CS 202
Quick Java Review

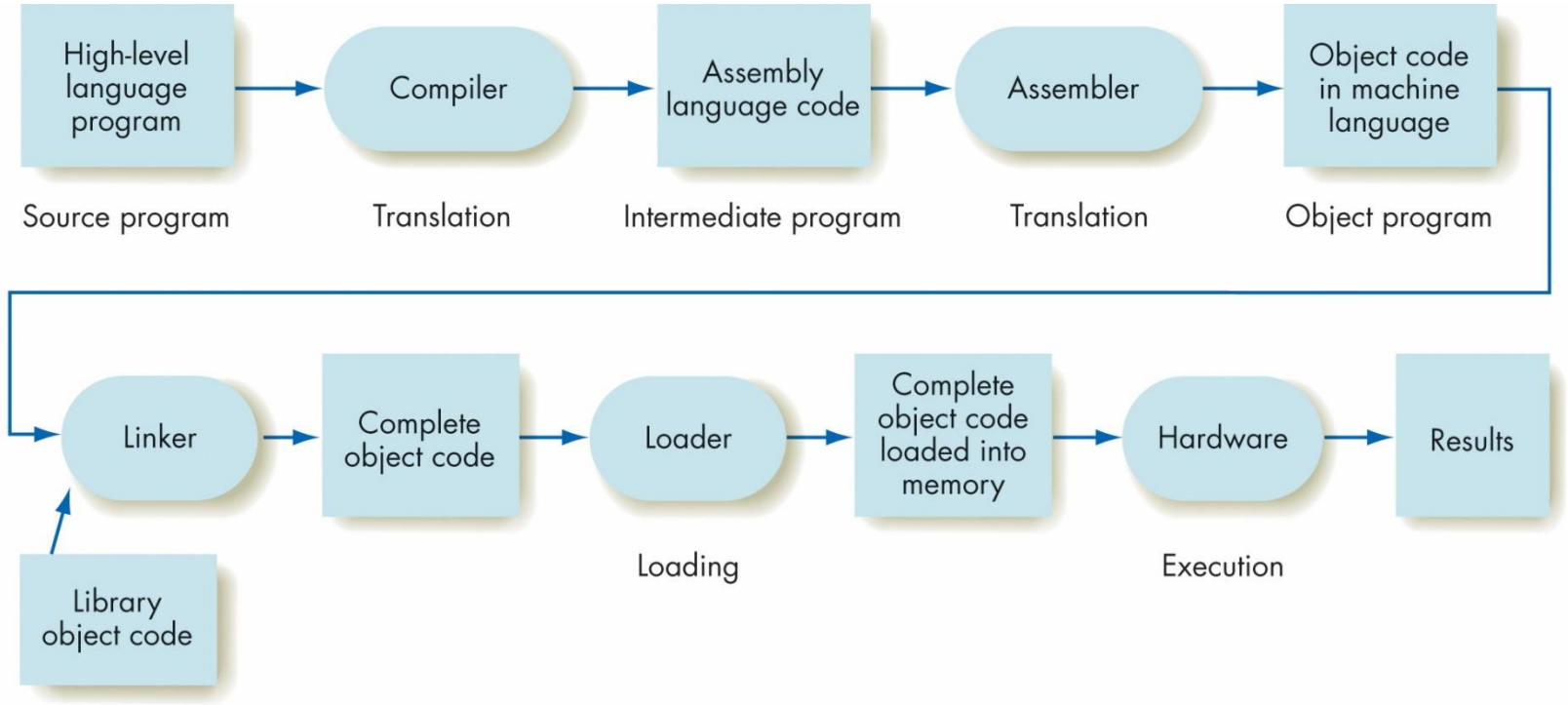
- Definition
  - Algorithm: method for solving a problem.
  - Data structure: method to store information.

- How to solve a problem
  - Define the problem
  - Find algorithm
  - Time/Space efficiency analysis
  - Improvement

- Learning the basics of important data structures
  - How to implement them
  - How to evaluate them
  - How to decide when to use them
- Practice programming skills through assignments

