OOP Design Principles

EECS 3311

Ilir Dema

demailir@eecs.yorku.ca

Overview of Last Lectures

- **Code-and-fix model**
  - write some code, debug it, repeat until finished
- **Waterfall model**
  - a relatively linear sequential approach
- **Iterative Models**
  - **Prototype model**
    - used to understand/evolve the requirements
  - **Spiral model**
    - a risk-driven model, assesses risks at each step, and does the most critical action immediately
- **Agile model**
  - **Extreme Programming (XP)**
  - **Scrum**

  focus on process adaptability and customer satisfaction, a good fit in fast changing environments

Outlines

- **Overview of Java**
  - Java Characteristics
  - Process of Run Java Code
  - Java Class and API
  - Primitive Types
  - Java Operators
  - Access modifiers
  - Java Utils
- **OOP Design Principals**
  - Abstraction
  - Generics
  - Encapsulation
  - Inheritance
  - Polymorphism
- **Junit**
  - Strategies to write good Junit test cases

Overview of Java

References

- https://docs.oracle.com/javase/tutorial/
  - Official Java docs and tutorial

- https://beginnersbook.com/java-tutorial-for-beginners-with-examples/
  - Contains many useful Java example code snippets

Java Characteristics

- Java is ***platform independent:*** the same program can run on any correctly implemented Java system

- Java is ***object-oriented:***
  - Structured in terms of ***classes***, which group data with operations on that data
  - Can construct new classes by ***extending*** existing ones

- Java designed as
  - A ***core language*** plus
  - A rich collection of ***commonly available packages***

- Java can be embedded in Web pages
  - java applet

Java Processing and Execution

- Begin with Java ***source code*** in text files: **Student.java**

- A Java source code compiler produces Java ***byte code***
  - Outputs one file per class: **Student.class**
  - May be standalone or part of an IDE

- A ***Java Virtual Machine*** loads and executes class files
  - May compile them to native code (e.g., x86) internally

