CSCE 314
Programming Languages

Final Review Part II

Dr. Hyunyoung Lee
Abstract Data Types (ADTs)

- Object-oriented programming is rooted at ADTs

- ADTs
  - encapsulate a state with a set of operations
  - specify an interface of a data type (set of operation signatures) such that the underlying type representing the state is not directly accessible
  - allow multiple implementations of the same ADT

- We saw examples of ADTs in Haskell, built with the help of Haskell’s module construct

- Many language features can serve for implementing ADTs
Inheritance

- Inheritance in OO is based on the idea that ADTs have a lot in common

- Lends to a hierarchical structure of ADTs, for example: Arrays and lists are both sequences

- Inheritance enables hierarchical definition of ADTs

- Assume ADT B has substantially the same functionality as ADT A. If B is defined to inherit from A, it suffices to encode the difference between their functionalities.
Inheritance and Subtyping Relation

- Inheritance allows a (more) economical description of related ADTs
- A subclass extends a superclass.

```java
class SavingsAccount extends BankAccount {
    // new methods
    // new instance variables
}
```

- `extends` induces a subtyping relation:
  `SavingsAccount <: BankAccount`

- Contrast with inheriting from an interface: Here, subclass inherits behavior and state from a superclass
Inheritance Hierarchy

- **OO Ideal**
  - a set of related classes can be easily implemented by extending other classes via inheritance
  - everything stays nicely open-ended and extensible, in case new needs arise—just add another class
- **Example:** List, Stack, Queue, Dequeue, PriorityQueue
- The inheritance relation induced by “extends” in Java is rooted by **Object**
- Not all languages (e.g., C++) have a dedicated root class
OO Definitions

There are many definitions. At least,

\[\text{OOP} = \text{encapsulated state} + \text{inheritance (with dynamic binding)}\]

An object is an entity that
- has a unique identity
- encapsulates state

State can be accessed in a controlled way from outside by means of \textit{methods} that have direct access to state. State is also initialized and finalized in a controlled way.
Class

- Blueprint from which individual objects are created
- A unit of specification of objects in an incremental way
  - achieved by declaring inheritance from other classes and by encoding the difference to inherited classes, for example:
    
    \[
    \text{Bicycle \{cadence, speed, gear, ...\}}
    \]

Note: OO languages that do not have the notion of a class exist (e.g. JavaScript)
Fundamental Data Types

Primitive data types: boolean, char, byte, short, int, long, float, double

Each with corresponding “wrapper” class:
Example Class: Stack

```java
public class Stack {
    protected class Node {
        int data;
        Node next;
        Node (int v, Node n) { data = v; next = n; }
    }
    public Stack() { stk = null; }
    public boolean empty() { return stk == null; }
    public int pop() {
        int result = stk.data;
        stk = stk.next;
        return result;
    }
    public int top () { return stk.data; }
    public void push (int i) { stk = new Node (i, stk); }
    private Node stk; // state variable, properly encapsulated
}
```
Instantiating and Invoking Class Members

class StackMain {
    public static void main (String args[])
    {
        Stack s = new Stack();
        s.push(1);
        s.push(3);
        s.pop();
        System.out.println( Integer.toString( s.top() ) );
    }
}

Note: static/class methods vs. instance methods
Access Control

How programming language restricts access to members of objects or classes.

- **Java**: public, private, protected, and “package” (no modifier)
- **C++**: public, protected, private

The meaning of access control modifiers vary from one language to another

- e.g., whether attributes of another object of the same type is accessible or not
Interfaces

- Interface is like a class definition, except for no method bodies or instance variables. Example:

  ```java
  public interface IStack {
    public boolean empty();
    public int pop();
    public int top();
    public void push(int i);
  }
  ```

- We can now plug-in many different implementations for the same interface:

  ```java
  public class Stack implements IStack { ... }
  public class AnotherStack implements IStack { ... }
  ```
Interfaces as ADTs

interface gives an interface against which to write code that is oblivious of the implementation of the interface

Given the following classes:

```java
class Coin {
    public getValue() { ... }
    ... 
}
class File {
    public getSize() { ... }
    ... 
}
```