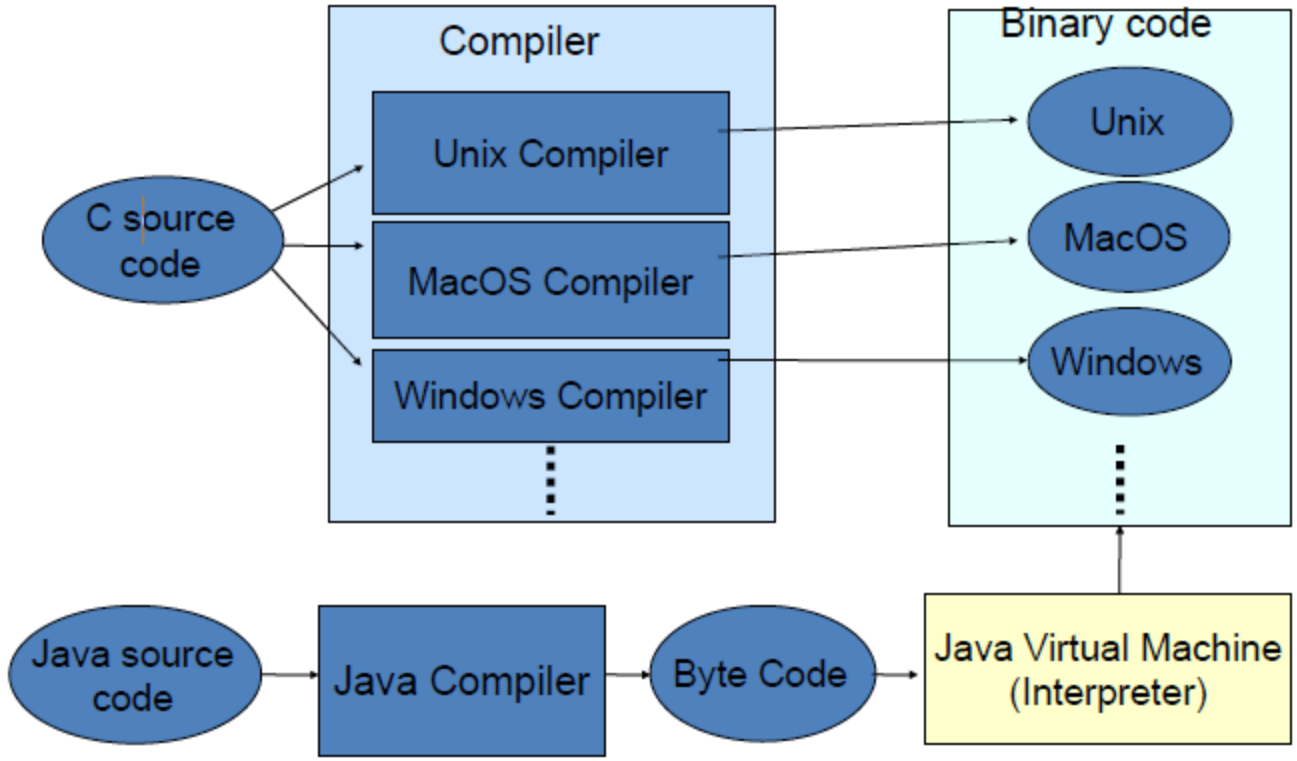
Java Review

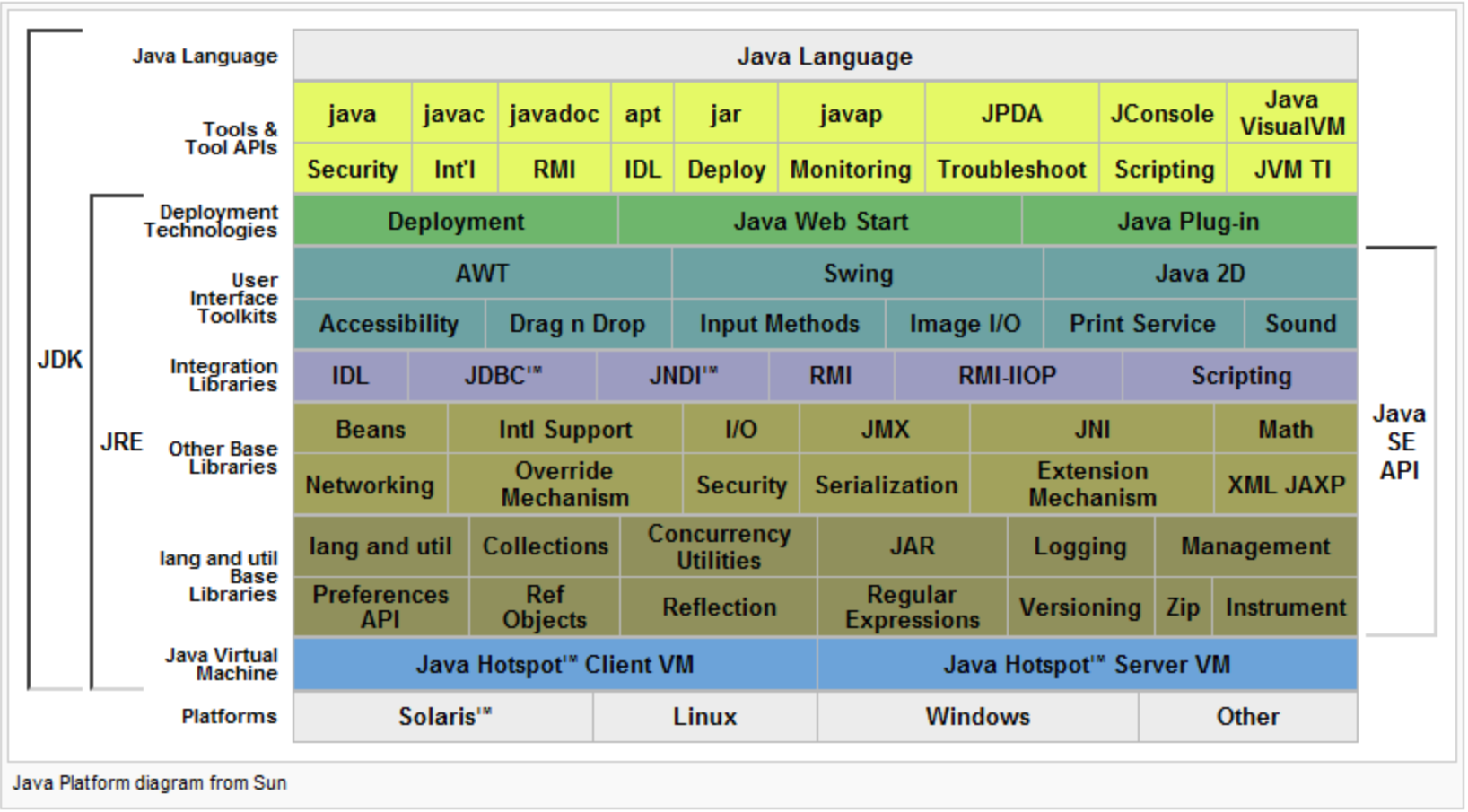
- Review of Java Programming Language
- Programming environments
- Language constructs
- Class, reference, and instantiation
- Inheritance, accessibility
- Exception and error handling
- Input and output



Transitions of a High-level Language Program

Java is “Platform-Independent”





Java Platform diagram from Sun

IDE	License	JVM	Platforms	GUI builder
BEA Workshop for WebLogic	Proprietary	Yes		Yes
BlueJ	GPL2+GNU linking exception	Yes	Windows, Mac OS X, Linux, Solaris	No
DrJava	Permissive	Yes	Windows, Mac OS X, Linux, Solaris	No
Eclipse JDT	EPL	Yes	Windows, Mac OS X, Linux, Solaris	No
Geany	GPL	No	Windows, Mac OS X, Linux, Solaris	No
Greenfoot	GPL	Yes	Windows, Mac OS X, Linux, Solaris	No
IntelliJ IDEA	Proprietary	Yes	Windows, Mac OS X, Linux	Yes
JBuilder	Proprietary	Yes	Linux, Solaris, Windows	Yes
JCreator	Proprietary	No	Windows	No
JDeveloper	Proprietary OTN JDeveloper License (freeware)	Yes	Windows, Mac OS X, Linux, generic JVM	Yes
jGRASP	Proprietary (freeware)	Yes	Windows, Mac OS X, Linux	No
KDevelop	GPL	No	Windows, Mac OS X, Linux, Solaris	Unknown
Monodevelop	GPL	No	Windows, Mac OS X, Linux, Solaris	Yes