Compiling and Executing a Java Program

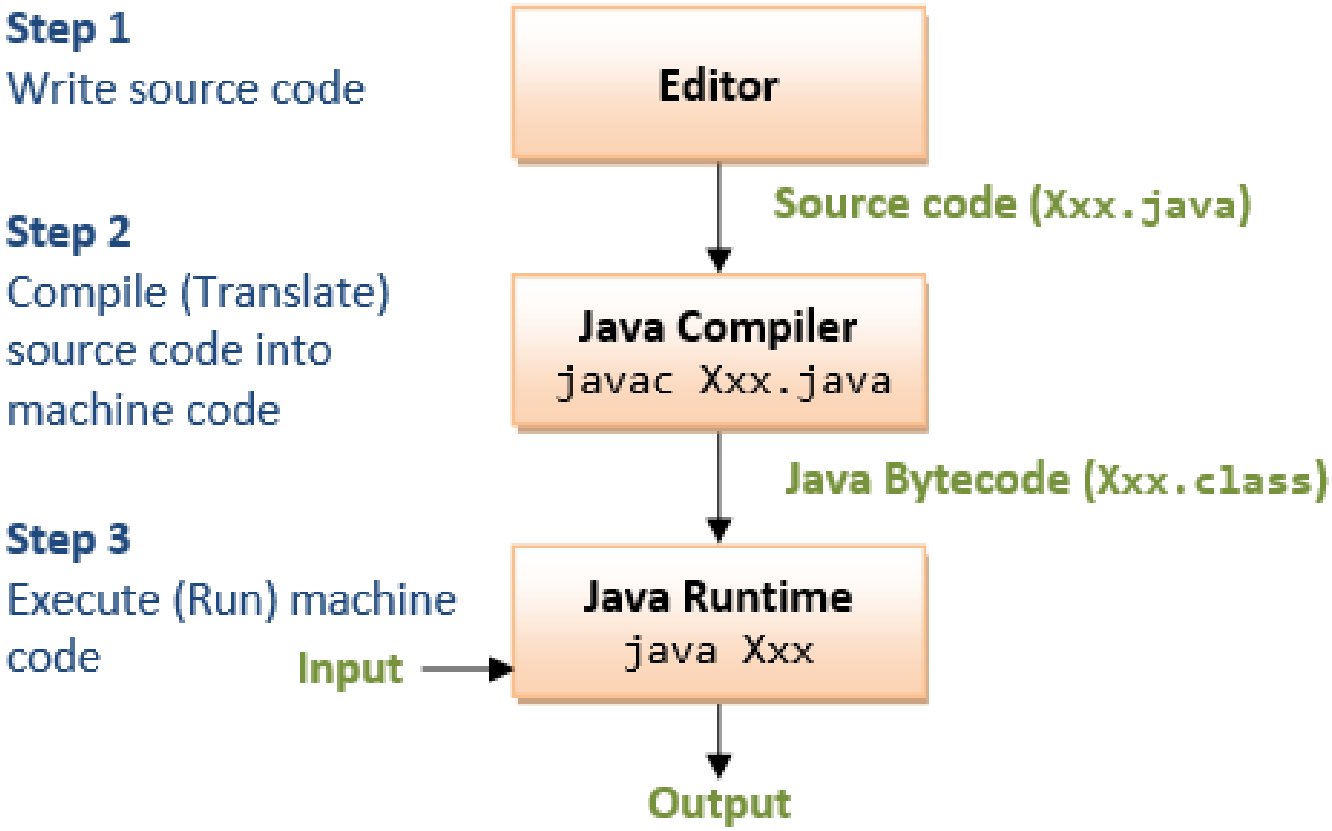
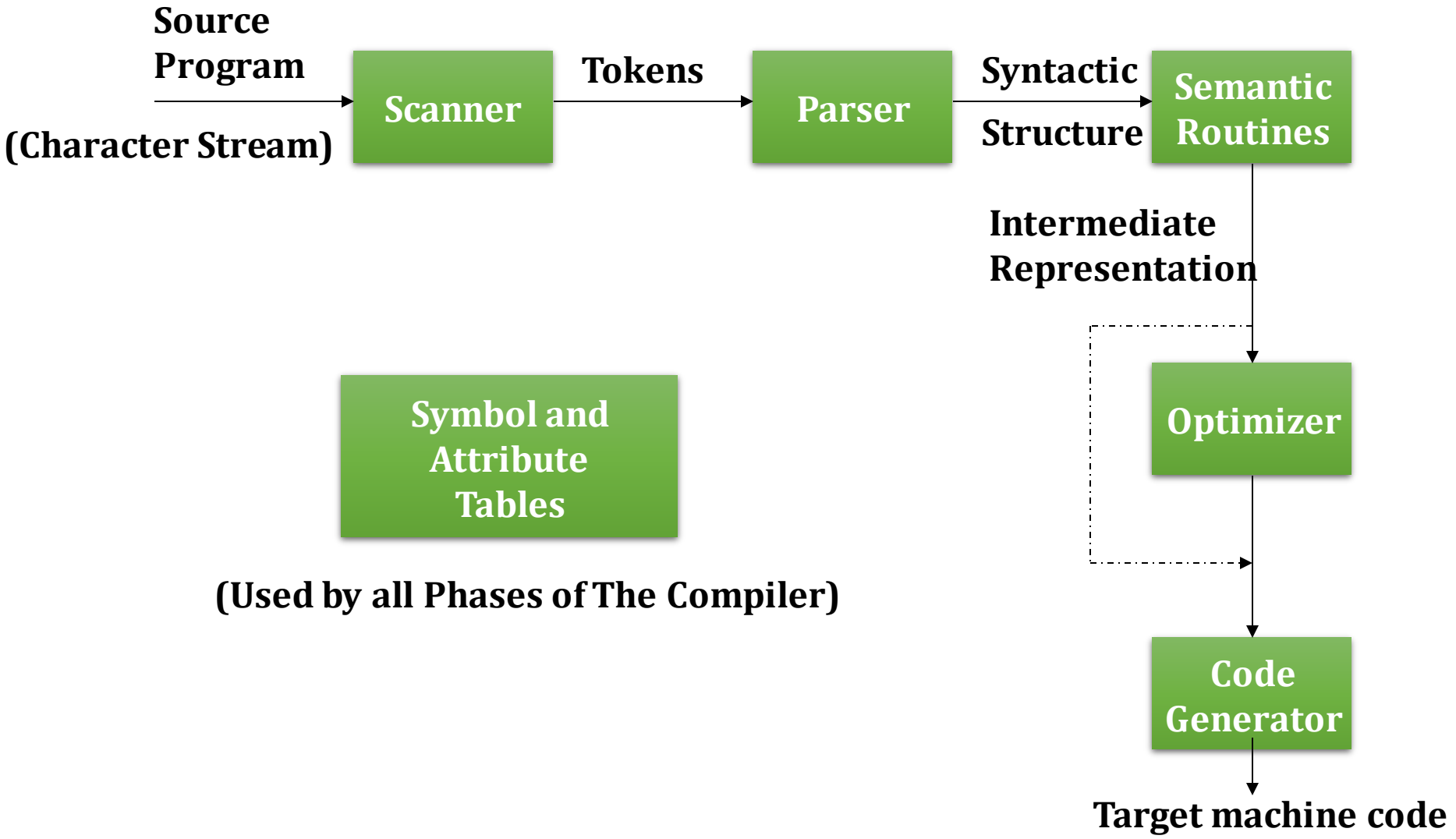
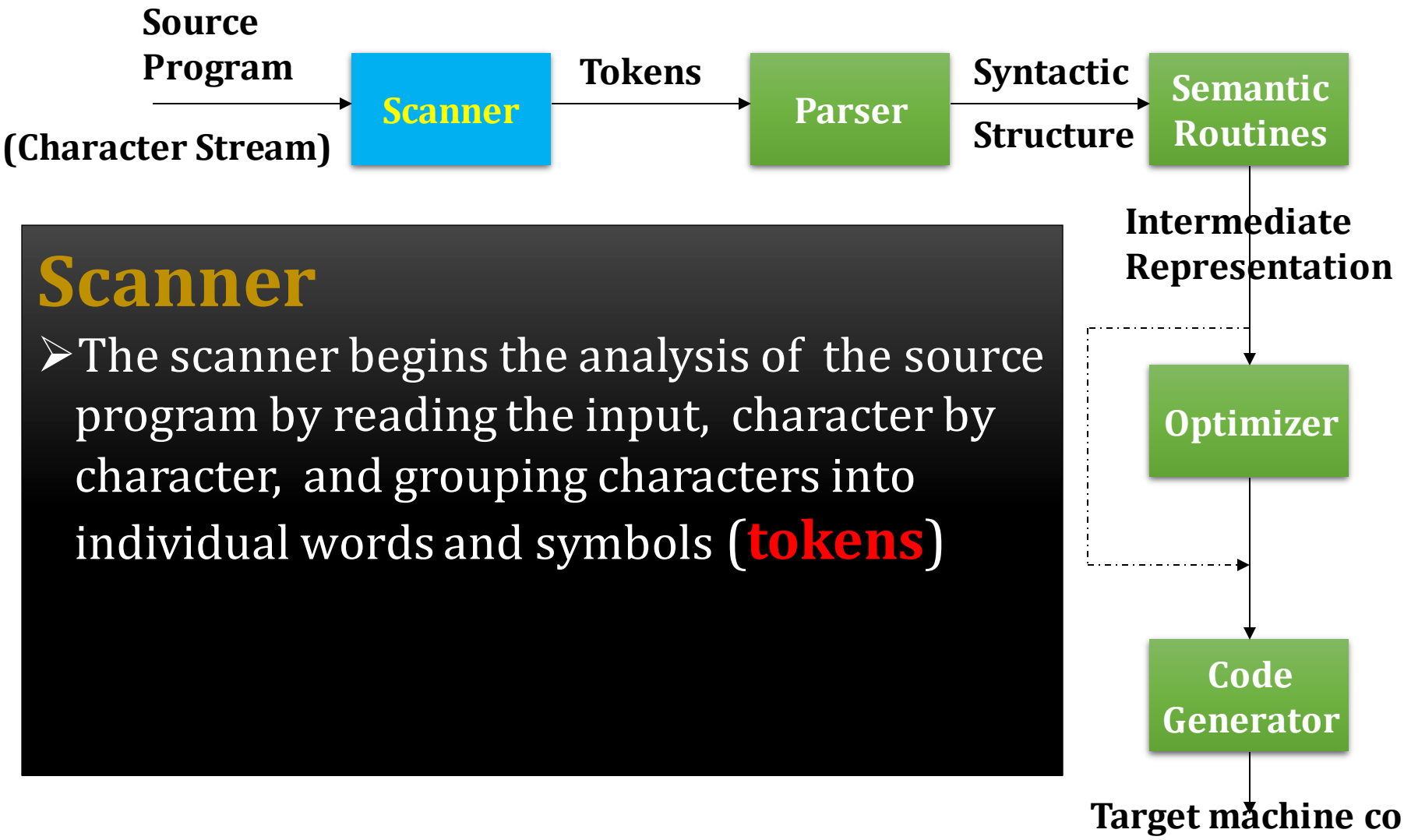


