Parallel Programming

A bit of JVM and Java recap connected to concurrency/parallelism

Today

- Sneak peek into JVMs
  - Brief overview of JVM components …
  - … spending a bit more time on bytecode
  - This material is **not examinable.** Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
  - Certain constructs and patterns that are connected to concurrency
  - More recap slides in the slide deck, not covered in class

Why Java?

Widely used programming language in academia and industry

Lots of courses downstream use Java

Lots of resources available online and in books

Sophisticated support of concurrency in the language and in libraries

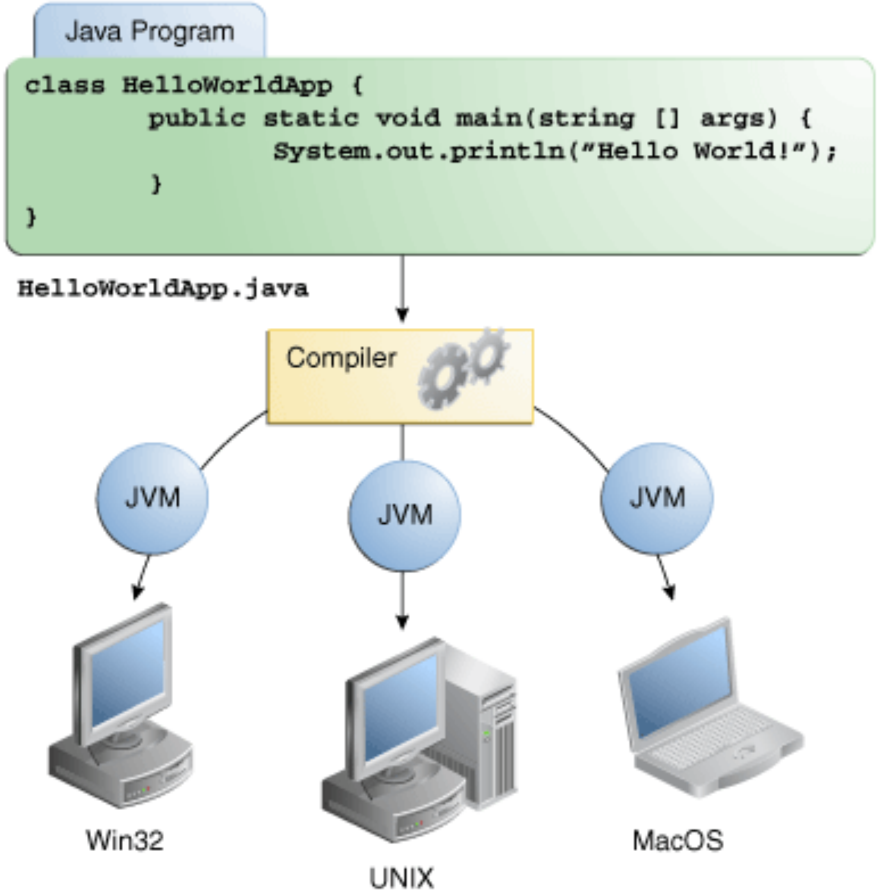
Java

Platform independence via bytecode interpretation

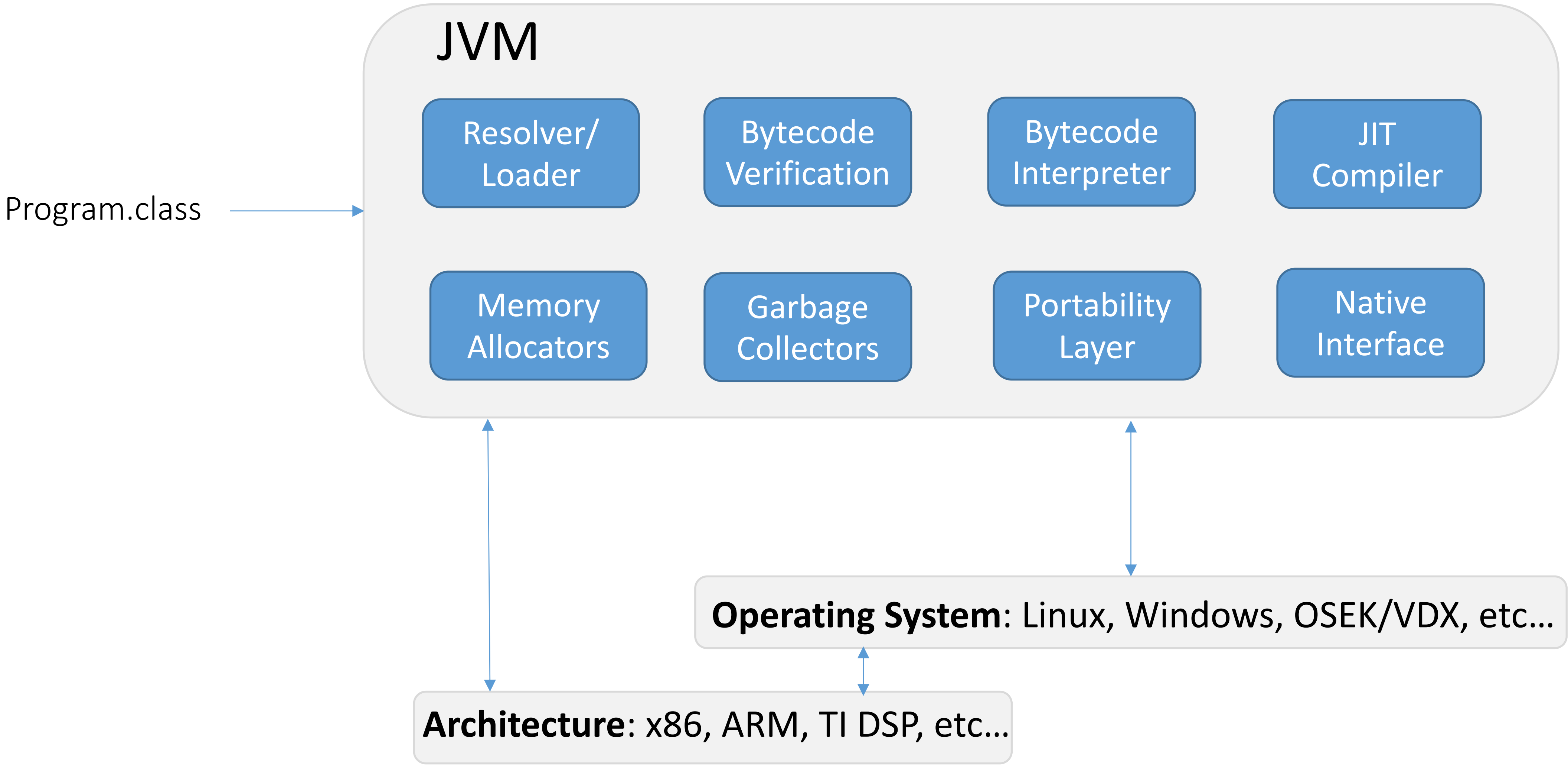
Java programs run (in theory) on any computing device (PC, mobile phones, Toaster, Windows, Linux, Android)

Java compiler translates source to byte code

Java virtual machine (JVM) interprets the bytecode of the compiled program



Key JVM Components



Resolver, Loader

Loads class files and setups their internal memory….but **when**?

```
class Test {

   public static void main(String args[]) {
      if (args[0].equals("nuf si HTE"))
         LoadMe  t = new LoadMe();
      }
}

class LoadMe()  {
   static int x = 0;
   static { System.out.println ("Got statically initialized"); }
}
```

The JVM has a choice here:

Eager: the JVM resolves the reference to LoadMe when class Test is first loaded.