MyEclipse	Proprietary	Yes		Yes
NetBeans	CDDL, GPL2	Yes	Windows, Mac OS X, Linux, Solaris	Yes
Rational Application Developer	Proprietary	Yes	Windows, Mac OS X, Linux, Solaris, AIX	Yes
Servoy	Proprietary	Unknown		Unknown
Xcode	Proprietary (freeware)	No	Mac OS X	No

Simple Example

- HelloWorld.java

```
class HelloWorld{
    public static void main(String args[]) {
    System.out.println("Hello, World!");
    }
}
```

- File name usually match class name.

Compile/Run

- Command to compile a Java class:
  - Javac HelloWorld.java
- Command to execute the main method of a Java class:
  - java HelloWorld

Compile/Run

- Using Integrated Development Environment makes your life much easier!
- We recommend Eclipse or NetBeans
  - Cross platform
  - Easy to program, debug, linking to jar files, getting document help…

“Primitive” Data Types

- Integral (not a class!)
  - int i=100;
- Decimal (not a class!)
  - float height;
  - double weight;
- Logical (not a class!)
  - Boolean isover;
- Character (not a class!)
  - char answer = ‘y’;

Wrapper Classes

- Each primitive type has a “wrapper class”.
  - Integer, Double, Character, Boolean ...
  - Contain added functionality.