Figure source: http://exploreyourskill.com/wp-content/uploads/2019/11/ProgrammingSteps-79955.png

The Structure of a Compiler



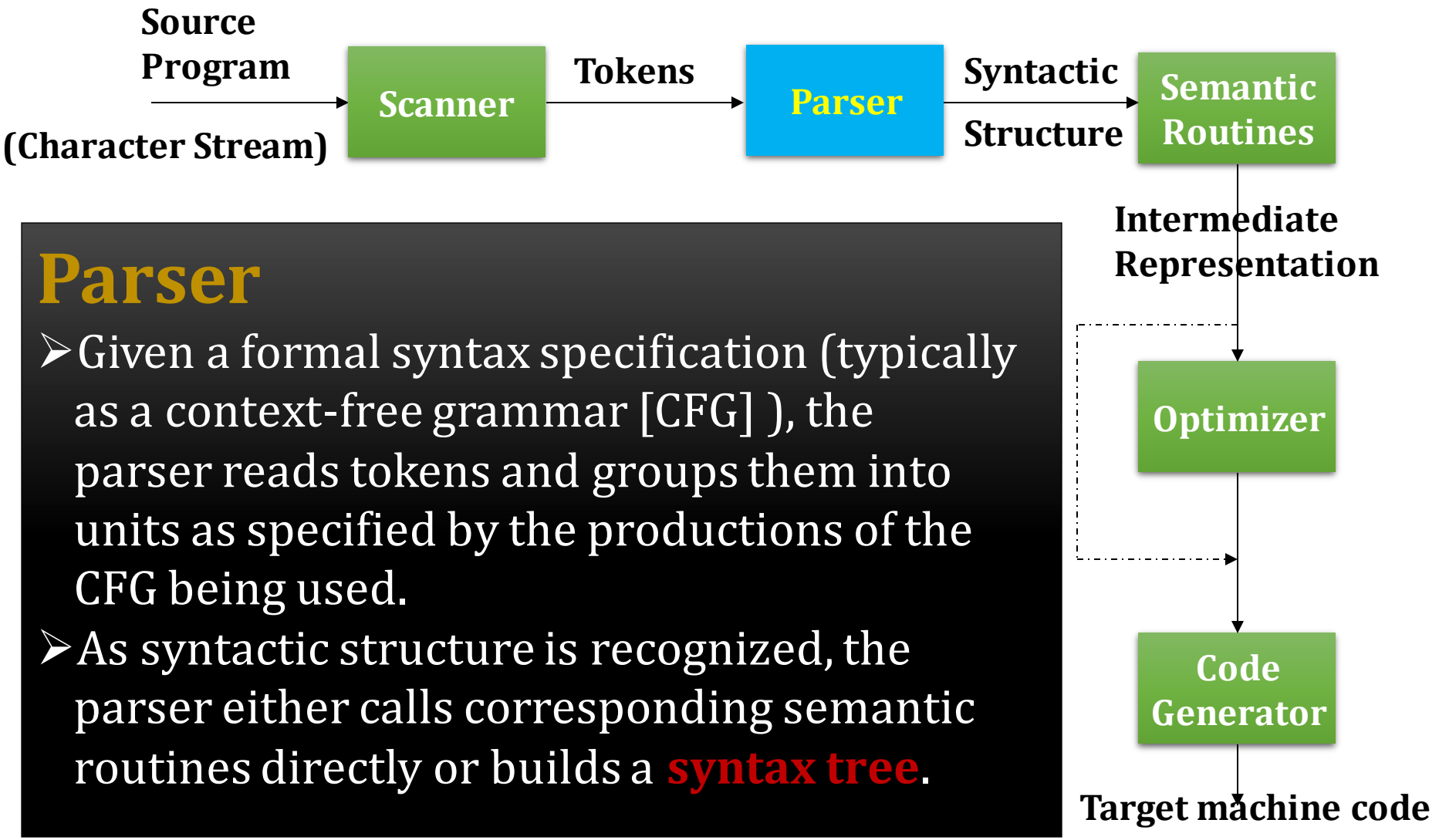
The Structure of a Compiler



Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (**tokens**)