Lazy: the JVM resolves the reference to LoadMe when it is actually needed (here, when the LoadMe object is created). Most JVMs are lazy.

Static initialization of the class is quite **non-trivial**, can be done **concurrently by many Java threads** (we will see what a Thread is later). Typically done before the class is used.

Bytecode Verification

Automatically verifies bytecode provided to JVM satisfies certain security constraints. Usually done right after the class is located and loaded, but before static initialization

- bytecodes type check
- no illegal casts
- no conversion from integers to pointers
- no calling of directly private methods of another class
- no jumping into the middle of a method
- ....and others

Minor problem:

Automated verification is undecidable.

Practically, this means the verifier may reject valid programs that actually do satisfy the constraints. ☹

The goal is to design a verifier that accepts as many valid programs as possible. ☺

Bytecode Interpreter

A program inside the JVM that interprets the bytecodes in the class files generated by the javac compiler using a **stack** and **local variable** storage.

- JVM is a **stack based abstract machine**: bytecodes pop and push values on the stack
- A set of registers, typically used for local variables and parameters: accessed by load and store instructions
- For each method, the number of stack slots and registers is specified in the class file
- Most JVM bytecodes **are typed**.

The bytecode interpreter is typically slow as its pushing and popping values from a stack…

One can speed-up the interpreter but in practice parts of code that are frequently executed simply get compiled by the Just in Time (JIT) compiler to native code (e.g., Intel x86, etc)…next.

Just-In-Time Compiler (JIT)

Compiles the bytecode to **machine code** (e.g., ARM, DSP, x86, etc) **on-demand**, especially when a method is frequently executed (hot method). JIT makes bytecodes fast ☺

```
class Test {

  public static int inc(int j) {return j + 1; }

  public static void main(String args[]) {
     int j = 0;
     for (int i = 0; i < 100000; i++)
        j  = inc(j);
 }}
```

Compilation of bytecode to machine code happens **during program execution**. Typically needs **profiling data** to know which method is hot or cold. Can be expensive to gather during execution.

In this example, method inc(int) is a hot method so may be **inlined** inside the main to avoid the overheads of function calling.

A modern JIT compiler has 100's of optimizations…

Memory Allocators

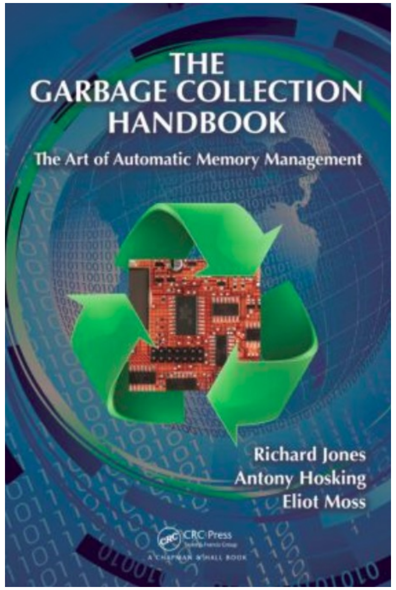
Consists of, often concurrent algorithms, which are invoked when your Java program allocates memory.

```
class Test {

  public static void main(String args[]) {

    A a = new A();

    int t[][] = new int[2][5];
}}
```

Object allocation in Java invokes the JVM memory allocator. The JVM memory allocator often has to ask the underlying OS for memory which it then **manages internally**.

The allocator algorithms typically **have to be concurrent** because multiple Java threads (we will learn about threads later) can allocate memory. Otherwise, if sequential, one may see major pause times in their application.

Garbage Collectors (GC)



JVM uses many different GC algorithms, often concurrent and parallel, invoked periodically to collect memory unreachable by your program.

```
class Test {

  public static void main(String args[]) {
    A a = new A();
    a = null;  // memory of A() now unreachable.
    System.gc();  // force GC
    while(true);   // wait till it triggers ☺
}}

class A() {
   public void finalize() {
     System.out.println("I got freed by the GC");
}}
```

Frees the programmer from having to free memory manually…which is good as it avoids tricky bugs.

Many different GC algorithms: generational, concurrent, parallel, mark and sweep, etc. Trade-off different performance goals. Concurrent GC algorithms are very difficult to get correct.

finalize() method called when GC collects the object.

Native Interface

When your Java program calls a **native** method, one has to convert the JVM parameters (e.g., what is on the stack) into machine registers (e.g, x86) following the calling convention.

```
class Test {

 public static native int print(double d);

  public static void main(String args[]) {

    print(5.2);

}}
```

As the JVM interpreter is executing the Java program, at some point it may have to call a native method: some code written in C or C++ say.