Task: Implement containers DataSet that keep track of the maximal and accumulated file sizes or values of coins...
class DataSet {
    ...
    public add(Coin x) {
        total = total + x.getValue();
        if (count == 0 ||
            max.getValue() < x.getValue())
            max = x;
        count++;
    }
    public Coin getMax() {
        return max;
    }
    private double total;
    private Coin max;
    private int count;
}

class DataSet {
    ...
    public add(File x) {
        total = total + x.getSize();
        if (count == 0 ||
            max.getSize() < x.getSize())
            max = x;
        count++;
    }
    public File getMax() {
        return max;
    }
    private double total;
    private File max;
    private int count;
}

public interface Measurable { double getMeasure(); }

class Coin implements Measurable {
    public getMeasure { return getValue(); }
    public getValue() { . . . }
    ...
}

class File implements Measurable {
    public getMeasure { return getSize(); }
    public getSize() { . . . }
    ...
}
public interface Measurable {  double getMeasure();  }

class Coin implements Measurable {
    public getMeasure {return getValue();}
    public getMeasure { . . . }
    . . .
}

class File implements Measurable {
    public getMeasure {return getSize();}
    public getSize() { . . . }
    . . .
}

class DataSet {
    . . .
    public add(Coin x) {
        total = total + x.getValue();
        if (count == 0 ||
            max.getValue() < x.getValue())
            max = x;
        count++;
    }
    public Coin getMax() {
        return max;
    }
    private double total;
    private Coin max;
    private int count;
}

class DataSet {
    . . .
    public add(Measurable x) {
        total = total + x.getMeasure();
        if (count == 0 ||
            max.getMeasure() < x.getMeasure())
            max = x;
        count++;
    }
    public Measurable getMax() {
        return max;
    }
    private double total;
    private Measurable max;
    private int count;
}
Substitutability via Subtyping

- We can use a `Coin` or a `File` where a `Measurable` is expected because of subtyping and substitutability.

- `class Coin implements Measurable` establishes that `Coin` is a subtype of `Measurable`.

- Symbolically, `Coin <: Measurable`.

- Substitutability: If `S <: T`, then any expression of type `S` can be used in any context that expects an expression of type `T`, and no type error will occur. As a type rule:

\[
\Gamma \vdash e : S \\
S <: T \\
\hline
\Gamma \vdash e : T
\]
Abstract Classes Compared to Interfaces

- Abstract classes can have fields that are not static and final, and public, protected, and private concrete methods
- With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default or static methods) are public
- A class can extend only one class, whether or not it is abstract, whereas it can implement any number of interfaces
- An interface can “extend” (but cannot “implement”) multiple interfaces
- An interface cannot be instantiated
- Example interfaces: Comparable, Cloneable, Serializable, etc.
(Mental) Model of Parametrized Classes

```java
public class Pair<T, U> {
    private T a; private U b;
    public Pair(T t, U u) { a = t; b = u; }
    public T getFirst() { return a; }
    public U getSecond() { return b; }
}

Pair<Integer, String> p = new Pair<Integer, String>(0, "");

public class Pair<Object, Object> {
    private Object a; private Object b;
    public Pair(Object t, Object u) { a = t; b = u; }
    public Object getFirst() { return a; }
    public Object getSecond() { return b; }
}
```
Parametric Polymorphism

Why does this work?

```java
public static <E> void print(E[] a) {
    for (E e : a) System.out.print(e.toString() + " ");
    System.out.println();
}
```

In Haskell, this did not work:

```haskell
print :: [a] -> IO ()
print ls = mapM_ (putStrLn . show) ls
```

But this did:

```haskell
print :: Show a => [a] -> IO ()
print ls = mapM_ (putStrLn . show) ls
```
Parametric Polymorphism (Cont.)

Java, too, needs constraints to type parameters

Without constraints, only operations that are supported for all types can be applied to values whose types are type parameters.

If no constraints, the constraint extends Object is assumed:

```java
public static <E extends Object> void print(E[] a) {
    for (E e : a) System.out.print(e.toString() + " ");
    System.out.println();
}
```

"E extends Object" justifies toString
Example of Constraints

Erroneous:  OK

\begin{verbatim}
public static <E extends Comparable<E>>
public static <E>
void print(List<E> a, E threshold) {
    for (E e : a)
        if (e.compareTo(threshold) < 0) // type error !!
            System.out.print(e.toString() + " ");
    System.out.println();
}
\end{verbatim}

\textbf{Comparoble} interface itself is really parametrized (to be discussed)
Unnamed Type Parameters - Wildcards

```java
static void printAll (List<?> l) {
    for (Object o : l) System.out.println(o);
}
```

Wildcards are both a convenience feature (more concise syntax), and to add support for co/contravariance for type parameters (discussed later).
F-Bounded Quantification

F-bounded quantification allows the type parameter being constrained to appear in its own bound: \(<T \text{ extends } A<T>>\)