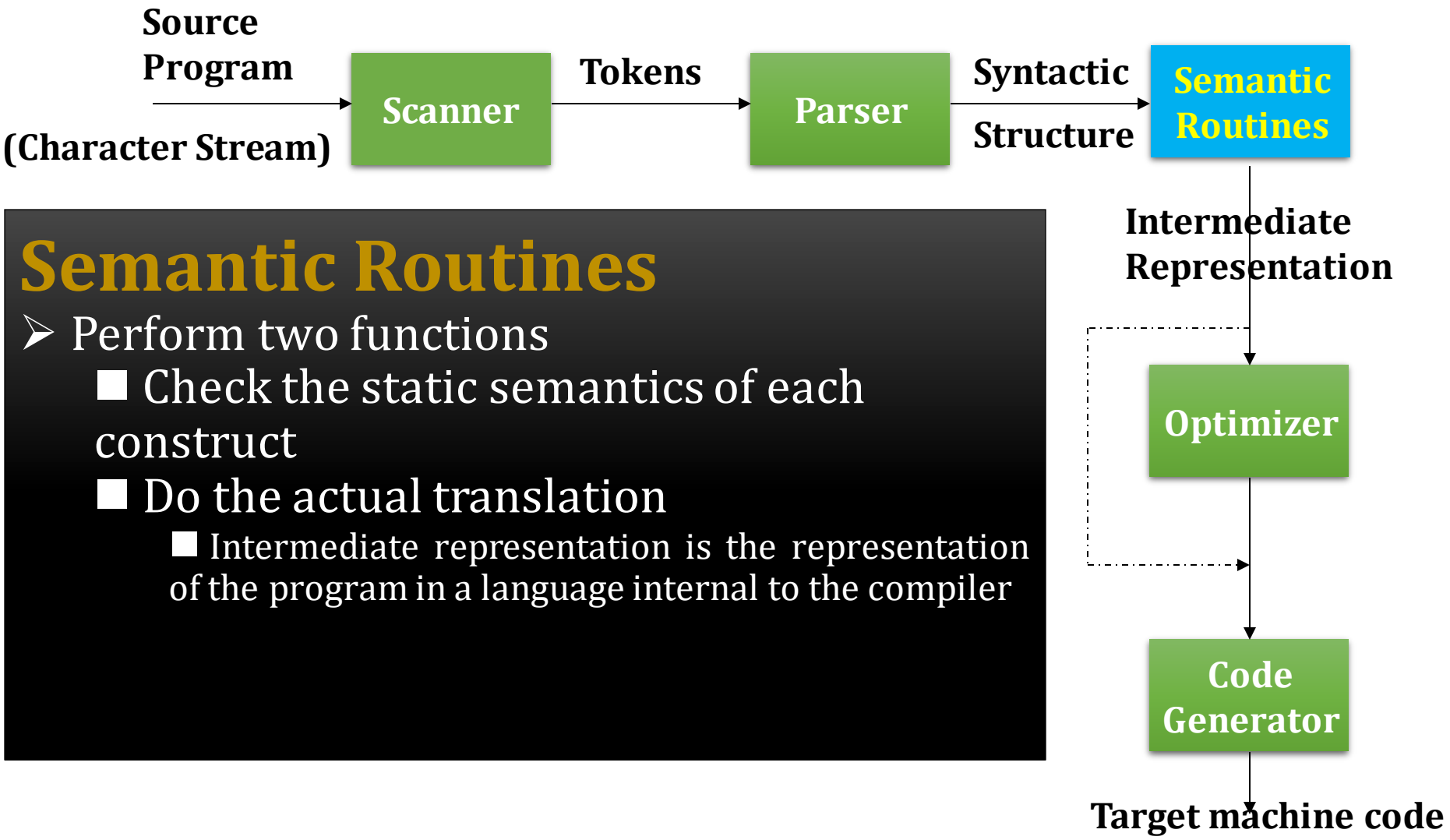
The Structure of a Compiler



Parser

- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parser reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a **syntax tree**.