- Example: String Class
  - String message;
  - String className= “CS202”;
  - className.contains(“202”)

- **API specifications:**
  https://docs.oracle.com/javase/10/docs/api/overview-summary.html

Expressions

- Add/Subtract
  x = y + 3;
  m = n –4;

- Multiply/Divide
  x = y / 3;
  m = 2 * 6;

- Boolean
  done = (x==y)&&((!z>10)||(z>=1));

Conversion between types

- Is this correct?

  double a = 5;

  correct
- Is this correct?

  int a = 5.0;

  incorrect

Conversion between types

- Low precision -> high precision
  - no problem, automatic conversion
- High precision -> low precision
  - needs explicit conversion
- float a = 5.5;int b = (int)a;

Conversion between types

Examples:

```
int x = 5;
float y, z;
y = x/2;
z = (float)x/2;
```

Results for y and z are different

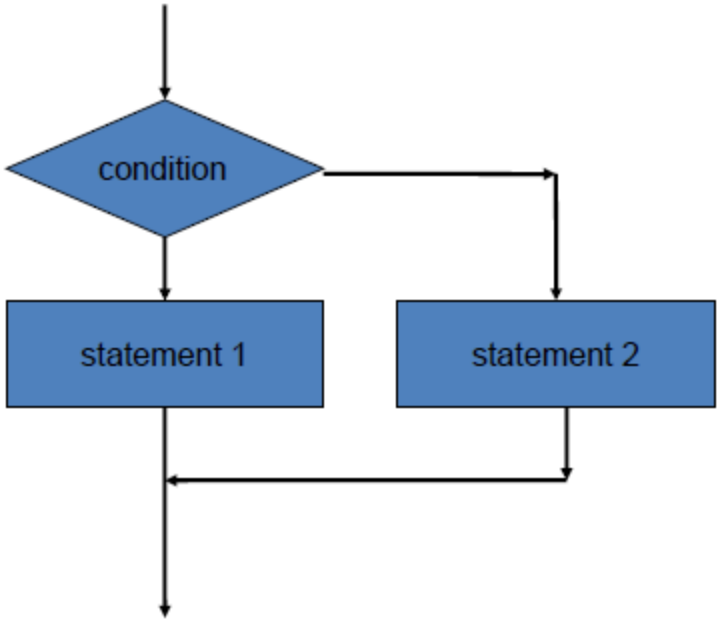
```
y = 2;
z = 2.5;
```

Flow Control Statements

- *if-else statements*
- *for loop*
- *while loop*
- *do-while loop*

if-else branching statements

```
if (<condition>)
{
    <statement 1>;
    …
}
else
{
    <statement 2>;
    …
}
```



if-else statements

- Given a positive integer n, is it odd or even?

```
if (n%2==0)
        System.out.println(n+“ is even”);
else
        System.out.println(n+“ is odd”);
```

if-else statements

- Another example:
  - Grading policy based on one exam.

85<= your score<=100, you will get grade A,
75<= your score < 85, you will get grade B;

65<= your score < 75, you will get grade C;

60<= your score < 65, you will get grade D;
Otherwise, you will get grade E;

if-else statements

```
if (yourscore>=85 && yourscore<=100)
    yourGrade= ‘A’;
else if (yourscore>=75 && yourscore<85)
    yourGrade= ‘B’;
else if (yourscore>=65 && yourscore<75)
    yourGrade= ‘C’;
else if (yourscore>=60 && yourscore<65)
     yourGrade= ‘D’;
else
    yourGrade= ‘E’;
```

if-else statements

```
if (yourscore>=85)
      yourGrade= ‘A’;
else if (yourscore>=75)
      yourGrade= ‘B’;
else if (yourscore>=65)
      yourGrade= ‘C’;
else if (yourscore>=60)
    yourGrade= ‘D’;
else
      yourGrade= ‘E’;
```

if-else statements

```
if (choice == 1)
   order = “scrambled egg”;
else if (choice == 2)
   order = “blueberry pancake”;
else if (choice == 3)
   order = “italian sausage”;
else
   order = “bacon”;
```

switch-case statements

```
switch(order) {
case 1:
order = “scrambled egg”;
case 2:
order = “blueberry pancake”;
case 3:
order = “italian sausage”;
default:
order = “bacon”;
}
```