The unsuccessful translate example:

\(<T \text{ extends } Movable> T \text{ translate}(T m) \{ \text{return } m.\text{move}(1, 1); \} \)

can now be written as:

interface Movable<T> { T move(int x, int y); }

\(<T \text{ extends } Movable<T>> T \text{ translate<T>>(T m) } \{ \text{return } m.\text{move}(1, 1); \} \)
Type System and Variance

Within the type system of a programming language, variance refers to how subtyping between complex types (e.g., list of Cats versus list of Animals, and function returning Cat versus function returning Animal) relates to subtyping between their components (e.g., Cats and Animals).
Covariance and Contravariance

Within the type system of a programming language, a typing rule or a type constructor is:

- **covariant** if it preserves the ordering $\leq$ of types, which orders types from more specific to more generic
- **contravariant** if it reverses this ordering
- **invariant** if neither of these applies.
Covariant Method Return Type

Return types of methods in Java can be covariant:

```java
class Animal {
    . . .
    public Animal clone() { return new Animal(); } }
class Panda extends Animal {
    . . .
    public Panda clone() { return new Panda(); } }
```

This is safe – whenever we call Animal’s clone(), we get at least an Animal, but possibly something more (a subtype of Animal)

```java
Animal a = new Panda();
. . .
Animal b = a.clone(); // returns a Panda, OK
```
Covariant Method Argument Type (Bad Idea)

Would this be a good idea?

class Animal {
    ... 
    public bool compare(Animal) { ... }
}
class Panda extends Animal {
    ... 
    public bool compare(Panda) { ... }
}

Covariant argument types are not type safe, and thus not supported in (most) mainstream languages including Java:

Animal a = new Panda();
Animal b = new Animal();
...
a.compare(b); // type error at run time
Contravariant Argument Types?

Contravariant argument types would be safe, but they have not been found useful and hence not supported in practical languages (not in C++ nor in Java).

```java
class Animal {
    public bool compare(Panda) {
        ...
    }
}
class Panda extends Animal {
    public bool compare(Animal) {
        ...
    }
}
```

Summary: When overriding methods in Java - invariant argument type, covariant return type.

Co- and contravariance of parameter and argument types are best captured with this subtyping rules between function types:

\[
\begin{align*}
U_1 & \ll U_2 & T_2 & \ll T_1 \\
U_2 & \rightarrow T_2 & U_1 & \rightarrow T_1
\end{align*}
\]
Co- and Contravariance and Generic Types

The same phenomenon occurs with generic definitions.

Subtyping of generic types:
Assume \( B<T> <: A<T> \) and \( \text{Integer} <: \text{Number} \)
Which of these are true (safe)?

\[
\begin{align*}
B<\text{Integer}> & <: A<\text{Integer}> \\
B<\text{Integer}> & <: B<\text{Number}> \\
B<\text{Integer}> & <: A<\text{Number}>
\end{align*}
\]

Only the first (pointwise subtyping)

Other two are forms of covariance, and unsafe (unless restricted suitably)
Example: Java Arrays

Object[] o = new String[]{"1", "2", "3"};  // OK
o[0] = new Integer(1);  // throws here
First statement OK. Java arrays are (unsafely) covariant:
String <: Object ⇒ String[] <: Object[]
You can think of String[] as Array<String>
From the type checker’s perspective, second OK

Integer <: Object

However, the second statement is an attempt to assign an Integer in place of String.

Full functionality of arrays does not safely allow co/contravariance.
   Reasons:
1. Covariant arrays would be safe for reading elements from them.
2. Contravariant arrays would be safe for writing elements into them.
3. Java arrays support both reading and writing.


Wildcards Bound From Above

Example: List<Line> and List<Shape>

```java
public class Canvas {
    public void draw(List<Shape> shapes) {
        for (Shape s: shapes) s.drawOn(this);
    }
}
```

```java
List<Line> ll; . . . ; canvas.draw(ll); // error
```

// Now with wildcards

```java
public class Canvas {
    public void draw(List<? extends Shape> shapes) {
        for (Shape s: shapes) s.drawOn(this);
    }
}
```

```java
List<Line> ll; . . . ; canvas.draw(ll); // OK
```

// ? reads “unknown”, corresponds to a fresh type variable

// Shape is the upper bound of the wildcard
Wildcards Bound From Below

Contravariance expressed with wildcards:

```java
<T> void copy(List<T> src, List<? super T> dest)
    { ... dest.add(src.get(i)) ... }
```