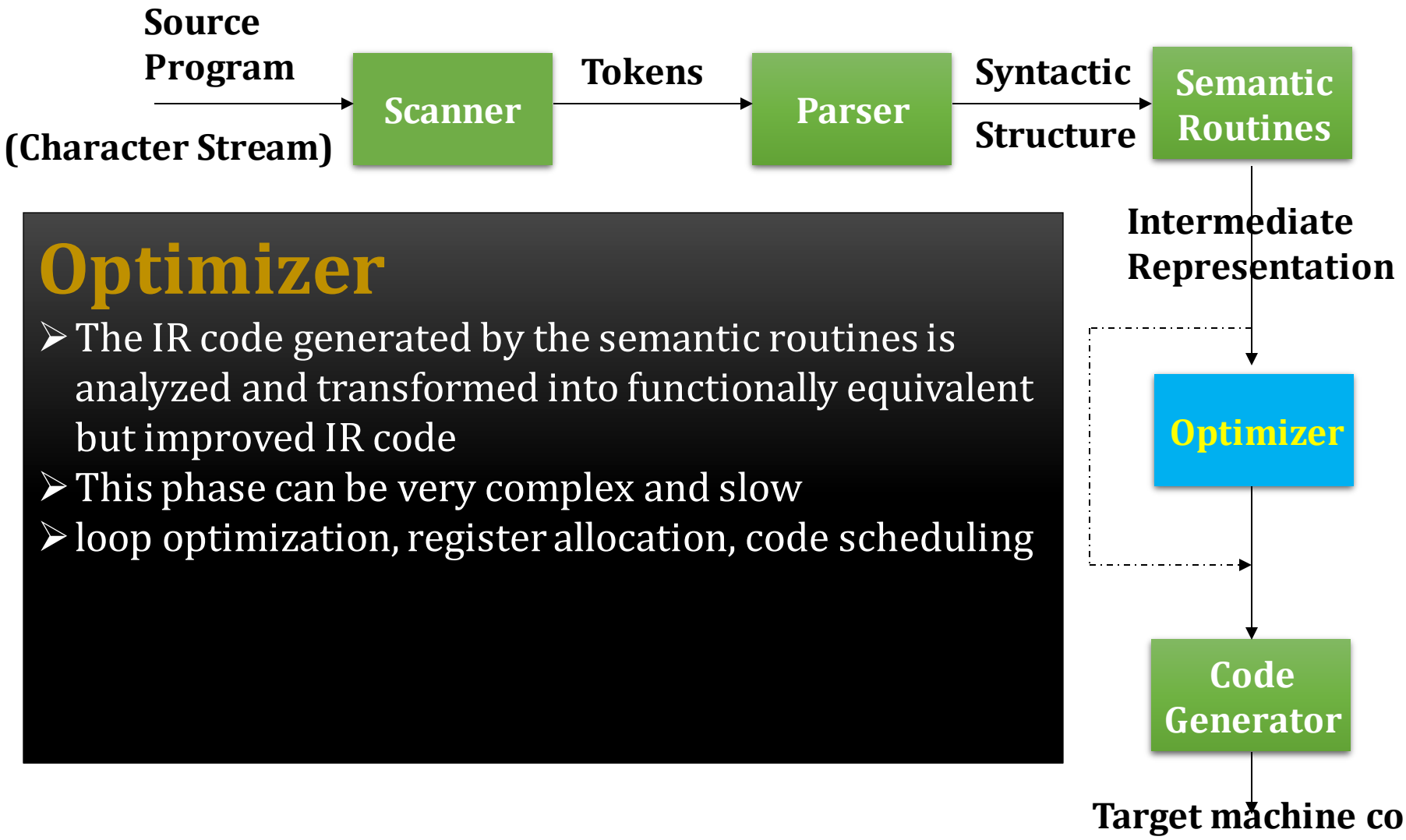
The Structure of a Compiler



Semantic Routines

- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
    - Intermediate representation is the representation of the program in a language internal to the compiler

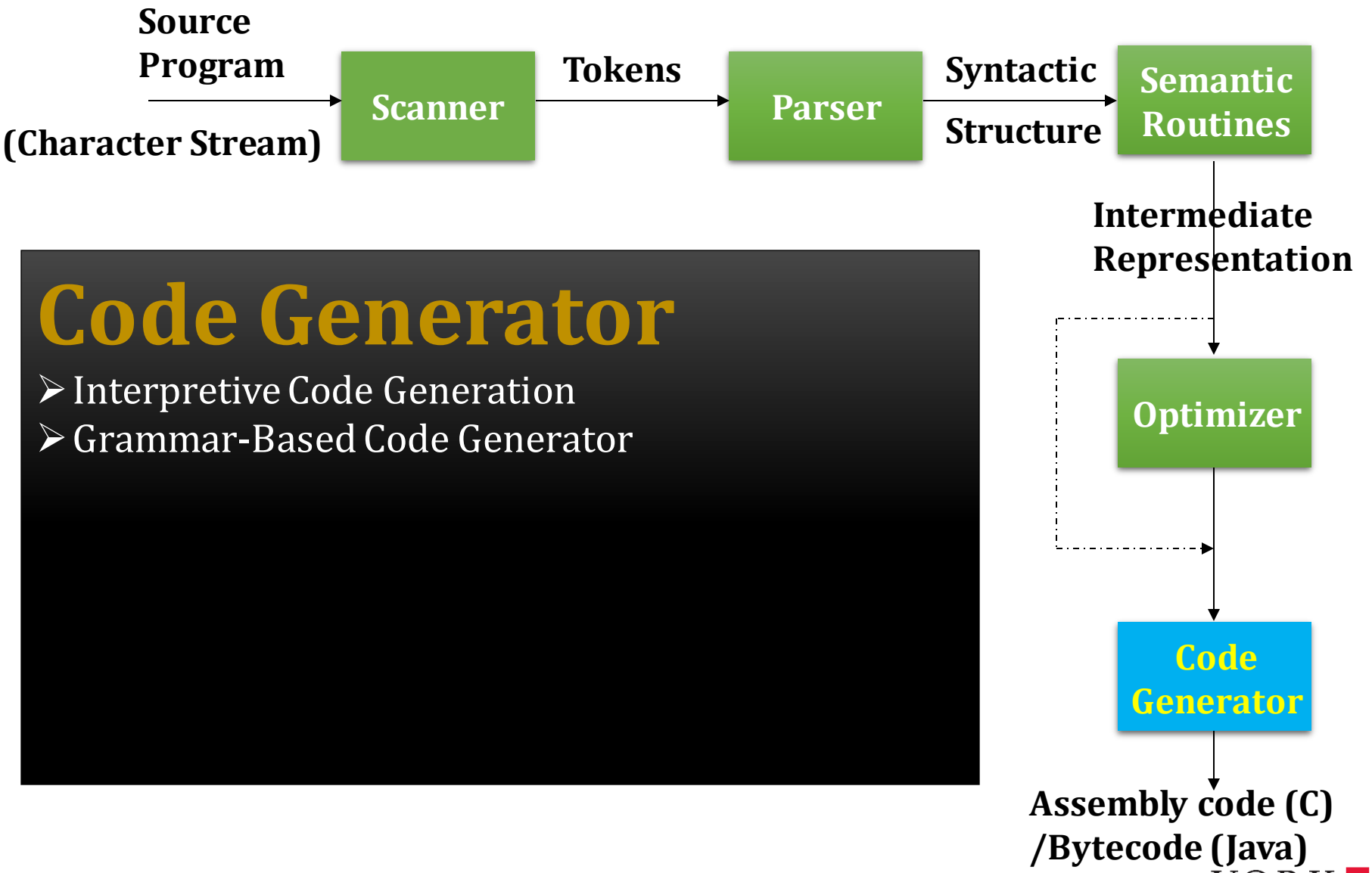
The Structure of a Compiler



Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- loop optimization, register allocation, code scheduling

The Structure of a Compiler



Code Generator

- Interpretive Code Generation
- Grammar-Based Code Generator

The Structure of a Compiler

position = initial + rate * 60

The Structure of a Compiler

position = initial + rate * 60

Symbol Table

position	id1
initial	id2
rate	id3

The Structure of a Compiler

position = initial + rate * 60

Scanner
[Lexical Analyzer]