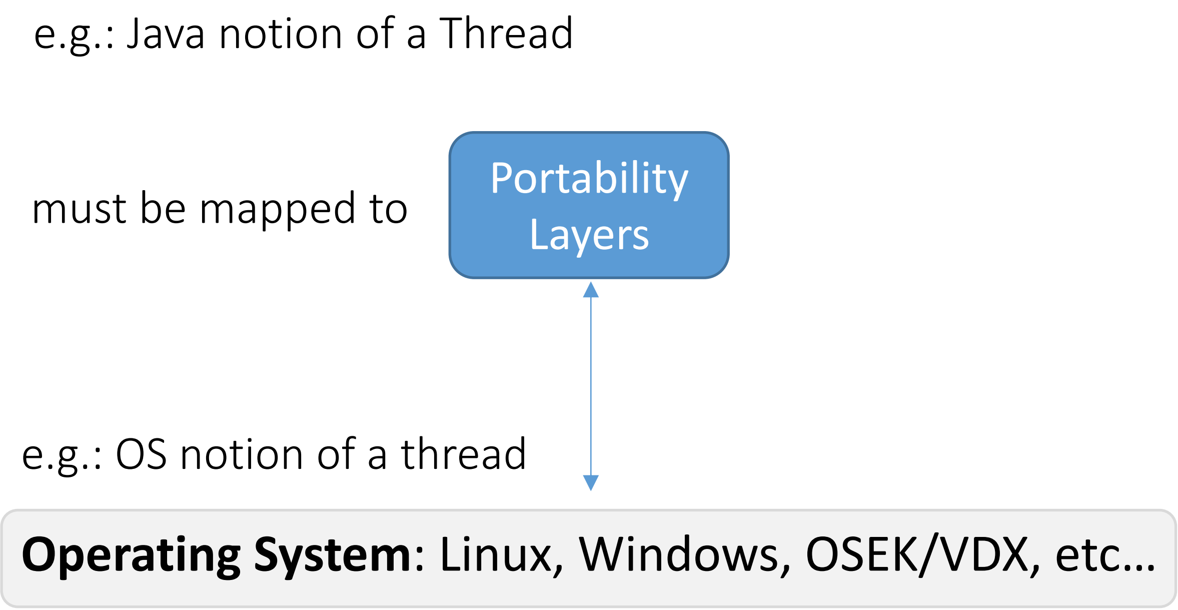
To do so, the JVM has to pass the parameters to the native method in a particular way so to interact with binaries on the particular platform.

This is not a large module but can be tricky to get the types correct.

java.lang.Object contains many native methods (e.g., starting a thread) for which the JVM provides an **internal implementation**.

Portability Layer

When the JVM is to run on top of Windows vs. Linux or x86 vs. ARM, the JVM designer must implement a small number of JVM constructs (e.g., synchronization, threading) using the primitives of the underlying operating system and architecture.



Example: Java provides its own notion of a thread (as **we will see later**). However, the operating system has a different notion of what a thread is. So this layer somehow needs to come up with a way to use the OS notion of a thread to provide the Java notion of a thread that the Java programmer expects.

If you want your JVM to run on a different OS, you need to write the portability layer.

Look inside via: javap –c Test

This kind of usage of ‘javap’ is not examinable but it may help you to get deeper understanding of how the Java language actually gets executed.

We will see how this is helpful later with constructs such as synchronized and volatile and why this is instructive.

You can find the meaning of all JVM instructions here:
https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5

Lets look a bit at the bytecodes…

```
class Test {

 static int x = 2018;
 double d;

 public static native int print(double d);
 public double pp(int a) {   return a;  }

 public static void main(String args[]) {
    Test t = new Test();
     t.d = t.pp(1)  + x;
     Test.print(t.d);
 }
}
```

javac Test.java

Test.class

What is inside here?

```
class Test {
  static int x;
  double d;

  Test();
   Code:
     0: aload_0
     1: invokespecial #1    // Method java/lang/Object."<init>": ()V
     4: return

  public static native int print(double);

  public double pp(int);
   Code:
     0: iload_1
     1: i2d
     2: dreturn
```

Constructor for class Test

- 0: aload_0 — Pushes content of local variable 0 (note: the variable is of a reference type) to the stack.
- 1: invokespecial #1 — Invoke constructor for the superclass of Test, that is, java.lang.Object…and clear the stack.
- public static native int print(double); — Native method. Its implementation could be provided for example in a C/C++ library.
- 0: iload_1 — Pushes content of local variable 1 (type integer) to stack.
- 1: i2d — convert the integer on the stack to a double.
- 2: dreturn — Pop value from stack and return it.

```
static {};
   Code:
     0: sipush      2018
     3: putstatic    #5        // Field x:I
     6: return
}
```

JVM invokes this code **before** main()

- 0: sipush 2018 — push constant 2018 of type short (hence: si) to stack
- 3: putstatic #5 — pop 2018 from stack and write it to static field x.

```
public static void main(java.lang.String[]);
  Code:
    0: new          #2                 // class Test
    3: dup
    4: invokespecial #3                // Method "<init>":()V
    7: astore_1
    8: aload_1
    9: aload_1
   10: iconst_1
   11: invokevirtual #4                // Method pp:(I)D
   14: getstatic     #5                // Field x:I
   17: i2d
   18: dadd
   19: putfield      #6                // Field d:D
   22: aload_1
   23: getfield      #6                // Field d:D
   26: invokestatic  #7                // Method print:(D)I
   29: pop
   30: return
```

- 0: new — Create the object of class Test and push it on the stack. Object **not yet initialized**! Triggers the JVM's memory allocator.
- 3: dup — Duplicate object reference on the stack.
- 4: invokespecial — Invoke constructor for top-of-stack object (pops it). **This initializes** it as we saw before. After, 1 reference to object remains
- 7: astore_1 — Store top-of-stack reference in local variable and pops it. Stack now empty.
- 8: aload_1 — Load reference from local variable 1 onto the stack.
- 9: aload_1 — Load reference from local variable 1 onto the stack again.
- 10: iconst_1 — push the constant 1 onto the stack
- 11: invokevirtual — Invoke method pp(). Pops the constant 1 and the reference. Method returns a value stored on top of the stack.
- 14: getstatic — Read the value of the static field 'x' and push it onto the stack.
- 17: i2d — Convert to a double.
- 18: dadd — Add two values on top of stack (pop them) and produce 1 value.
- …..// we skip 19, 22 and 23 here: can do it yourself.
- 26: invokestatic — We are invoking a **native method**! The native interface component of the JVM will make sure local vars/stack information is converted to the particular binary interface…

Today

- Sneak peek into JVMs
  - Brief overview of JVM components …
  - … spending a bit more time on bytecode
  - This material is **<u>not examinable</u>.** Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
  - Certain constructs and patterns that are connected to concurrency
  - More recap slides in the slide deck, not covered in class

Structure of a Java program

class: a program

```
public class name {
    public static void main(String[] args) {
        statement;
        statement;
        ...
        statement;
    }
}
```

method: a named group of statements

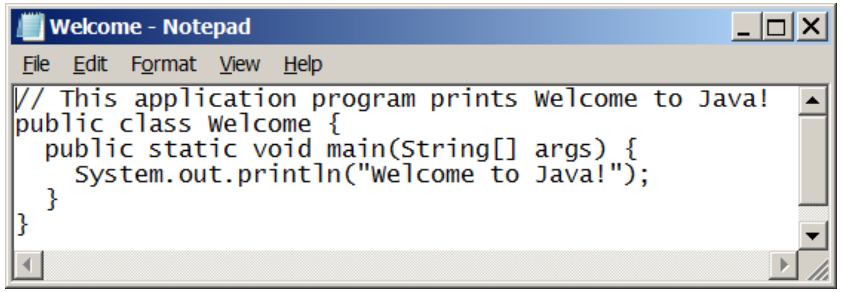
statement: a command to be executed

Every executable Java program consists of a **class** that contains a **method** named `main`, that contains the **statements** (commands) to be executed.

