Numbers

covering

** **`Math`**, **`Random`**, **`Number`**: Class and Methods

** Variables, Methods and Classes: **`static`** and **`final`**

** Wrapper Classes and Autoboxing (*from Java 5.0*)

** Recursion

Chapters 3-5 (sections 3.5, 4.4, 4.6, 5.1, 5.4) – "Big Java" book
Chapter 10 + Appendix B – "Head First Java" book
Chapters 2, 5, 8, 10-11, 20 (sections 2.7, 5.10, 8.6, 8.7 10.13, 11.14) – "Introduction to Java Programming" book
Chapter 9 – "Java in a Nutshell" book

`final`: Variables, Methods and Classes

- Sometimes we don't want *child classes* to override the implementation in the *parent*!
  - This means we want to restrict inheritance!
- Java keyword **`final`**.
  - Using this keyword will prevent child classes (or anyone else) modifying the variable/method this applies to.

```
public final int topSpeed = 100;
public final void stop() {...}
```

No changes can be made to value of **`topSpeed`**.
No children can override method **`stop()`**!

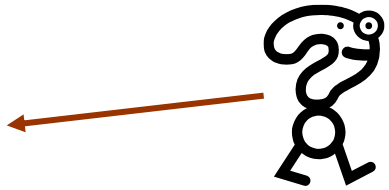
  - We can prevent a class from being inherited at all!

```
public final class Square {...}
```

No further specialisations are allowed!
Any attempt to extend **`Square`** will cause an error!

Class Variables & Methods

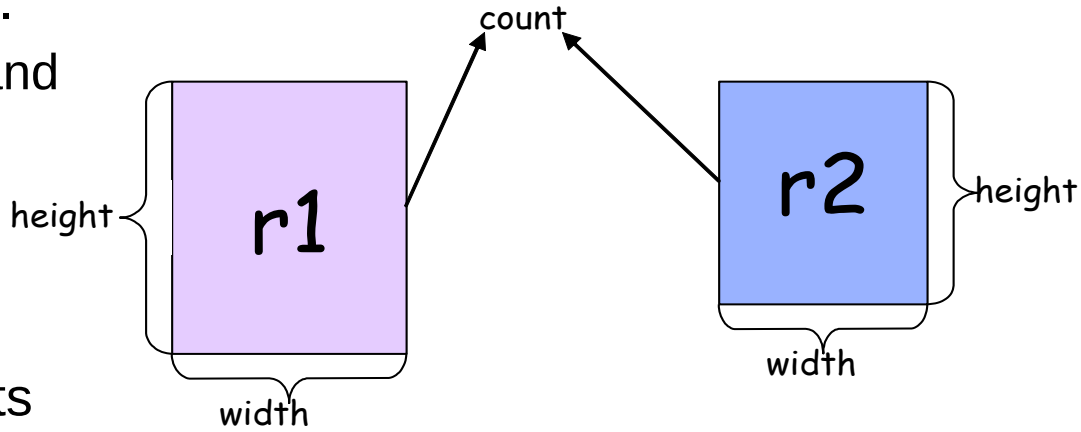
- Suppose we want to keep track of how many instances of a class are made.
  - Question: How can we *automatically* keep track?
    - cannot leave it to the objects;
    - cannot use an instance variable.
  - Answer: Use a *class variable.*



- A class variable is created when the *class* is created, rather than when an *object* is created.
  - There may be *n* instances of the class, but there will only be one instance of the class variable.
  - To declare a class variable, use the modifier **`static`**.

Using `static`

- Normally every instance of an object has its own copy of all variables and methods defined in the class.
  - Example: Rectangles **`r1`** and **`r2`** have a copy of **`width`**, **`height`**, **`area()`**, **`draw()`**, etc.



- If you declare something as *static*, it means that all objects have the **same** copy of that variable/method.
- Static methods can *only* reference static variables!
- Use *static* when a single copy of the data will suffice.
  - Example: Count the number of instances of rectangles.
- Static is *like global variables* but applies classwide.
- Static methods *become available when the class is loaded* (created), not when you make an instance of it.