Tokens

id1 := id2 + id3 * 60

Symbol Table

position	id1
initial	id2
rate	id3

The Structure of a Compiler

position = initial + rate * 60

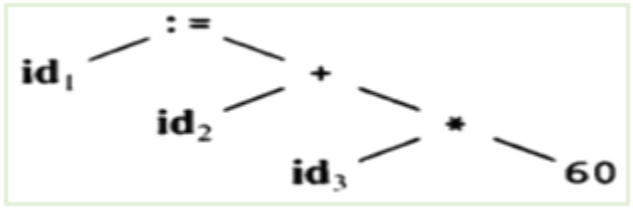
Scanner
[Lexical Analyzer]

Tokens

id1 = id2 + id3 * 60

Parser
[Syntax Analyzer]

Parse tree



Symbol Table

position	id1
initial	id2
rate	id3

The Structure of a Compiler

position = initial + rate * 60

⇩

Scanner
[Lexical Analyzer]

Tokens

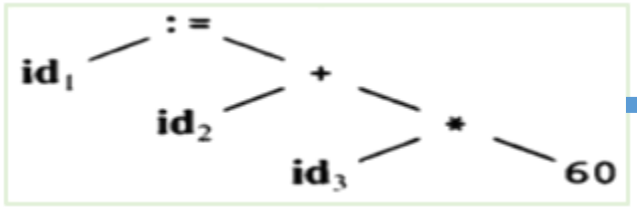
⇩

id1 = id2 + id3 * 60

⇩

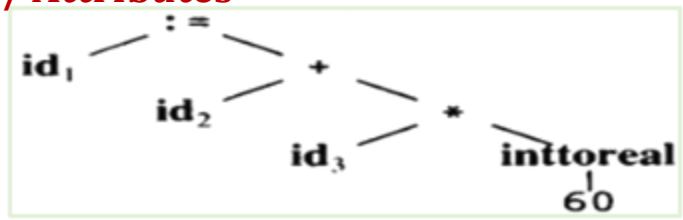
Parser
[Syntax Analyzer]

Parse tree



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree
w/ Attributes



YORK UNIVERSITÉ UNIVERSITY

The Structure of a Compiler

position = initial + rate * 60

⇩

Scanner
[Lexical Analyzer]

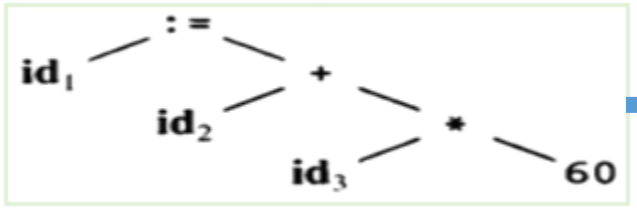
Tokens ⇩

id1 = id2 + id3 * 60

⇩

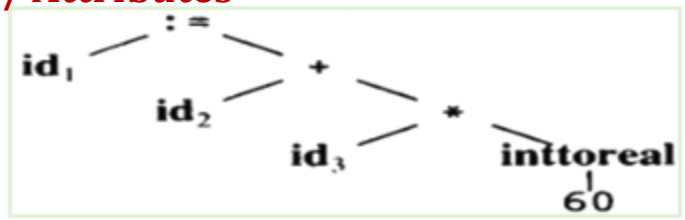
Parser
[Syntax Analyzer]

Parse tree ⇩



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes ⇩



⇩

Code Optimizer

⇩ **Optimized Intermediate Code**

temp1 = id3 * 60
Id1 = id2 + temp1

YORK UNIVERSITÉ UNIVERSITY

The Structure of a Compiler

position = initial + rate * 60

⇩

Scanner
[Lexical Analyzer]

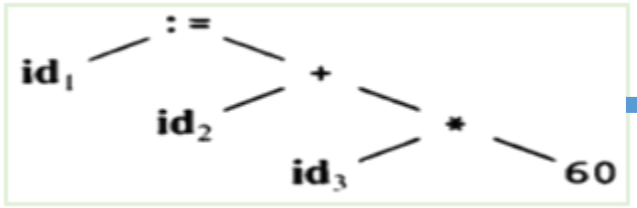
Tokens ⇩

id1 = id2 + id3 * 60

⇩

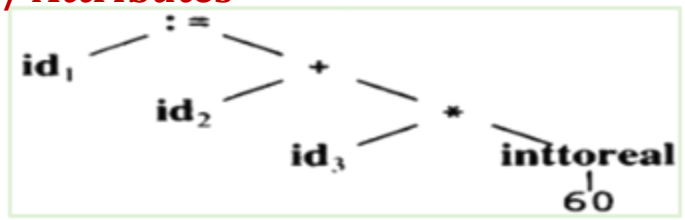
Parser
[Syntax Analyzer]

Parse tree ⇩



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes ⇩



⇩

Code Optimizer

Optimized Intermediate Code ⇩

temp1 = id3 * 60
Id1 = id2 + temp1

⇩

Code Optimizer

Target assembly code ⇩

```
MOVF    id3, R2
MULF    #60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Source code vs Bytecode

```
public class Test_Null {


  public static void main(String[] var0) {
     Object var1 = null;
     System.out.println(((String)var1).toLowerCase());
  }