What’s the problem here?

switch-case statements

```
switch(order) {
    case 1:
    order = “scrambled egg”;
    break;
    case 2:
    order = “blueberry pancake”;
    break;
    case 3:
    order = “italian sausage”;
    break;
    default:
    order = “bacon”;
}
```

Looping Constructs

- *for loop usually uses a counter for control*

- *While and do-while loops*
  - *usually uses a Boolean statement for control*

for loop

for(<initialize counter>; <condition>; <update counter>)

{

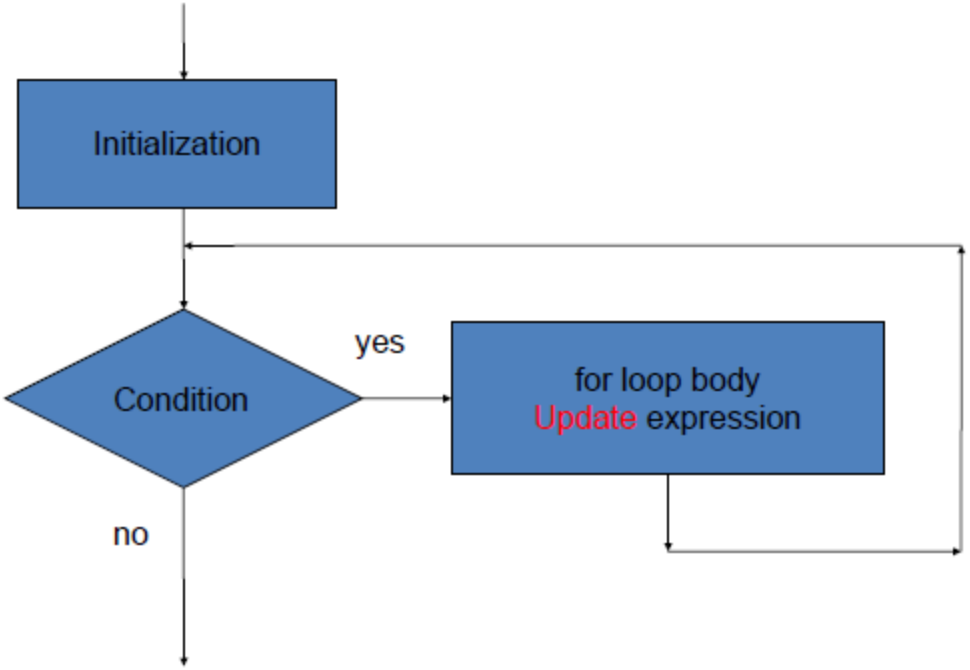
<statement 1>;

<statement 2>;

… …

}

for loop


Initialization
Condition
yes
for loop body
Update expression
no

A simple example

- Print *n asterisks*

```
for (i=0 ; i<n ; i++)
{
    System.out.println( “*” );
}
```

Display Rectangle

- Display a 3x4 rectangle formed by *

```
****
****
****
```

Nested for loop

```
for (i=0; i<3; i++)
{
    for(j=0; j<4; j++);
    System.out.print(“*”);
    System.out.println();
}
```

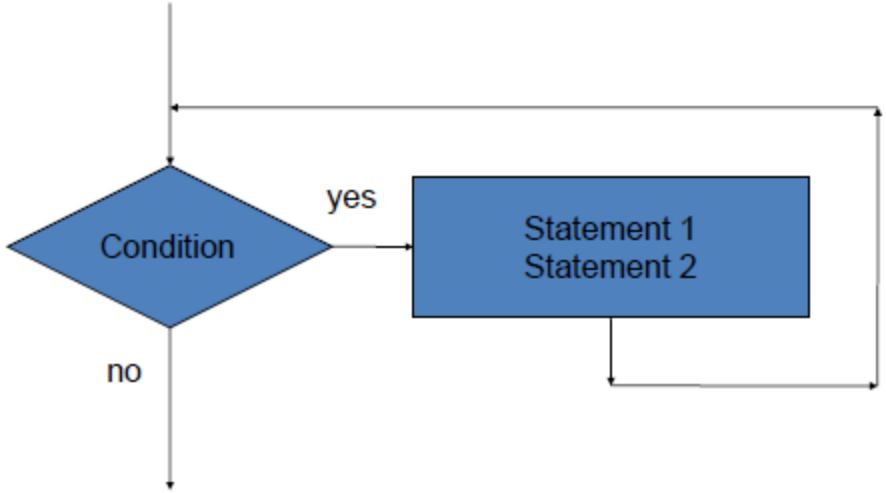
Two Loop Constructs

- *while loop*
  - *Check loop condition*
  - Then loop body

- *do-while loop*
  - Runs loop body first
  - Then checks condition
  - runs body at least once