Example (1/2): Using `static`

```
public class Rectangle extends Object {
  private static int count = 0;
  private int width;
  private int height;
  public Rectangle(int width,int height) {
    this.width = width;
    this.height = height;
    count++;
  }
  public static int getCount() {
    return count;
  } // end getCount() method
```

(cont.)

```
    /* we don't actually want to provide setCount() */
    public int getWidth() {...}
    public int getHeight() {...}
    public void setWidth(int width) {...}
    public void setHeight(int height) {...}
    public void draw() {...}
    public int area() {...}
} // end class Rectangle
```

Example (2/2): Static access

- Can access *static methods* even when no instance of a class has been created!

```
System.out.println("There are " +
                   Rectangle.getCount() +
                   " rectangles.");
Rectangle r1;
r1 = new Rectangle(10,20);
System.out.println("There are " +
                   Rectangle.getCount() +
                   " rectangles.");
```

What is the output?

Constants in Java – Almost!

- We can use the **`static`** modifier to declare "constants". Here we declare a class to be a "wrapper" for universal constants …

```
public class Math extends Object {
  public static float E = 2.718281f;
  public static float PI = 3.141592f;
  // ...
} // end class Math
```

- To access these values e.g.,

```
circleArea = Math.PI * (radius * radius);
```

This is messy, because the statement below is legal:

```
Math.PI = 3.0f;
```

Using `final` & `static` with Variables

- We can use the **`final`** modifier to ensure that nobody changes our static variables.
  - **Example**:

```
public class Math extends Object {
  public static final float E = 2.718281f;
  public static final float PI = 3.141592f;
  // ...
} // end class Math
```

Any attempt to assign a value to a **`final`** variable will cause an error!

`Math` Class and Methods

- Methods of class **`Math`**: almost like global methods!
  - Act on the argument but are not affected by an instance variable state.
  - **Example**: **`int x = Math.min(56,12);`** always does the same thing and doesn't use instance variables. → Method's behaviour doesn't need to know about a given object.

- **`Math` Class**:
  - Doesn't have instance variables.
  - Can't make an instance of class.

So is the **`Math`** class an abstract or an interface?

Static (*aka* Class) Methods

- We can also have *static* (class) *methods*, as well as variables.
- This is how **`java.lang.Math`** is declared:

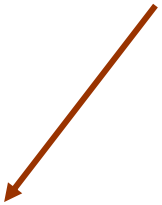
```
public class Math extends Object {
  public static int max(int a, int b) {
    return ((a > b) ? a:b); // if-then-else
  }
  public static int min(int a, in b) {
    return ((a < b) ? a:b);
  }
  public static double sin(double a) {...}
  public static double cos(double a) {...}
  public static double tan(double a) {...}
  public static double log(double a) {...}
} // end class Math
```

It would make no sense to implement these functions as instance methods. This isn't very OO! So use with caution.

Practice Exercise 1

```
public class Rectangle extends Object {
  private int width;
  private int height;
  public static int getWidth() {
    return this.width;
  }
  public static int getHeight() {
    return this.height;
  }
  public static void setWidth(int newWidth) {
    this.width = newWidth;
  }
  public static void setHeight(int newHeight){
    this.height = newHeight;
  }
  // rest of Rectangle code ...
} // end class Rectangle
```

What is wrong with this code?



Methods in `Math` Class: `static`

- **Classes** that can't be instantiated:
  - **Abstract classes** and **interfaces**.
  - Classes with **private constructors**.
    - Only code inside the class can invoke a private constructor!
- Methods in class **`Math`** are static.
  - Invoking a **`Math`** method:

```
Dog myDog = new Dog();
myDog.catchBall();
```

calling a non-static method: use a reference variable name

```
Math.max(10,20);
```

calling a static method: use the class name

`static` and `final` Variables

- What we already **know**:
  - **All instances** of the same **class** share one copy of a static variable.
  - **Initialisation of static variables** happens before any object of the class is created.
  - A **non-initialised static variable** will have the default value of that variable type.
  - Variables that are **`static`** and **`final`** cannot be changed.
    - Convention is to name **`static final`** variables in all caps.
- **Initialisation** of **`static final`** variables:
  - When the variable is **declared**.

OR

  - In a **static initialiser**: block of code that runs when a class is loaded, before any code can use the class.

`static` and **`instance`** variables: not the same!

Example: Using a Static Initialiser

```
public class Bar {
  public static final double BAR_SIGN;

  static {
    BAR_SIGN = (double) Math.random();
  }
}
```

naming convention for **`static final`** variables: all caps

static block to initialise **`static final`** variables

`final` Variables, Methods and Classes – RECAP

- **`final`** variable: cannot change its value.

```
class Foof {
  final double weight = 15.6;
  final int whuffie;
  Foof() {
    whuffie = 3;
  }
}
```

`final` variables must either be initialised when declared or in the constructor

- **`final`** method: cannot override the method.

```
class Poof {
  final void calcWhuffie() {
    // important things that must not be overridden
  }
}
```

- **`final`** class: cannot extend the class.

```
final class MyFinalClass {
  // class cannot be extended
}
```

`final` classes and **`abstract`** classes: not the same!