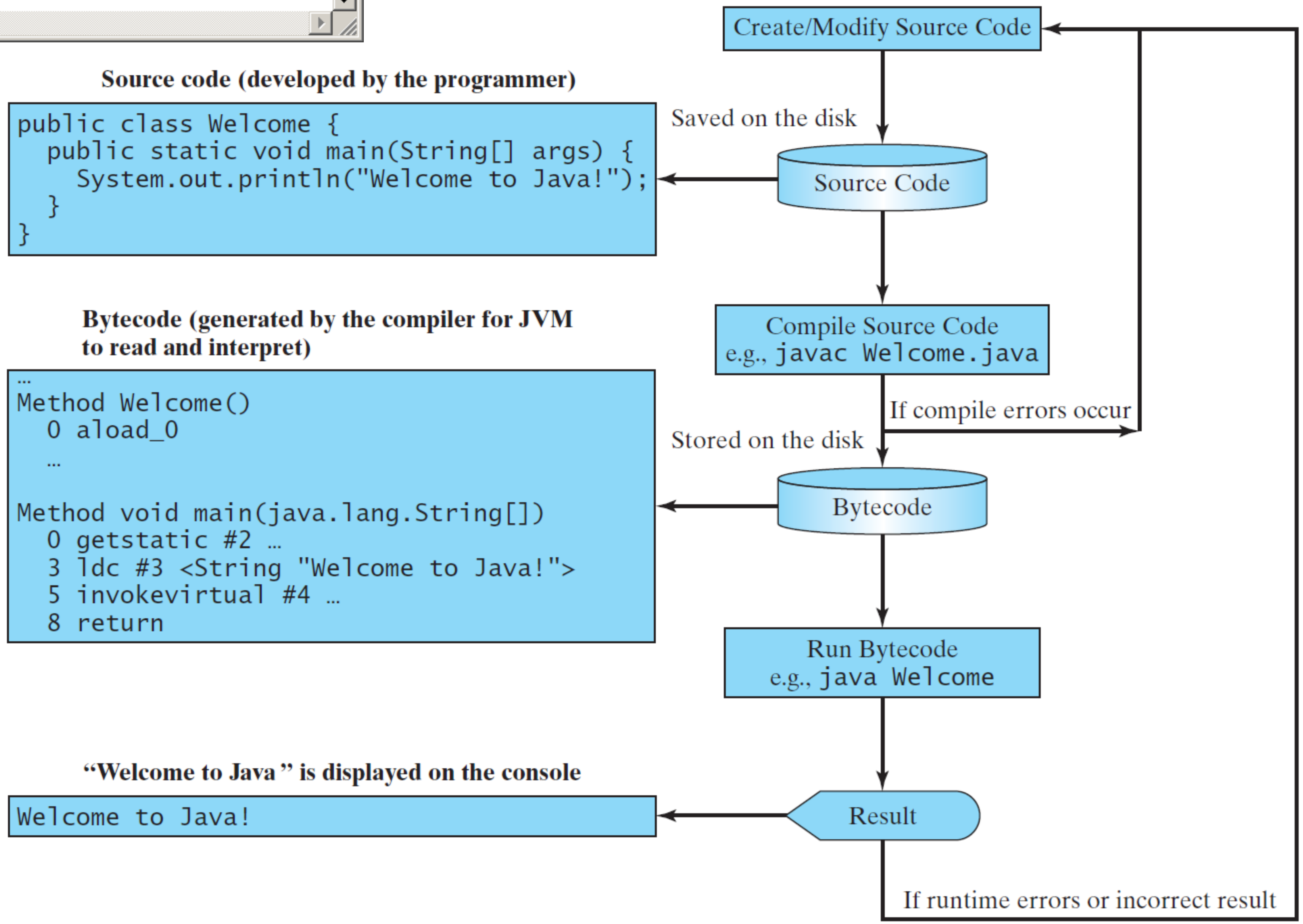
Keywords

Keyword: An identifier that you cannot use (to name methods, classes, etc) because it already has a reserved meaning in Java.

```
abstract    default    if          private       this
boolean     do         implements  protected     throw
break       double     import      public        throws
byte        else       instanceof  return        transient
case        extends    int         short         try
catch       final      interface   static        void
char        finally    long        strictfp      volatile
class       float      native      super         while
const       for        new         switch        ...
continue    goto       package     synchronized
```



Creating, Compiling, and Running Programs



Different kinds of errors

1. Compiler errors
2. Runtime errors
3. Logic errors

Data types

Type: A category or set of data values.

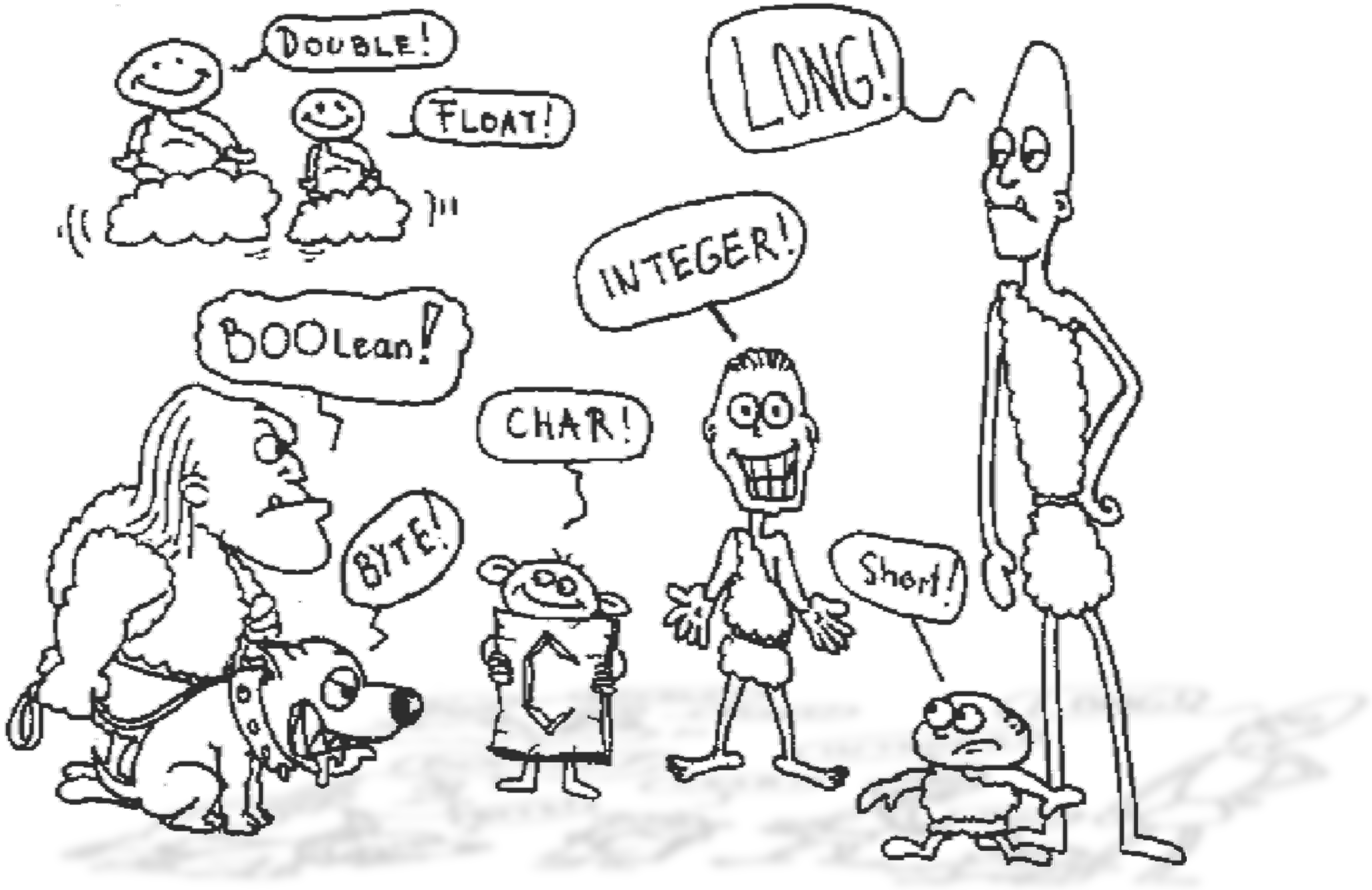
- Constrains the operations that can be performed on data
- Many languages ask the programmer to specify types