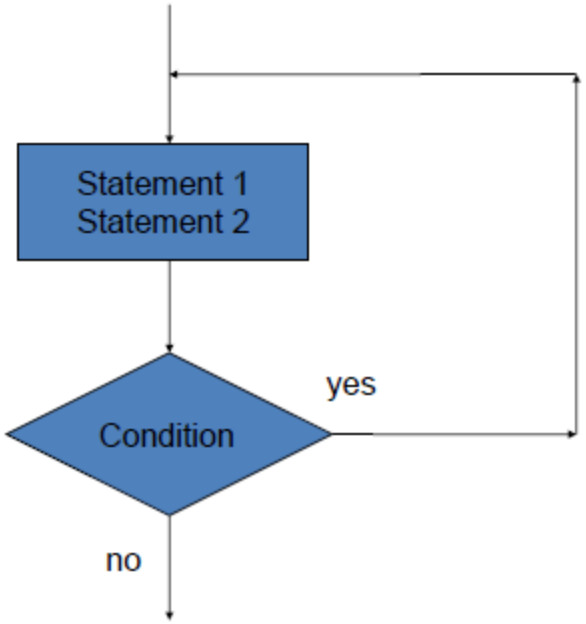
while statement

```
while (<condition>)
{
<statement1>;
<statement2>;
}
```



do-while statement

```
do {
   <statement1>;
   <statement2>;
   } while (<condition>);
```



Compounding Interest

- Suppose you have $1,000 in a savings account that earns 4% interest per year.
- How many years until you double your money?

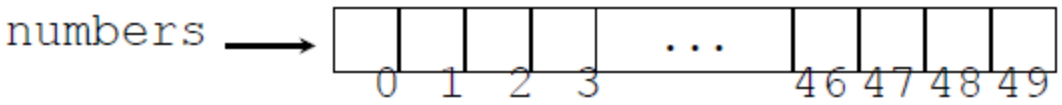
Compounding Interest

```
double my_money= 1000.0;
lnt n = 0;
while (my_money< 2000.0) {
    my_money= my_money* 1.04;
    n = n + 1;
}
System.out.println(“My money will double in ”
+ n + “ years”);
```

Arrays

- An array is a contiguous piece of storage for elements of compatible types; it's just a reference:

```
int[] numbers = new int[50];
for (inti= 0; i< numbers.length; i++) {
        numbers[i] = i;
}
```



numbers.length == 50

Exercise

- Write a function int getLargerValue(int x, int y) in Java, where *x, y are integer. This* function returns the larger value of two integers *x* and *y.*

```
int getLargerValue(int x, int y) {
    if(x >= y) return x;
    else return y;
}
```

Exercise

- Each of the following pieces of code has a problem/bug that causes either a compilation error or an execution error. Explain what the problem is in one sentence.

a)
```
int[] array = {1, 2, 3, 4, 5 };
System.out.println(array[5]);
```

- Arrays begin their indices at 0. This array has 5 elements, indexed 0 through 4. The reference to array[5] will cause a runtime error since there is no such element.

Exercise

b) String a;

System.out.println(a);

- The String a is declared but never initialized. It will compile and print out null.

c) int a = 5.0;

System.out.println(a);

- An int variable is declared but a double literal is assigned to it. This will cause a compiler error since information is potentially lost.

Exercise

Consider a Java program that defines the following class:

```
import myLibrary;
public class JavaExample {
//code for the class
}
```

What should be the name of the Java file that contains this program?

a) myLibrary.java
b) JavaExample.java
c) JavaExample.class
d) JavaExample.txt

Exercise

Consider the following code:

```
String river = new String("CS202");
System.out.println(river.length());
```

What would be the output of the code?

a) 5

b) 6

c) 7

d) CS202

Class

- **Class**
  - Definition of a group of entities which contain both variables (data elements) and the associated operations (methods).
  - Think of it as a package that packs both variables and associated methods
  - A class is only a description, it does not yet create an object.
  - An object is created via instantiation:

```
Apple oneapple = new Apple();
```

Reference

- **Referencing** Objects:

  String msg;

  msg = new String(“hello world”);

  String welcome = msg;

  A reference assignment does not create a new object (e.g. allocate memory space) –it merely assigns the pointer to an existing object.

  An object can have multiple references:

  String hello = msg;

Instantiation

- **Instantiate** an Object

  String welcome = new String(msg);

  Instantiation creates a new object (e.g. allocate memory space) and performs initialization.