Practice Exercise 2

- Which statement(s) would cause a compilation error if inserted where indicated?

```
public class ParameterUse {
  public static void main(String[] args) {
    int x = 0;
    final int y = 1;
    int[] z = {2};
    final int[] w = {3};
    useArgs(x, y, z, w);
  }
  static void useArgs(final int a, int b,
                      final int[] c, int[] d) {
    // INSERT CODE HERE
  }
}
```

1. `a++;`
2. `b++;`
3. `b = a;`
4. `c[0]++;`
5. `d[0]++;`
6. `c = d;`

Practice Exercise 3

- What is wrong with the code below?

```
public class Foo {
  Circle c = new Circle();
  public void method1() { method2(); }
  public static void method2() {
    System.out.println("What is radius " + c.getRadius());
  }
  public static void main(String[] args) { method1(); }
}
```

- Is it possible to:
  - invoke an instance method or reference an instance variable from a static method?
  - invoke a static method or reference a static variable from an instance method?

Math Methods

- **Math.random()**: returns a **double** in range **0.0-1.0** (excluding).

```
double rnd =
       Math.random()*5.0;
// returns 0.0 ≤ rnd < 5.0
```

- **Math.abs(double num)**: returns a **double** that is the absolute value of **num**; method is overloaded to take/return an **int** value.

```
double absNum =
       Math.abs(-123.45);
// returns absNum = 123.45
```

- **Math.round(float value)**: returns a value rounded to the nearest **int** value; method is overloaded to take a **double** value and return a **long** value.

```
int roundedValue =
    Math.round(-10.8f);
// returns roundedValue = -11
```

- **Math.min(int a, int b)**: returns an **int** that is the minimum value between **a** and **b**; method is overloaded to take/return a **long**, **float** or **double** value.

```
double minValue =
       Math.min(123.45,
                123.46);
// returns minValue = 123.45
```

- **Math.max(int a, int b)**: returns an **int** that is the maximum value between **a** and **b**; method is overloaded to take/return a **long**, **float** or **double** value.

```
double maxValue =
       Math.max(123.45,
                123.46);
// returns maxValue = 123.46
```

There are more **Math** methods!

Random Class

- The **Random** class is part of the **java.util** package and provides methods that generate random numbers.
- Example:

```
import java.util.Random;

public class RandTest {
  public static void main(String[] args) {
    Random r = new Random();
    float aRandomFloat = r.nextFloat();
    int aRandomInt = r.nextInt();
    System.out.println("A random float is " + aRandomFloat);
    System.out.println("A random int is   " + aRandomInt);
  }
}
```

… and things for you to try out!

Wrapper Classes

- **Wrapping Classes**: used when a variable of a primitive type needs to be treated as an object.
  - Every primitive type has a wrapper class.
  - Wrapper classes are part of **`java.lang`** package → no need to import them:

Boolean
Character
Byte
Short
Integer
Long
Float
Double

different name from their primitive type

- **Wrapping** *versus* **Unwrapping**:

```
int i = 10;
Integer iWrapped = new Integer(i);
int unWrapped = iWrapped.intValue();
```

All wrapper classes have a similar method, e.g. **`charValue()`**, **`booleanValue()`**

Autoboxing (*from Java 5.0*)

- **Autoboxing**: automatic wrapping → conversion from primitive type to wrapper object is automatic!
- **Before Java 5.0**: Variables of primitive types and object reference variables were never treated interchangeably!
- **Wrapping** and **unwrapping** an **`int`** in **`ArrayList`** of primitive integers: before and after Java 5.0

```
public void doNumsNewWay() {
  ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();
  listOfNumbers.add(3);
  int one = listOfNumbers.get(0);
}
```

```
public void doNumsOldWay() {
  ArrayList listOfNumbers = new ArrayList();
  listOfNumbers.add(new Integer(3));
  Integer one = (Integer) listOfNumbers.get(0);
  int oneUnwrapped = one.intValue();
}
```

you can't declare an **`ArrayList<int>`**

Using Autoboxing: Examples (1/2)

- Method Arguments: you can pass either a reference or a matching primitive to a method that takes in a wrapper type; reverse is also true!

```
void takeNumber(Integer i) { }
```

```
int x = 1;
takeNumber(x);
```