- Examples: integer, real number, string

Internally, computers store everything as 1s and 0s

```
104 → 01101000
```

Primitive Types in Java



Source: Christian Ullenboom: *Java ist auch eine Insel*,
Galileo Computing, 8. Auflage, 2009, ISBN 3-8362-1371-4

Java's primitive types

Primitive types: 8 simple types for numbers, text, etc.

Name	Description		Examples
`int`	integers	(up to $2^{31} - 1$)	`42, -3, 0, 926394`
`double`	real numbers	(up to 10^{308})	`3.1, -0.25, 9.4e3`
`char`	single text characters		`'a', 'X', '?', '\n'`
`boolean`	logical values		`true, false`

Why does Java distinguish integers vs. real numbers?

Primitive Types – Complete List

Integer:

- **byte** (8 Bits)
- **short** (16 Bits)
- **int** (32 Bits)
- **long** (64 Bits)

Range: -2147483648 ... 2147483647
or -2^{31} to 2^{31}-1

Real numbers:

- **float** (32 Bits)
- **double** (64 Bits)

Examples: 18.0 , -0.18e2 , .341E-2

Characters (Unicode):

- **char** (16 Bits)

65536 different values, allows for non-English characters

Booleans:

- **boolean**

Values: **true** , **false**
Operators: &&, | |, !

Type conversion (Casting)

Java is a strongly typed language, so the compiler can detect type errors at compile time

```
int myInt;
float myFloat = -3.14159f;

myInt = myFloat;
```

Compile time error

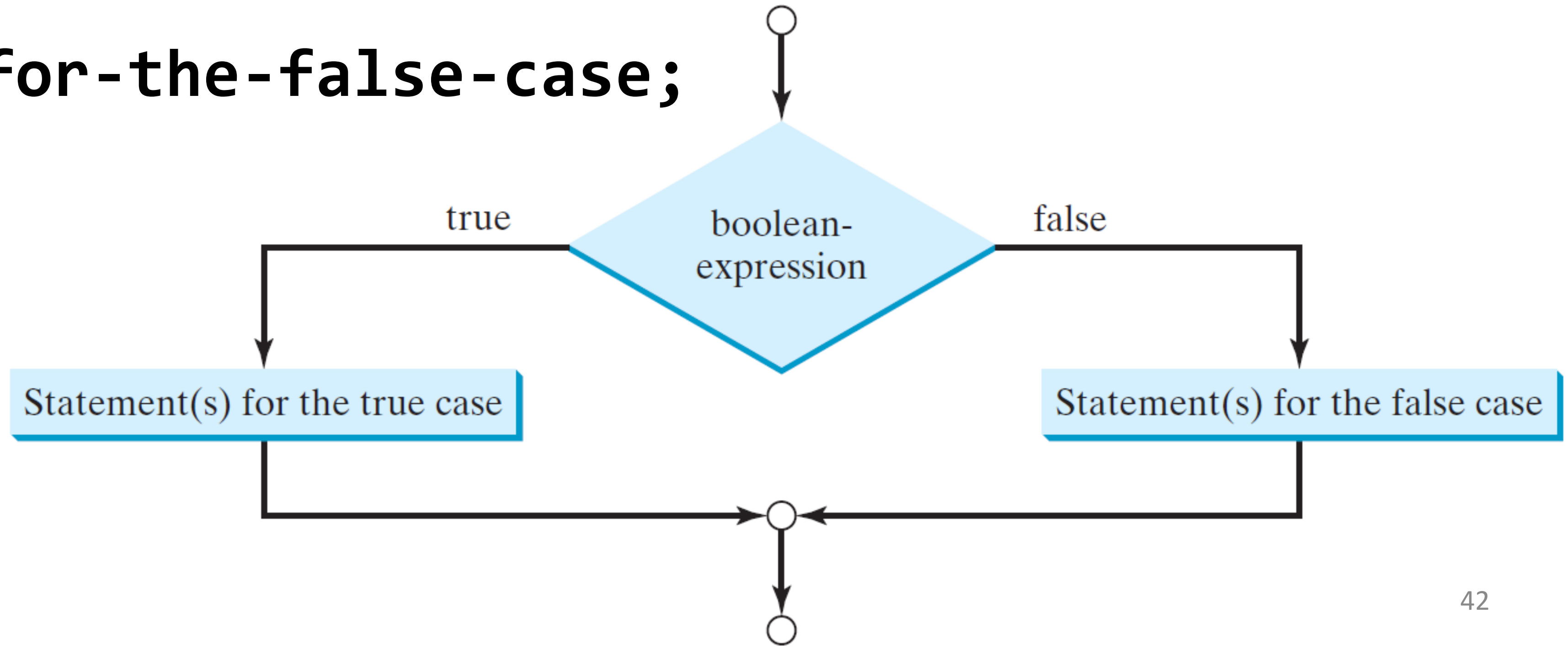
```
int myInt;
float myFloat = -3.14159f;

myInt = (int)myFloat;
```