  Compare:

  String y = new String(“abc”);

  String z = “def”;

  String w = z;

Constructors

- **Constructors** are methods that are called when instantiating an object. The main purpose is to initialize relevant variables.

```
Class Apple {
        private float value;
        Apple(){ value=0.0f; }
        Apple(float v){ value = v; }
        … …
}
```

Constructors

- **Constructors** must have the same name with the class name; they must NOT have return value; they can be overloaded.
- **The default constructor**

  String z = new String();

  Compare the above with:

  String z;

Nested Class Definition

- You can certainly define a class within a class

```
class Apple {
    public class AppleTaste{
    ……
    }
    Apple(){…}
    Apple(float v) { … }
}
```

Static Variables / Methods

- Some variables / methods are defined as static:

```
class Apple {
        public static int value;
        public static void main(…)
        ……
}
```

- How are they different from other variables / methods?

Static Variables / Methods

- Static variables exist (are allocated in memory) without any class instantiation.
  - In contrast, other variables do not exist until you have an actual object (instance).

- All class instances refer to the same static variables (i.e. they exist globally)In contrast, other variables have unique local copies for each different object.

- Example:

  Math.PI;

Static Variables / Methods

- Static methods may be called without any class instantiation
  - In contrast, other methods cannot be called until you have an actual object (instance)

- Static methods cannot call non-static methods, or refer to non-static variables.
  - Non-static variables / methods do not even exist if you don’t have a instance yet.

Accessibility / Visibility

- Access to variables or methods respects the declared accessibility (visibility)
  - **public**= always accessible.
  - **protected**= accessibly only in the class and any inherited class
  - **private**= accessible only in the class itself.

Accessibility / Visibility

- Analogy: think of families and secrets
  - **public**= known facts by your neighbors
  - **protected**= secrets protected by family members (not known to your neighbors)
  - **private**= secrets owned by individuals (not even shared among family members)

- Think about the prelim exam question.

Parameter Passing

- You often need to provide parameters (arguments) when calling a method.
- Java passes parameters using **call-by-value**
  - For a primitive type (int, double…), the value is passed to the method being called.
    - This means the method cannot modify the original argument

Parameter Passing

- Java passes parameters using **call-by-value**
  - For a class type parameter, the value being passed is the reference to an object.
    - This means the method can actually modify class members

Parameter Passing

- Example 1:

```
public static void modify(int val) {
        val = 5;
}
……
public static void main() {
        int a = 10;
        modify(a);
        System.out.println(a);
}
```

Parameter Passing

- Example 2:

```
public static void modify(Point val) {
        val.x = 5;
}
……
public static void main() {
        Point a = new Point(0,0)
        modify(a);
        System.out.println(a.x);
}
```

Parameter Passing

- Example 3:

```
public static void modify(Point val) {
        val = new Point(5,5);
}
……
public static void main() {
        Point a = new Point(0,0)
        modify(a);
        System.out.println(a.x);
}
```

The Math Class

- Math class defines many useful math functions.
  - abs, min, max, floor, ceil …
  - log, pow, sin, cos, tan, sqrt …

Exception and Error Handling

- Exceptions provide a way to handle errors (often caused by I/O, such that the program cannot continue)

- **try-catch-finally** sequence

```
try {
// statements that perform I/O
} catch (IOException ex) {
System.out.println("Error occurred");
System.exit(1);
} finally {
}
```

Exception and Error Handling

- A lot of methods, especially I/O related, require you to handle exceptions.
- You either have to use **try-catch** to explicitly handle the exception, or you can use the **throws** clause to defer the handling to the calling method.
- Eventually an exception must be handled somewhere; otherwise the compiler will complain about un-checked exceptions.

Exception and Error Handling

- Exception can be a convenient way to replace messy nested if statements:

```
Step A
if (Step A successful) {
        Step B
        if (Step B successful) {
                Step C
                if (Step C unsuccessful) report error in Step C
                }
        else {
        report error in Step B
        }
} else {
report error in Step A
}
```

Exception and Error Handling

- Exception can be a convenient way to replace messy nested if statements:

```
try{
Step A
Step B
Step C
} catch (exception indicating C filed) {
report error in Step C
} catch (exception indicating B failed) {
report error in Step B
} finally {
report error in Step A
}
```

Object-Oriented Programming (OOP) in Java

- A **class** is a template in accordance to which objects are created
- Functions defined in a class are called **methods**
- Variables used in a class are called **class scope variables**, **data fields**, or **fields**
- The combination of data and related operations is called **data encapsulation**
- An **object** is an instance of a class, an entity created using a class definition