- Return Values: you can return either a reference or a matching primitive on a method with a primitive return type; reverse is also true!

```
int giveNumber() {
  return x;
}
```

```
Integer x = 1;
int x = 1;
```

- Boolean Expressions: where a boolean value is expected, you can use either an expression evaluating to a boolean, a primitive or a matching wrapper.

```
if (boolean) {
  System.out.println("true");
}
```

```
(10 > 9)
boolean x = true;
Boolean x = true;
```

Using Autoboxing: Examples (2/2)

- Operations on Numbers: in operations where a primitive type is expected, you can use a wrapper type!

```
Integer i = new Integer(10);
i++;
System.out.println("New value is: " + i);
```

Output is ...
11

- Assignments: a variable declared as a wrapper (or primitive) can be assigned a matching wrapper (or primitive).

```
Integer d = x;
```

```
int x = 10;
```

```
Integer x = 10;
```

… and things for you to try out!

Static Methods in Wrappers

- **Wrappers** have *parse* methods: they take a **`String`** and return a primitive value.
  - **Example**: Converting a **`String`** to a primitive value.

```
String str1 = "10";
String str2 = "123.45";
String str3 = "true";
int i = Integer.parseInt(str1);                  // i=10
double d = Double.parseDouble(str2);             // d=123.45
boolean b = new Boolean(str3).booleanValue();    // b=true
```

different naming

```
String anotherStr = "ten";
int anotherInt = Integer.parseInt(anotherStr);
```

Compiles but will not run; things that can't be *parsed* cause a **`NumberFormatException`**.

Static Imports (*from Java 5.0*)

- What are they?
  - When using a static method or variable, you can *import it* and save on typing.
  - **Example**:

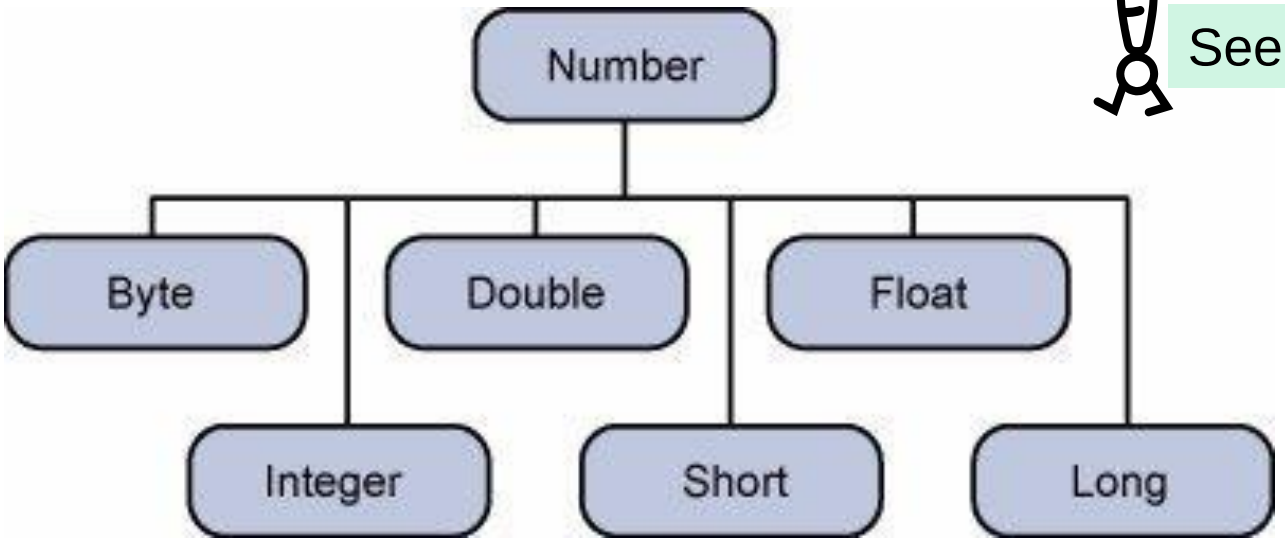
```
import java.lang.Math;
class BeforeStaticImports {
  public static void main(String [] args) {
    System.out.println("square root is " + Math.sqrt(4.0));
  }
}
```

```
import static java.lang.System.out;
import static java.lang.Math.*;
class WithStaticImports {
  public static void main(String [] args) {
    out.println("square root is " + sqrt(4.0));
  }
}
```

Using static imports can make code difficult to read.