Explicit type cast (truncates value)

```
Bytecode:
    0: ldc        #2             // float -3.14159f
    2: fstore_2
    3: fload_2
    4: f2i
    5: istore_1
    6: return
```

The Two-way `if` Statement

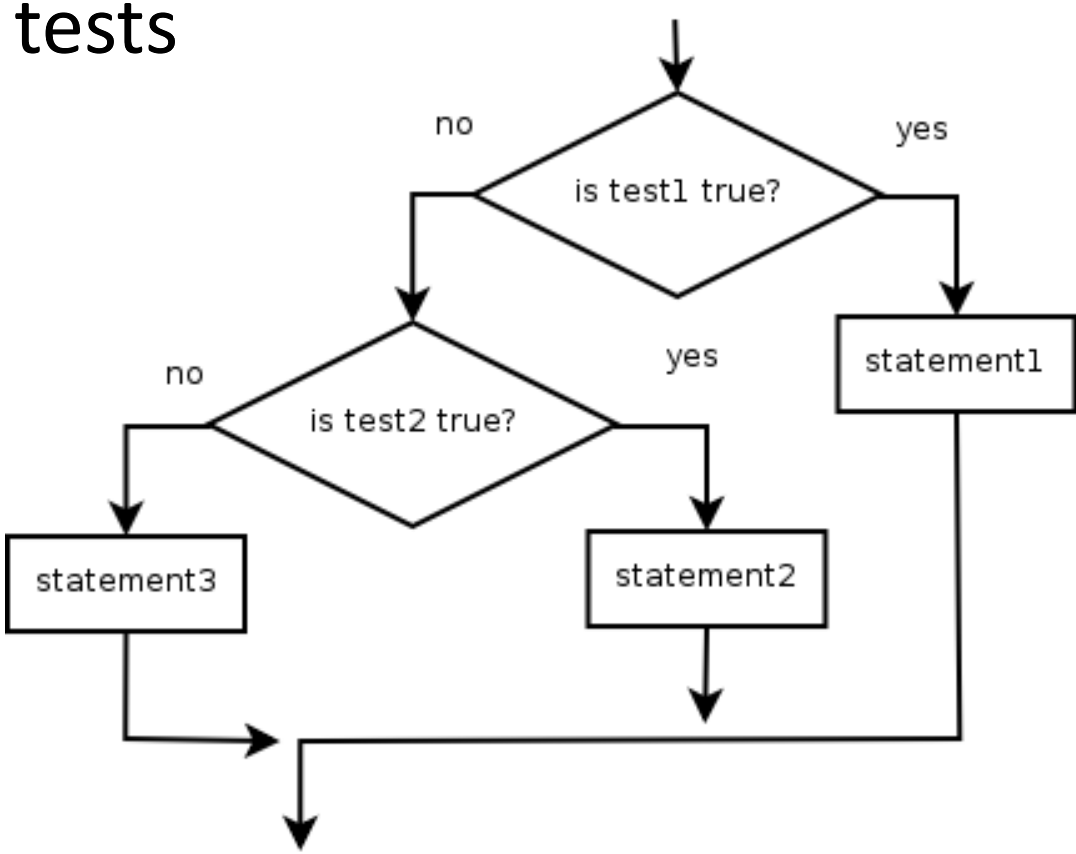
```
if (boolean-expression) {
      statement(s)-for-the-true-case;
}
else {
      statement(s)-for-the-false-case;
}
```



Nested `if/else`

Chooses between outcomes using many tests

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```



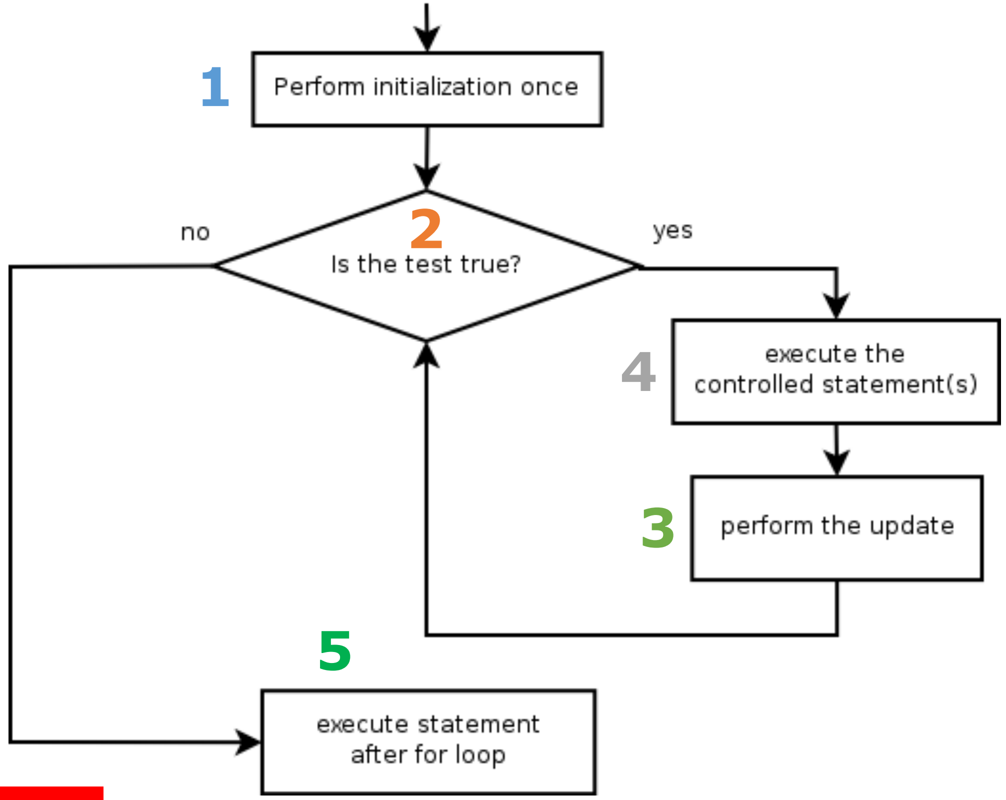
Example:

```
if (x > 0) {
    System.out.println("Positive");
} else if (x < 0) {
    System.out.println("Negative");
} else {
    System.out.println("Zero");
}
```

Tip: in parallelism/concurrency…try to have the if /else's read from a local variable.

