=0; i<10; i++)
    System.out.println(i);
```

- The loop body can also be a statement block. We use indentation to make it clear which statements are inside the loop.

Multiplication tables

```
import sheffield.*;
public class TimesTable {
   public static void main(String args[]) {

      final int MAX_TABLE = 12; //number of table rows

      EasyReader keyboard = new EasyReader();
      int number = keyboard.readInt("Enter a number: ");

      for (int i=0; i<=MAX_TABLE; i++) {
         System.out.print(i+" times "+number+" = ");
         System.out.println(number*i);
      }

   }
}
```

Output of the `TimesTable` program

```
Enter a number: 9
0 times 9 = 0
1 times 9 = 9
2 times 9 = 18
3 times 9 = 27
4 times 9 = 36
5 times 9 = 45
6 times 9 = 54
7 times 9 = 63
8 times 9 = 72
9 times 9 = 81
10 times 9 = 90
11 times 9 = 99
12 times 9 = 108
```

More about the loop control variable

- Usually we increment (add one to) the control variable on each loop. However, we can also count in larger steps using modifications of the form;

  `i+=10`

  which is an abbreviation of the statement

  `i=i+10`

- For example, the following loop counts from 0 to 100 in steps of 10 (0, 10, 20 ... 90, 100):

```
for (int i=0; i<=100; i+=10)
   System.out.println(i);
```

Counting down

- We can count down using the decrement operator; this loop counts down from 10 to 0 (10, 9, 8 ... 1, 0):

```
for (int i=10; i>=0; i--)
  System.out.println(i);
```

- We can count down in larger steps too; this counts from 100 down to 0 in steps of 5 (100, 95, 90 ... 10, 5, 0):

```
for (int i=100; i>=0; i-=5)
  System.out.println(i);
```

Nested loops

- The body of a loop can contain another loop. A loop within a loop is called a **nested loop**.
- We can form a grid of multiplication tables with the following nested loop:

```
EasyWriter screen = new EasyWriter();
for (int i=1; i<=5; i++) {
    for (int j=1; j<=10; j++) {
          screen.print(i*j,3);
          screen.print(" ");
    }
    screen.println();
}
```

- Again, note the use of **indentation** in this example.

Output of the multiplication program

- The output is as follows; note that **`i`** counts the rows (this is the outer loop), and **`j`** counts the columns in each row (this is the inner loop):

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
```

- We use **`println`** at the end of each inner loop to move to a new row.

Nested loops

```
EasyWriter screen = new EasyWriter();
for (int i=1; i<=5; i++) {
    for (int j=1; j<=10; j++) {
          screen.print(i*j,3);
          screen.print(" ");
    }
    screen.println();
}
```

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
```

Comparison of `for`, `while` and `do`

- The following are equivalent:

```
for (int i=startValue; i<=stopValue; i++) {
  statements
}
```

```
int i=startValue;
while (i<=stopValue) {
  statements
  i++;
}
```

```
int i=startValue;
if (startValue<=stopValue)
  do {
    statements
    i++;
  } while (i<=stopValue);
```

Guidelines for using `for`, `while` and `do`

- If we know the number of repetitions before entering the loop, use `for`
- The `for` statement is much clearer for simple counting loops it is also efficient
- If we do not know the number of repetitions before entering the loop, `while` is usually appropriate
- The `do` statement is indicated if we know that the loop body must be executed at least once

Summary of key points

- There are four more numeric types; **`byte`**, **`short`**, **`long`** and **`float`**
- Enums are like a type you declare yourself and are used when there are a fixed number of possible values
- Java provides looping statements for repeating things
- The **`while`** and **`do`** statements are conditional loops used when we don't know in advance how many times we will go around the loop. A **`while`** statement tests its condition at the start and a **`do`** at the end of the loop body
- The **`for`** statement is used when we know in advance how many times we will go around the loop and has a control variable that counts iterations and can be used inside the loop
- You can stop a program in an infinite loop by pressing Ctrl+C