}
```

Source code vs Bytecode

The corresponding Test_Null.class

```
public static main(java.lang.String[] arg0) { //([Ljava/lang/String;)V
    L1 {
      aconst_null
      astore1
    }
    L2 {
      getstatic java/lang/System.out:java.io.PrintStream
      aload1
      invokevirtual java/lang/String.toLowerCase()Ljava/lang/String;
      invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
    }
    L3 {
      return
    }
```

Classes and Objects

- The ***class*** is the unit of programming

- A Java program is a ***collection of classes***
  - Each class definition (usually) in its own **.java** file
  - *The file name must match the class name*

- A class describes ***objects (instances)***
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics

- These characteristics are:
  - ***Data fields*** for each object
  - ***Methods*** (operations) that do work on the objects

The "Object" Class

- **Object** is the *superclass* of all Java classes

- The Object class provides some common behaviors to all the objects such as object can be *compared*, object can be *cloned*, object can be *notified* etc.

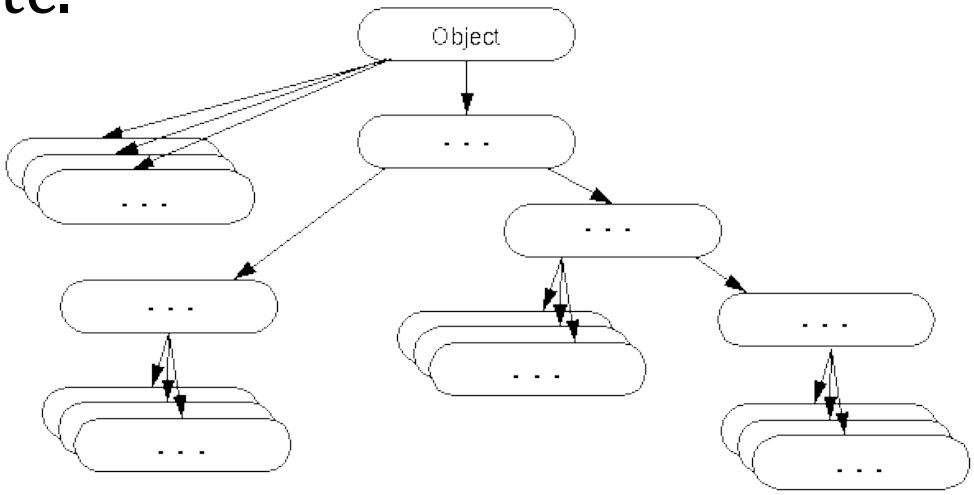


Figure source: https://www.javatpoint.com/images/core/objectclass.gif

Grouping Classes: The Java API

- **API** = *Application Programming Interface*

- A ***package*** consists of some related Java classes:
  - Swing: a GUI (graphical user interface) package
  - AWT: Application Window Toolkit (more GUI)
  - **util: utility data structures**

- The ***import*** statement tells the compiler to make available classes and methods of another package

- A ***main*** method indicates where to begin executing a class (if it is designed to be run as a program)

Primitive data types

- Eight primitive data types:
  - **byte**, **short**, **int** and **long** data types are used for storing whole numbers.
  - **float** and **double** are used for fractional numbers.
  - **char** is used for storing characters(letters).
  - **boolean** data type is used for variables that holds either true or false.

String is non-primitive type!

Primitive Data Types

Data type	Range of values
`byte`	-128 .. 127 (8 bits)
`short`	-32,768 .. 32,767 (16 bits)
`int`	-2,147,483,648 .. 2,147,483,647 (32 bits)
`long`	-9,223,372,036,854,775,808 .. ... (64 bits)
`float`	$+/-10^{-38}$ to $+/-10^{+38}$ and 0, about 6 digits precision
`double`	$+/-10^{-308}$ to $+/-10^{+308}$ and 0, about 15 digits precision
`char`	Unicode characters (generally 16 bits per char)
`boolean`	True or false

Primitive vs Non-primitive (Object)

- Primitive types are predefined in Java.
- Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.

int vs Integer

```
public class Examples {
    int id = 0;

 public static void main(String[] args) {
    Examples ex = new Examples();
    ex.id.?

 }
}
```

int vs Integer

```
public class Examples {
    int id = 0;

 public static void main(String[] args) {
    Examples ex = new Examples();
    ex.id.?

 }
}
```

```
public class Examples {
    Integer id = 0;

 public static void main(String[] args) {
    Examples ex = new Examples();
    ex.id.

 }
}
```

- byteValue() : byte - Integer
- compareTo(Integer anotherInteger) : int - Integer
- doubleValue() : double - Integer
- equals(Object obj) : boolean - Integer
- floatValue() : float - Integer
- getClass() : Class<?> - Object
- hashCode() : int - Integer
- intValue() : int - Integer

Java Operators

1. subscript **[ ]**, call **( )**, member access **.**
2. pre/post-increment **++ --**, boolean complement **!**, bitwise complement **~**, unary **+ -**, type cast (type), object creation **new**
3. **/ %**
4. binary **+ -** (**+** also concatenates strings)
5. signed shift **<< >>**, unsigned shift **>>>**
6. comparison **<, <=, >, >=**, class test **instanceof**
7. equality comparison **== !=**
8. bitwise and **&**
9. bitwise or **|**

Java Operators

11. logical (sequential) and **&&**
12. logical (sequential) or **||**
13. conditional **cond ? true-expr : false-expr**
14. assignment **=**, compound assignment **+=, -=, *=, /=, <<=, >>=, >>>=, &=, |=**

…

```
max = (a > b) ? a : b;
```

?

Access Modifiers – Public, Private, Protected & Default

the scope of access modifiers

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