Loops: walkthrough

```
                1           2      3
for (int i = 1; i <= 4; i++) {
    System.out.println(i + " squared = " + (i * i));4
}
System.out.println("Whoo!"); 5
```



Output:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
Whoo!
```

What can happen if 'i' is not a local variable and we have many processes? ☺

Categories of loops

Bounded loop: Executes a known number of times.

- The `for` loops we have seen are bounded loops.
  - print "hello" 10 times.
  - print each odd number between 5 and 127.

Unbounded loop: where number of times its body repeats is unknown in advance.

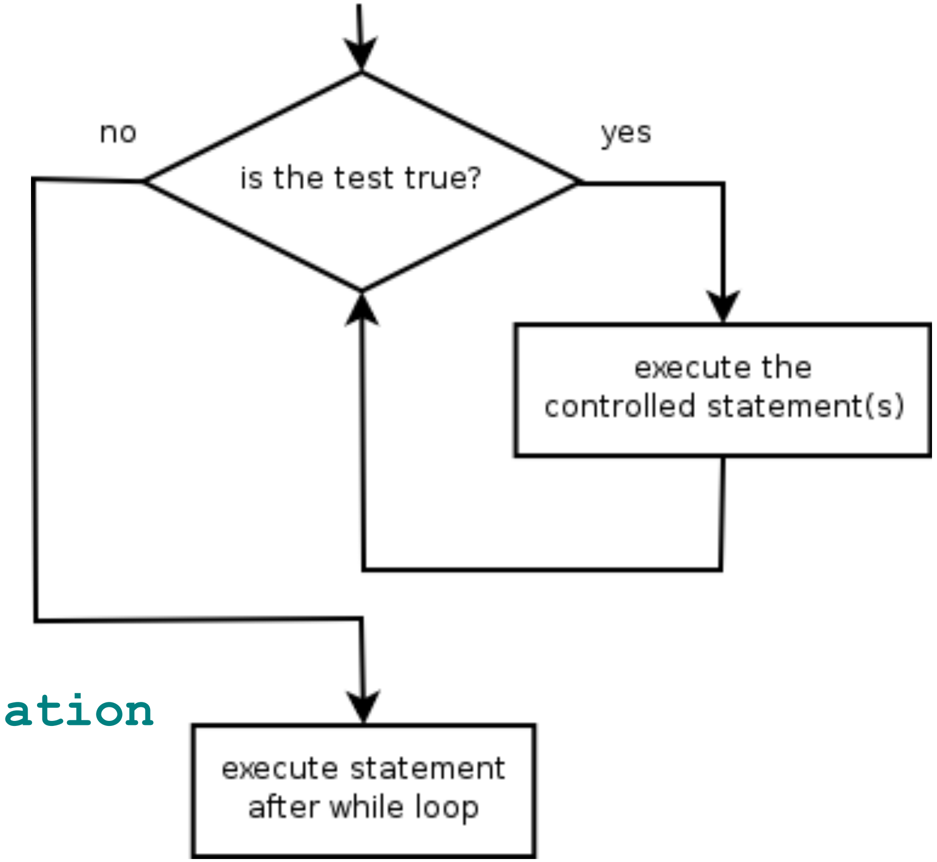
  - e.g. repeat until the user types "q" to quit.

How would you write the mutual exclusion algorithm for a single participant from last lecture?
What kind of loop would we use?

The `while` loop

`while` loop: repeatedly executes its body as long as a logical test is true.

```
while (test) {
    statement(s);
}
```



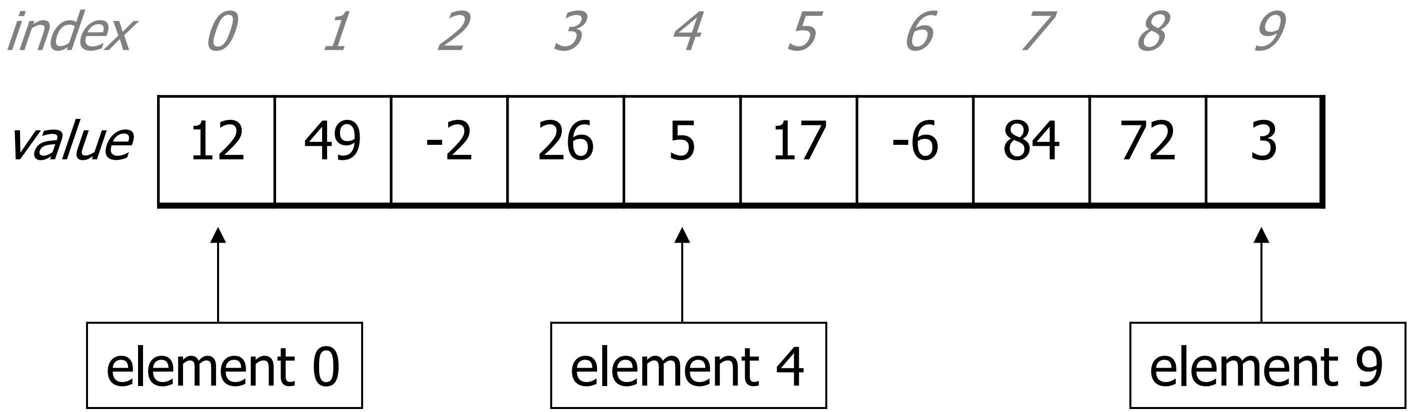
Example:

```
int num = 1;                      // initialization
while (num <= 200) {              // test
    System.out.print(num + " ");
    num = num * 2;                // update
}
// output:  1 2 4 8 16 32 64 128
```

Arrays

Array: object that stores many values of the same type.

- **element**: One value in an array.
- **index**: A 0-based integer to access an element from an array.



How are multi dimensional arrays represented?

Example: int [] [] PP = new int[10][20]

Strings

String: An object storing a sequence of text characters.

- Unlike most other objects, a `String` can be created without `new`.

```
String name = "text";
String name = expression;
```

- Examples:

```
String name = "ETH 2021 Parallel Programming";
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```

Indexes

Characters of a string are numbered with 0-based *indexes*:

```
String name = "R. Kelly";
```

index	0	1	2	3	4	5	6	7
character	R	.		K	e	l	l	y

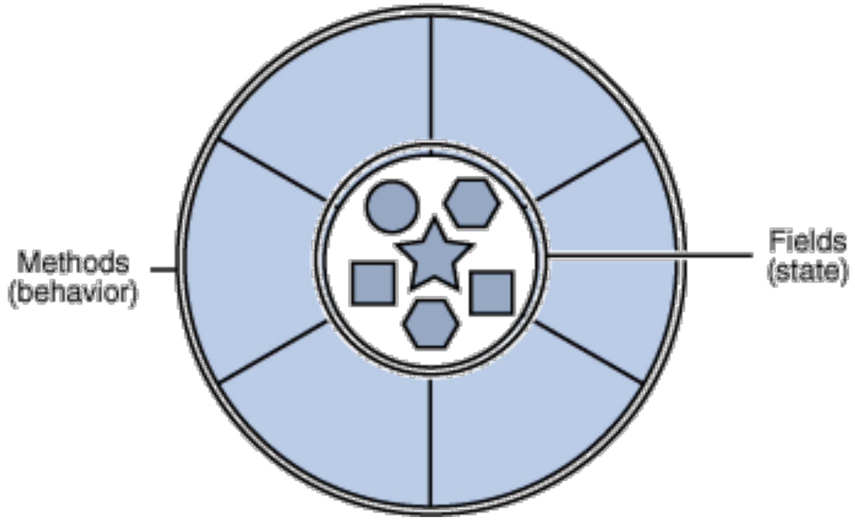
- First character's index : 0
- Last character's index : 1 less than the string's length

- The individual characters are values of type `char`

Objects

Object: An entity that contains data and behavior.