List<Object> lo;
List<String> ls;
copy(ls, lo); // ok
copy(lo, ls); // not ok

Two alternative formulations of copy:

```java
<T> void copy(List<? extends T> src, List<T> dest);
<T, S extends T> void copy(List<S> src, List<T> dest);
```

The “fresh type variable” model describes wildcards to some extent, but does not apply in all contexts.
Type Erasure

1. Backwards compatibility was a hard constraint for design

2. Generic Java programs are compiled to class files that run on unchanged JVM.

3.Generic types are used during type checking, then erased according to a set of translation rules:
   a. Type parameters are replaced with their bounds
   b. Parametrized types throw away the parameters
   c. Casts inserted where necessary

[Note: Compared to time prior generics, run-time casts still prevail (no improvement on performance) but they’ll never throw an exception.]
Threads

- Thread is an independently executed unit of a program.
- The JVM takes care of scheduling threads, typically each active thread gets a small amount of processing time in its turn, with rapid switching between threads.
- In other words: Programmer does not control how much of which thread gets executed when (preemptive scheduling).
- In a system with more than one processing units, threads may execute in parallel.
Threads vs. Processes

Process
1. self-contained execution environment
2. own memory space
3. one Java application, one process (not always true)

Thread
1. at least one per process
2. shares resources with other threads in the process, including memory, open files
3. every (Java) program starts with one thread (+ some system threads for GC etc.)
4. concurrency is attained by starting new threads from the main thread (recursively)
public interface Runnable {
    void run();
}

public class MyRunnable implements Runnable {
    public void run() {
        // task here . . .
    }
}

Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
Causal Order

Concurrent program:

- All execution states of a given thread are totally ordered
- Execution states of the concurrent program as a whole are partially ordered
Extending Thread

Task for a thread can be specified also in a subclass of Thread

```java
public class MyThread extends Thread {
    public void run() {
        // task here
    }
}
Thread t = new MyThread();
t.start();
```

Benefits of using Runnable instead:

- It does not identify a task (that can be executed in parallel) with a thread object
- Since Runnable is an interface, the class implements Runnable could extend another class
- Thread object typically bound with the OS’s thread
- Many runnables can be executed in a single thread for better efficiency, e.g., with thread pools
Thread States

A thread can be in one of the following states:

• new: just created, not yet started
• runnable: after invoking `start()`. Not scheduled to run yet
• running: executing
• blocked: waiting for a resource, sleeping for some set period of time. When condition met, returns back to runnable state
• dead: after return of `run` method. Cannot be restarted.
Thread t = new Thread()

t.start()

return of run() method

condition is met

waiting for monitor lock

Object.wait with no timeout

Thread.sleep()

New

Runnable

Blocking

Waiting

Timed_waiting

Terminated
Race Condition Example
(BankAccount.java)

- **Object** `account` is shared among several threads

- First thread reads account’s balance; second thread preempts, and updates the balance; first thread updates the balance as well, but based on incorrect old value.

- deposit and withdraw methods’ postconditions are not guaranteed to hold (what are their postconditions?)

```
public void deposit(double amount) {
    balance = balance + amount;
    ...  // Other code...
}
```
Race Condition Example (Cont.)

- Removing the long sleeps will not help
- Pre-empting occurs at byte/native code level, and does not respect Java’s expression/statement boundaries
- Note: Local variables, function parameters, return values stored in thread’s own stack, only have to worry about instance variables and objects on the heap
Synchronization With Locks

- Lock object guards a shared resource
- Commonly a lock object is an instance variable of a class that needs to modify a shared resource:

```java
public class BankAccount {
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();
        ...;
    }
    ...
    private Lock balanceChangeLock;
}
```
Synchronization With Locks (Cont.)

Code manipulating the shared resource guarded with a lock

```java
public class BankAccount {
    public BankAccount() {
        balanceChangeLock = new ReentrantLock(); ... 
    }
    private Lock balanceChangeLock;
}

balanceChangeLock.lock();
// manipulate balance here
balanceChangeLock.unlock();
```

Better:
```java
try {
    // manipulate balance here
} finally {
    balanceChangeLock.unlock();
}
```
Example

```java
public void deposit(double amount) {
    balanceChangeLock.lock()
    try {
        System.out.println("Depositing " + amount);
        double nb = balance + amount;
        System.out.println("New balance is " + nb);
        balance = nb;
    } finally {
        balanceChangeLock.unlock();
    }
}
```