```
1  class Square{
2     private double num = 100;
3     private int square(int a){
4      return a*a;
5     }
6  }
7  public class Examples{
8     public static void main(String args[]){
9      Square obj = new Square();
10     System.out.println(obj.num);
11     System.out.println(obj.square(10));
12    }
13 }
```

This example throws compilation error, why?

Java Collections Framework

- A ***collection*** is a unit that contains a group of objects, e.g., a list of students …

- Java defines a **collections framework since JDK 1.2**, which is a unified architecture for *representing* and *manipulating collections*, allowing them to be manipulated independent of the details of their representation.

- Java collections framework contains:
  - Interfaces
  - Implementations
  - Algorithms

- **documents:**https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

Benefits of Java Collections Framework

- **Reduced Development Effort** – It comes with almost all common types of collections and useful methods to iterate and manipulate the data.

- **Increased Quality** – Using core collection classes that are well tested increases our program quality rather than using any home developed data structure.

- **Reusability and Interoperability**

Java Collections

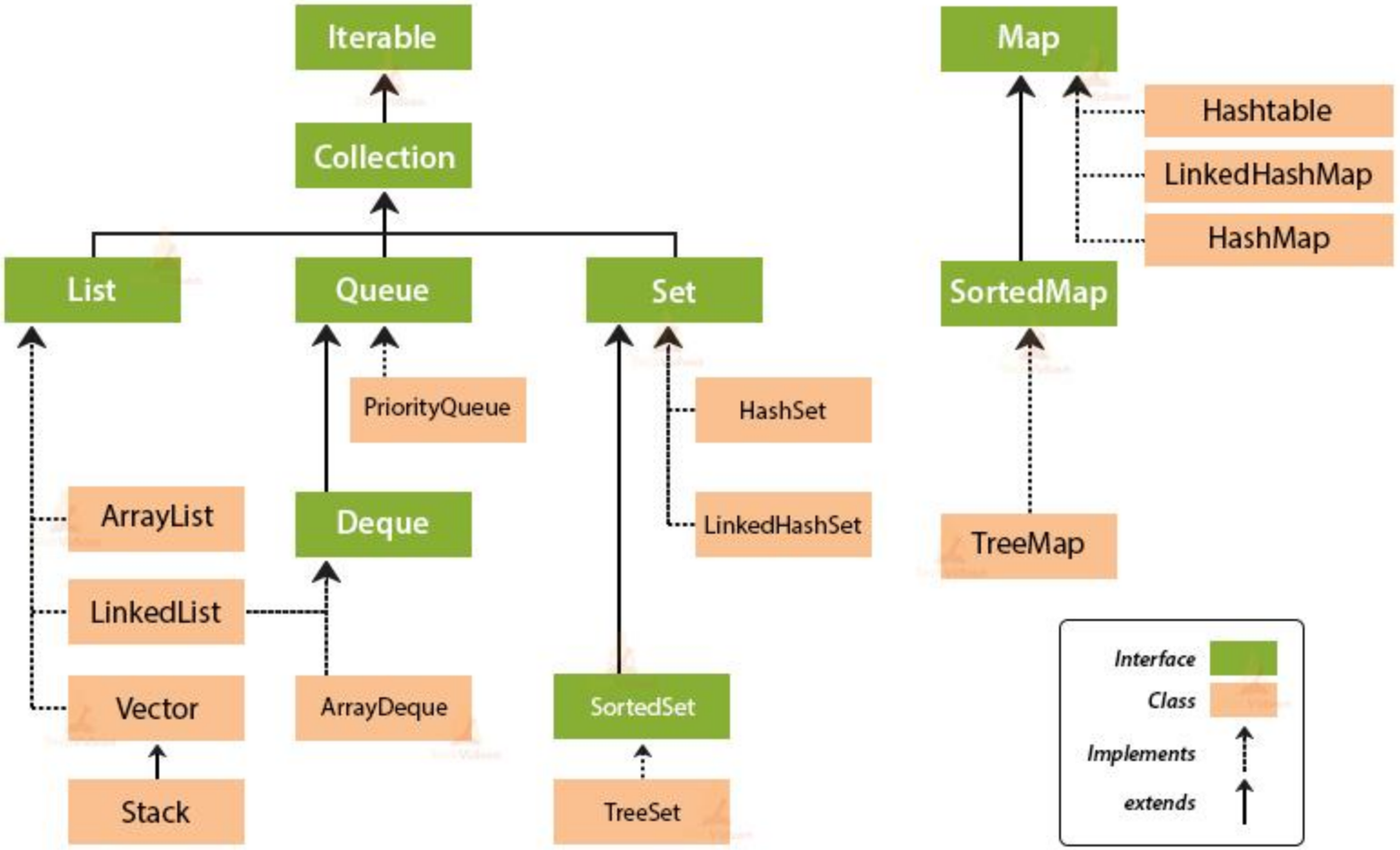


Figure source: https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/03/collection-framework-hierarchy-in-java.jpg

public interface Iterable<T>

- T - the type of elements returned by the iterator

- Implementing this interface allows an object to be the target of the "for-each loop" statement, i.e., *allows it to be iterated*

Modifier and Type	Method and Description
default void	**forEach**(**Consumer**<? super T> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
Iterator<T>	**iterator**() Returns an iterator over elements of type T.
default **Spliterator**<T>	**spliterator**() Creates a **Spliterator** over the elements described by this Iterable.

```java
List<String> list = new ArrayList<String>();

// add elements
list.add("A");
list.add("B");
list.add("C");

// Iterate through the list
for( String element : list ){
    System.out.println( element );
}
```

public interface Collection<E> extends Iterable<E>

- Collection — the root of the collection hierarchy.
- A collection represents a group of objects known as its *elements*.
- Some types of collections allow duplicate elements, and others do not.
- Some are ordered and others are unordered.
- Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

public interface Collection<E> extends Iterable<E>

E - the type of elements in this collection

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Iterators

- An `Iterator` is an object that enables you to *traverse through a collection* and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

public interface Set<E> extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

public interface List<E> extends Collection<E>

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

public interface Queue<E> extends Collection<E>

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

public interface Map<K,V>

- K - the type of keys maintained by this map;V - the type of mapped values
- An object that ***maps keys to values***.
- A Map cannot contain duplicate keys;
- Each key can map to at most one value.
- Hashtable

public interface Map<K,V>

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Map example

```
Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("A", 1);
    map.put("B", 2);
    map.put("C", 3);
    map.put("D", 4);

for(Map.Entry<String, Integer> entry: map.entrySet()) {
    System.out.println(entry.getKey()+" "+entry.getValue());
}
```

Outlines

- **Overview of Java**
  - Java Characteristics
  - Process of Run Java Code
  - Java Class and API
  - Primitive Types
  - Java Operators
  - Access modifiers
  - Java Utils
- **OOP Design Principals**
  - Abstraction
  - Generics
  - Encapsulation
  - Inheritance
  - Polymorphism
- **Junit**
  - Strategies to write good Junit test cases