- *data*: variables inside the object
- *behavior*: methods inside the object
  - You interact with the methods; the data is hidden in the object.



Constructing (creating) an object:

Type objectName = `new` **Type** (**parameters**) ;

Calling an object's method:

objectName . methodName (**parameters**) ;

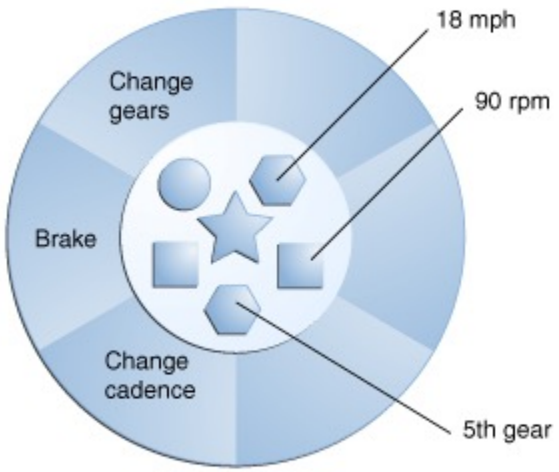
http://docs.oracle.com/javase/tutorial/java/concepts/object.html

Bicycle: An example

Software objects are similar to real world objects

They have *states* and *behaviors*

- *States* are stored in variables (Java: fields)
- *Behavior* is exposed via methods



Methods are used for object-to-object communication

Methods hide the internal state (and the concrete implementation) from the outside world (developer)

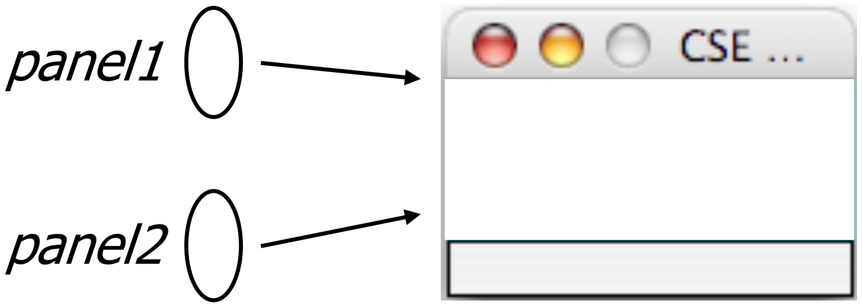
http://docs.oracle.com/javase/tutorial/java/concepts/object.html

Object references

Arrays and objects use reference semantics.

- *efficiency*. Copying large objects slows down a program.
- *sharing*. It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;    // same window
panel2.setBackground(Color.CYAN);
```

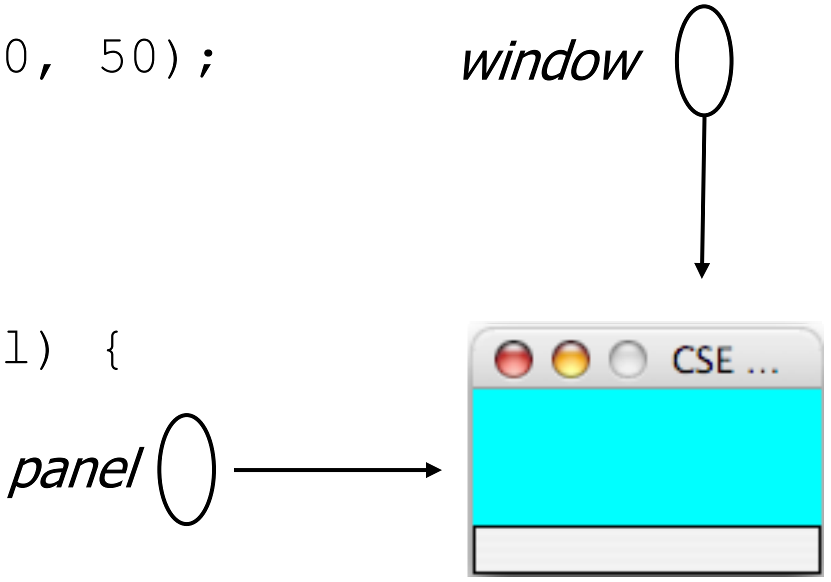


Pass by reference value (Objects)

When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.

- If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```



Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are *not* tied to a specific object.

Static constants are final variables shared by all the instances of the class.

CONCURRENCY ISSUES?

Instance Variables, and Methods

Instance variables belong to a specific class *instance*.

Instance methods are invoked by an *instance* of the class.

Exceptions

Advantages of exception handling:

- enables a method to throw an exception to its caller
- without this, a method must handle the exception or terminate the program

```
public static int quotient(int number1, int number2) {
   if (number2 == 0)
     throw new ArithmeticException("Divisor cannot be zero");

   return number1 / number2;
 }
```

Language features vs. parallelism: Guidelines

- Keep variables as ‘local’ as possible: global variables means they can be accessed by various parallel activities. While when its local to the process/thread, we are safe against inadvertent accesses to the variable.
- If possible, avoid aliasing of references: aliasing can lead to unexpected updates to memory through a process that accesses a seemingly unrelated variable (named differently).
- If possible, avoid mutable state, in particular when aliased: aliasing is no problem if the shared object is immutable, but concurrent mutations can make bugs *really* hard to reproduce and investigate (“Heisenbugs”)

Language features vs. parallelism: Guidelines

- Exceptions vs. Concurrency/Parallelism:
    - Exceptions tend to be less effective with parallelism because the cause of the error may be far earlier in the execution than where the exception triggers. Hence, the stack trace of the exception can be less informative and useful.
    - Exceptions thrown in a thread (parallel process) don't automatically reach the main program, and thus might go completely unnoticed. This can make it (even) more complicated to track down the root cause of a bug.

Summary

- Sneak peek into JVMs
    - Brief overview of JVM components …
    - … spending a bit more time on bytecode
    - This material is **not examinable.** Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
    - Certain constructs and patterns that are connected to concurrency
    - More recap slides in the slide deck, not covered in class