Encapsulation

- Objects make the connection between data and methods much tighter and more meaningful
- The first OOL was Simula; it was developed in the 1960s in Norway
- The **information-hiding principle** refers to objects that conceal certain details of their operations from other objects so that these operations may not be adversely affected by other objects

Class Methods and Class Variables

- Static methods and variables are associated with the class itself and are called **class methods** and **class variables**
- Non static variables and methods are called **instance variables** and **instance methods**
- The method `main()` must be declared as `static`

Generic Classes

```
class IntClass {
    int[] storage = new int[50];
    ..................
}

class DoubleClass {
    double[] storage = new double[50];
    ..................
}

class GenClass {
    Object[] storage = new Object[50];
    Object find(int n) {
        return storage[n];
    }
    ..................
}
```

Abstract Data Types

- An item specified in terms of operations is called an **abstract data type**
- In Java, an abstract data type can be part of a program in the form of an interface
- **Interfaces** are similar to classes, but can contain only:
  - Constants (**final** variables)
  - Specifications of method names, types of parameters, and types of return values
- **Abstract** classes
  - a class declared **abstract** can include defined methods

Abstract Data Types (continued)

```
interface I {
    void If1(int n);
    final int m = 10;
}
class A implements I {
    public void If1(int n) {
        System.out.println("AIf1 " + n*m);
    }
}
abstract class AC {
    abstract void ACf1(int n);
    void ACf2(int n) {
        System.out.println("ACf2 " + n);
    }
}
class B extends AC {
    public void ACf1(int n) {
        System.out.println("BACf1 " + n);
    }
}
```

Inheritance

- OOLs allow for creating a hierarchy of classes so that objects do not have to be instantiations of a single class
- **Subclasses** or **derived classes** inherit the fields and methods from their base class so that they do not have to repeat the same definitions
- A derived class can override the definition of a `non-final` method by introducing its own definition

Inheritance

- You can define a class by inheriting from another class:

  class FujiAppleextends Apple {

  public String origin;

  ……

  }

  The FujiAppleclass automatically inherits variables / methods defined in the Apple class

Inheritance

- The FujiApple class automatically inherits variables / methods defined in the Apple class
  - However, remember that only **public** and **protected** variables defined in the parent class are accessibly in the inherited class.
  - **private** variables are not accessible.

Polymorphism

- **Polymorphism** is the ability of acquiring many forms
- **Dynamic binding** is when the type of method to be executed can be delayed until run time
- **Static binding** is when the type of response is determined at compilation time
- **Dynamic binding** is when the system checks dynamically the type of object to which a variable is currently referring and chooses the method appropriate for this type

Polymorphism (continued)

```
class A {
   public void process() {
      System.out.println("Inside A");
   }
}
class ExtA extends A {
   public void process() {
      System.out.println("Inside ExtA");
   }
}
```

Polymorphism (continued)

then the code

```
A object = new A();
object.process();
object = new ExtA();
object.process();
```

results in the output

```
Inside A
Inside ExtA
```

Input and Output

- To use the classes for reading and writing data, the `java.io` package has to include the statement:

```
import java.io.*;
```

- To print anything on the screen, use the statements:

```
System.out.print(message);
System.out.println(message);
```

- To read one line at a time, the method `readLine()` from `BufferedReader` is used

Input/Output

- InputOutput.java
  - Input your name, print out your name
  - Input your age, print out our age
  - Input your height, print out your height
  - Using Scanner.java

InputOutput.java

```
import java.util.*;
class InputOutput{
    public static void main(String args[]) {
    Scanner S = new Scanner(System.in);
    String name;
    System.out.print("enter your name: ");
    name = S.next();
    System.out.println("Your name is: "+ name);
```

InputOutput.java

```
    int age;
    System.out.print("enter your age: ");
    age = S.nextInt();
    System.out.println("Your age is: " + age );

    float height;
    System.out.print("enter your height(feet): ");
    height = S.nextFloat();
    System.out.println("Your height is: " + height + " feet" );
    }
}
```

Exercise

(c) Consider the following class definition:

```
class MyObject {
public int A;
protected int B;
private int C;
}
```

What members of MyObject have visibility to (can be accessed by) any other class?

a) only A
b) only B
c) both A and B
d) none

Exercise

(d) Given the same class definition:

```
class MyObject {
public int A;
protected int B;
private int C;
}
```

What members of MyObject have visibility to (can be accessed by) a class that inherits MyObject?

a) only A
b) only B
c) both A and B
d) all of A, B, and C