Wrapper Classes and `Number` (1/2)

- All wrapper classes are subclasses of the **`Number`** abstract class (part of the **`java.lang`** package).



See the Java API.

  - For example, we can construct a **`Number`** object of type **`Integer`**:

```
Number num = new Integer(10);
```

Wrapper Classes and `Number` (2/2)

- Using a **`Number`** object instead of a primitive variable:
  - When the method's argument expects an object, e.g.

```
void takeNumber(Integer i) { }
```

    - Often used when manipulating collections of numbers.
  - Subclasses of **`Number`** provide constants to, e.g. represent the upper and lower bounds of the corresponding data type (**`MIN_VALUE`** and **`MAX_VALUE`**, respectively).
    - Example: **`System.out.println(Long.MIN_VALUE);`** → -2^{63}
  - Can use class methods to convert values to/from other primitive types, to convert to/from strings, and to convert between number systems (e.g. decimal, binary, hexadecimal).
    - Example:

```
public class IntegerDemo {
  public static void main(String[] args) {
    int i = 181;
    System.out.println("Number    = " + i);
    System.out.println("Hex value = " +
                       Integer.toHexString(i));
  }
}
```

Practice Exercise 4

- What is the output of the program below?

```
public class Round {
  public static void main(String[] args) {
    System.out.println(Math.round(-0.5) + " " +
                       Math.round(0.5));
  }
}
```

- Which of these classes define immutable objects?

  Character

  Byte

  Short

  Object

… and things for you to try out!

Recursion

- In general, methods can call other methods! (*)
- **Methods that call themselves**, directly or indirectly, are known as recursive.
- A **recursive method only knows how to solve the simplest case(s)** of a problem. This is known as the base case(s) (*aka* stopping condition).
- The **definition of many mathematical functions** is done through the use of recursion!
- **Classic problems** best solved using recursion:
  - calculation of factorial *n!* and of Fibonacci numbers;
  - resolution of Tower of Hanoi problem.

(*) But there are some exceptions, e.g. static methods cannot invoke non-static methods!

Algorithm for Recursion

- What happens when a **recursive method is called**?
  - If the method is called on a base case, then it returns the base case, i.e. the simplest solution.
  - If the method is called on a more complex case, then it divides the problem into two parts:
    1. a piece that it knows how to do;
    2. a piece that it doesn't know how to do, but that is solved by calling another method (called a recursive call).
       - *The clever bit is that this 2nd method is the same as the 1st method, but the problem is slightly simpler.*

Need to be aware of possible infinite recursion.

Example using Recursion

- **Factorial *n!*** of a non–negative integer ***n***:
  - Product n! = n(n − 1)(n − 2) . . . 1, n>1
    with 1! = 1 and by definition, 0! = 1.
  - So, 5! = 5*4*3*2*1 = 5*(4*3*2*1) = 5*4!
- A **recursive definition of factorial** is:

  n! = n(n − 1)! , n>1 (with 1! = 1 and 0! = 1)

Java implementation of **factorial n!**

```
public class CalculateFactorial {
  public long factorial(long number) {
    if (number <= 1)
      return 1;
    else
      return number*factorial(number - 1);
  }
}
```

base case

recursive call

Recursion *versus* Iteration

- **Recursion**:
  - another form of program control that uses repetition without a loop control;
  - sometimes allows for a problem's solution to be specified in a clear and simple manner;
  - **but** brings additional overhead to programs:
    - Everytime a program calls a recursive method, space needs to be assigned for the method's local variables and parameters. ➔ Extra memory required + time to manage the extra space.

- **Iteration**:
  - Can be used to solve a recursive problem in a non-recursive manner.
  - Is usually more efficient than recursion.

How can we solve the *n!* problem using iteration?

- **Recursion *or* Iteration**: When?
  - Depends on the nature of the problem!
  - Rule of thumb is to:
    - use an iterative approach if that is the obvious solution;
    - avoid using recursion if there are concerns about the program's performance.

Commercial software does not usually use recursion.

Practice Exercise 5

- For the method below:
  - What is **`mystery(1,7)`**?
  - Will the method terminate for every pair of integers **`a`** and **`b`** between **`0`** and **`100`**? Describe what the method returns, given integers **`a`** and **`b`** between **`0`** and **`100`**.

```
public int mystery(int a, int b) {
  if (0 == b)
    return 0;
  else return a + mystery(a, b-1);
}
```

… and things for you to try out!