Above could be improved – critical sections should be as short as possible.
Lock Ownership

• Thread owns the lock after calling `lock()`, if another thread does not own it already

• If lock owned by another thread, scheduler deactivates thread that tries to lock, and reactivates periodically to see if lock not owned anymore

• Ownership lasts until `unlock()` called

• “Reentrant” lock means the thread owning a lock can lock again (e.g., calling another method using the same lock to protect its critical section)
Per Method Synchronization

- Java ties locks and synchronization: **object locks and synchronized methods**
- The granularity may not always be desirable. Example:
  ```java
  public class BankAccount {
      public synchronized void deposit(double amount) {
          System.out.println(" Depositing “ + amount);
          double nb = balance + amount;
          System.out.println("New balance is “ + nb);
          balance = nb;
      }
      public synchronized void withdraw(double amount) { . . . }
  }
  - Synchronized methods automatically wraps the body into 
    lock; try {body} finally {unlock }
Deadlock

- Our bank account allows overdraft. Attempts to remedy:
  
  ```java
  if (account.getBalance() >= amount) account.withdraw(amount);
  ```

- Does not work, thread may be preempted between test of balance and withdrawing

- Next attempt
  
  ```java
  public void withdraw(double amount) {
      balanceChangeLock.lock();
      try {
          while (balance < amount) {} // wait balance to grow
          double nb = balance - amount;
          balance = nb;
      } finally {
          balanceChangeLock.unlock();
      }
  }
  ```

Waiting, however, prevents other threads of updating balance.
Deadlock

A situation where two or more processes are unable to proceed because each is waiting for one of the others to do something.

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.” -- Statute passed by the Kansas (USA) State Legislature, early in the 20th century
Condition Objects

**Condition object** allows a temporary release of a lock

```java
public class BankAccount {
    public BankAccount() {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition();
    }
    public void withdraw(double amount) {
        balanceChangeLock.lock();
        try {
            while (balance < amount) sufficientFundsCondition.await();
            ...
        } finally {
            balanceChangeLock.unlock();
        }
    }
    private Lock balanceChangeLock;
    private Condition sufficientFundsCondition;
}
```

Current thread unblocked by a call to `signalAll()`, a notification to all threads blocked with a particular condition object
public void deposit(double amount) {
    balanceChangeLock.lock();
    try {
        . . .
        sufficientFundsCondition.signalAll();
    } finally {
        balanceChangeLock.unlock();
    }
}

- Notification with `signalAll` means: something has changed, it is worthwhile to check if it can proceed
- `signalAll` must be called while owning the lock bound to the condition object
Example: Synchronized Counter

```java
public class SynchronizedCounter {
    public final class Counter {
        public synchronized int getValue() { return value; }
        public synchronized void increment() { ++value; }
        private int value = 0;
    }
}
```

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

1. It is not possible for two invocations of synchronized methods on the same object to interleave.

2. When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.
Obtaining a class object

The entry point to reflection operations: java.lang.Class
Various means to obtain an object of Class type
1. From instance (using Object.getClass()):
   MyClass mc = new MyClass();
   Class c = mc.getClass();
2. From type:
   c = boolean.class;
3. By name:
   Class cls = null;
   try {
       cls = Class.forName("MyClass");
   } catch (ClassNotFoundException e) {
       . . .
   }
   // use cls
Obtaining other classes from a Class object

- `Class.getSuperclass()`
- `Class.getClasses()`
  - returns all public class, interface, enum members as an array. Includes inherited members. Example:
    ```java
    Class<?>[] c = Character.class.getClasses();
    ```
- `Class.DeclaredClasses()`
  - similar to `getClasses()` but includes non-public classes and does not recur to base classes
- `Class.getEnclosingClass()`
  - immediately enclosing class
Class Modifiers

Java’s class modifiers

- public, protected, and private
- abstract
- static
- final
- strictfp
- Annotations

Example:

class public static MyClass { . . . }

. . .

int mods = MyClass.class.getModifiers();
if (Modifier.isPublic(mods))
    System.out.println("public class");
Accessing Members

- Same set of methods for fields, methods, and constructors.
  Let X be one of Field, Method, or Constructor:
  
  ```java
  X Class.getDeclaredX(String)
  X Class.getX(String)
  X[] Class.getDeclaredXs()
  X[] Class.getXs()
  ```

- "Declared" versions obtain private members too
- "non-Declared" versions obtain inherited members
- Method and constructor versions need the parameter types as parameters too:
  ```java
  Method getDeclaredMethod(String name,
  Class<?> . . . parameterTypes)
  throws NoSuchMethodException, SecurityException
  ```
Reflection Example

```java
Class c = Class.forName(className);
System.out.println(c + " {");
int mods;
Field fields[] = c.getDeclaredFields();
for (Field f : fields) {
    if (!Modifier.isPrivate(f.getModifiers()) &&
        !Modifier.isProtected(f.getModifiers()))
        System.out.println("\t" + f);
}
Constructor[] constructors = c.getConstructors();
for (Constructor con : constructors) { System.out.println("\t" + con); }
Method methods[] = c.getDeclaredMethods();
for (Method m : methods) {
    if (!Modifier.isPrivate(m.getModifiers())) { System.out.println("\t" + m); }
}
System.out.println("}"");
```
Java Virtual Machine and Java

- The Java Virtual Machine (JVM) is a stack-based abstract computing machine.
- JVM was designed to support Java -- Some concepts and vocabulary from Java carry to JVM
- A Java program is a collection of class definitions written in Java
- A Java compiler translates a class definition into a format JVM understands: class file.
- A class file contains JVM instructions (or bytecodes) and a symbol table, and some other information. When a JVM reads and executes these instructions, the effect is what the original Java program called for: the class file has the same semantics as the original Java program.
The JVM specification defines:

1. A set of instructions and a definition of the meanings of those instructions called bytecodes.

2. A binary format called the class file format, used to convey bytecodes and related class infrastructures in a platform-independent manner.

3. An algorithm for identifying programs that cannot compromise the integrity of the JVM. This algorithm is called verification.
Representation of Memory

Typical CPU instruction set views memory as array of bytes

- **Construct object:** allocate contiguous sequence of bytes
- **Access a field:** access bytes at a specific offset
- **Call a function:** jump to a location in memory where function resides

JVM allows no byte-level access

- **Direct operations for allocating objects, invoking methods, accessing fields**
The Main Loop of a JVM Interpreter

do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
Class Loaders

• Data in class file format do not have to be stored in a file. They can be stored in a database, across the network, as part of Java archive file (JAR), or in variety of other ways.

• Essential component of using class files is the class ClassLoader, part of the Java platform. Many different subclasses of ClassLoaders are available, which load from databases, across the network, from JAR files, and so on. Java-supporting web browsers have a subclass of ClassLoader that can load class file over the Internet.

• If you store your information in some nonstandard format (such as compressed) or in a nonstandard place (such as a database), you can write your own subclass of ClassLoader.
The Verifier

• To ensure that certain parts of the machine are kept safe from tampering, the JVM has a verification algorithm to check every class.

• Programs can try to subvert the security of the JVM in a variety of ways:
  • They might try to overflow the stack, hoping to corrupt memory they are not allowed to access.
  • They might try to cast an object inappropriately, hoping to obtain pointers to forbidden memory.
  • The verification algorithm ensures that this does not happen by tracing through the code to check that objects are always used according to their proper types.
Internal Architecture of JVM
Runtime Data Areas in JVM

- Method area: contains class information, code and constants
- Heap: memory for class instances and arrays. This is where objects live.
- Java stack (JVM stack): stores “activation records” or “stack frames” - a chunk of computer memory that contains the data needed for the activation of a routine
- PC registers - program counters for each thread
- Native method stacks
Runtime Data Areas

Method Area
- Contains class information
- One for each JVM instance
- Shared by all threads in JVM
- One thread access at a time

Heap
- Contains class instance or array (objects)
- One for each JVM instance
- Facilitates garbage collection
- Expands and contracts as program progresses
Objects Representation in Heap

- An object reference
- Pointer into heap
- Pointer to class data
- Instance data
- Instance data
- Instance data
- Instance data

The heap

The method area

Class data
Runtime Data Areas: JVM Stack

- Each thread creates separate JVM stack
- Contains frames: current thread’s state
- Pushing and popping of frames
Stack Frame

- Local Variables
  - Organized in an array
  - Accessed via array indices
- Operand Stack
  - Organized in an array
  - Accessed via pushing and popping
  - Always on top of the stack frame
  - Work space for operations
- Frame Data
  - Constant Pool Resolution: Dynamic Binding
  - Normal Method Return
  - No exception thrown
  - Returns a value to the previous frame
public class BankAccount {
    private double balance;
    public static int totalAccounts = 0;
    public BankAccount() {
        balance = 0;
        totalAccounts++;
    }
    public void deposit( double amount ) { balance += amount; }
}

public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}