

Programming for Chemical and Life Science Informatics

I573 - Week 4
(Programming with the CDK)

Rajarshi Guha

3rd February, 2009

What Do We Need?

- The CDK jar files
 - ▶ You can pick and choose which jars you need
 - ▶ Easier to use the big one that has everything and their dependencies
 - ▶ [Last stable release](#) (1.0.4)
 - ▶ [Nightly build](#)
- A Java IDE

- The CDK [wiki](#)
- The nightly build [page](#)
- Code [snippets](#)
- cdk-user mailing list
- [Keyword list](#) and [feature list](#)

What Does the CDK Provide?

- Fundamental chemical objects
 - ▶ atoms
 - ▶ bonds
 - ▶ molecules
- More complex objects are also available
 - ▶ Sequences
 - ▶ Reactions
 - ▶ Collections of molecules
- Input/Output for a wide variety of molecular file formats
- Fingerprints and fragment generation
- Rigid alignments, pharmacophore searching
- Substructure searching, SMARTS support
- Molecular descriptors
- Integration with R and Weka

Who's Using It?

- [ToxTree](#) - toxicity classifier
- [Knime](#) - workflow tool
- [Bioclipse](#) - workbench for chem- and bioinformatics
- [Evince](#) - a QSAR modeling tool
- [JChemPaint](#) - a 2D structure editor

Some Design Principles

- Abstraction is the key principle
 - ▶ Applies to chemical objects such as atoms, bonds
 - ▶ Also applied to input/output
- Rather than directly creating an atom object we use a factory method

```
IAtom atom = DefaultChemObjectBuilder.getInstance().  
                newAtom();
```

and not

```
IAtom atom = new Atom();
```

Some Design Principles

- By using `DefaultChemObjectBuildere` we don't worry how an atom or bond is created
- By using this type of abstraction we can load any file format without having to specify it
- Another aspect is the use of interfaces rather than concrete types
- A molecule is a collection of atoms and bonds
 - ▶ A graph view is not imposed directly
 - ▶ Molecular features such as aromaticity are properties of the atoms and bonds

- Create a carbon atom object using

```
IAtom atom = DefaultChemObjectBuilder.getInstance().  
    newAtom("C");
```

- You can also specify 2D or 3D coordinates
- Once you have an atom, you can get/set coordinates, charge, hydrogen count, parity etc.

- Creating a bond is similar to creating an atom. As before, use `DefaultChemObjectBuilder`

```
IBond bond = DefaultChemObjectBuilder.  
    getInstance().  
    newBond(atom1, atom2, order);
```

- It is also possible to specify the stereo orientation of the bond.

- A molecule is fundamentally represented as an `AtomContainer` object
- For many purposes this is fine
- In some cases specialization is required so we have
 - ▶ Crystal
 - ▶ Ring
 - ▶ Fragment
- Some methods require a specific subclass
- Many methods simply require an `AtomContainer`, so you can use any of the above subclasses

- Similar procedure to getting an atom or bond object

```
IAtomContainer container = DefaultChemObjectBuilder.  
    getInstance().  
    newAtomContainer();
```

- Then we populate it

```
container.addAtom( atom1 );  
container.addAtom( atom2 );  
container.addBond( atom1, atom2, 1.5);
```

Molecules

- What can do with a molecule object?
- Loop over atoms

```
for (IAtom atom : container.atoms()) {  
    // blah  
}
```

- Get a specific atom by serial number

```
IAtom atom = container.getAtom(4);
```

- Get the serial number for a given atom

```
int serial = container.getAtomNumber(anAtom);
```

- In general, it's not a great idea to depend on atom serial numbers (i.e., ordering)

Input/Output - SMILES

- A “line notation” to represent chemical structures
- Originated in 1988 by [Dave Weininger](#) of Daylight CIS
 - ▶ [Theory](#)
- Extensively used in cheminformatics - but has drawbacks
- Reading and writing SMILES is a useful skill
- Various web tools let you paste in a SMILES and see the corresponding structure
 - ▶ [Daylight Depict](#)
 - ▶ [IU dynamic SMILES viewer](#)

SMILES Examples

- CC(=O)O - acetic acid
- c1ccccc1 - benzene
- CC(=O)OC1=CC=CC=C1C(=O)O - aspirin
- CC1=C(C=C(C=C1)NC(=O)C2=CC=C(C=C2)CN3CCN(CC3)C)NC4=NC=CC(=N4)C5=CN=CC=C5 - Gleevec

SMILES Primer

- Elements are represented as their symbols
- Adjacent elements are connected by a single bond if no bond symbol is specified
- Can specify bond symbols:
 - ▶ CC two carbons connected by a single bond
 - ▶ C-F a carbon and fluorine connected by a single bond
 - ▶ C=C two carbons connected by a double bond
 - ▶ C#C two carbons connected by a triple bond
- Hydrogens are generally omitted
- Lower case symbols imply sp^2 hybridization (which usually implies aromaticity)

SMILES Primer - Branches

- Branches are represented by parentheses
- Any good SMILES parser will handle 100 or more branches
 - ▶ Probably incomprehensible to a human!
- Best to visualize these in a depictor
- CCCC - a straight chain
- CC(CC)CC - one branch on the second carbon
- CC(CN)C(NC(C)(C))C - branches can be contained within branches

SMILES Primer - Rings

- Rings are indicated by “marking” the start atom and end atom of the ring with a number
- c1ccccc1 - benzene. The first c1 indicates the start atom of the ring
...
- An atom can be part of two rings: C1CCC12CCC2
- or three rings C1CCCC123CCCC2CC3 (but this is probably too unstable to exist)

SMILES Primer - Aromaticity

- Tricky subject - lots of ambiguities
- The Daylight spec itself is ambiguous
- A lower case symbol indicates sp^2 hybridization, but the spec implies that they are part of aromatic systems
- The toolkit must decide what to do
 - ▶ CDK requires that you determine aromaticity - so cccc shows up as butane
 - ▶ Daylight will consider the atoms to sp^2 and then deduce aromaticity - so cccc shows up as butadiene
- One way to avoid this ambiguity is to specify a Kekule structure
 - ▶ Force aromaticity detection

SMILES Primer - Disconnected Components

- Sometimes molecules are associated with another molecule
 - ▶ dimer
 - ▶ salt
- Such non-covalent associations are represented using a period
- CCC.CCC two propane molecules
- CC(=O)[O-].[Na+] sodium acetate

Input/Output

- SMILES are a common format
- SMILES parsing, you will have to handle the `InvalidSmilesException`

```
SmilesParser sp = new SmilesParser(
    DefaultChemObjectBuilder.
        getInstance());
IMolecule molecule = sp.parseSmiles("CCCC(CC)CC=CC=CC");
```

- Given a molecule object, we can create a SMILES from it

```
SmilesGenerator sg = new SmilesGenerator();
String smiles = sg.createSMILES(molecule);
System.out.println("SMILES = "+smiles);
```

- Reading files from disk is a common task
- A little more complex due to abstraction but is quite general

```
File file = new File("mols.sdf");
FileReader fileReader = new FileReader(file);
MDLV2000Reader reader = new MDLV2000Reader(fileReader);
IChemFile chemFile = (IChemFile) reader.
    read(new ChemFile());
List containers = ChemFileManipulator.
    getAllAtomContainers(chemFile);
```

- This is nice since it allows you to get all the molecules in the file at one go
- Not a good idea for very large SD files
- In such cases, use [IteratingMDLReader](#)
- Allows you to iterate over arbitrarily large SD files
- There is also an [IteratingSMILESReader](#) for big SMILES files

- Writing files is also supported for a wide variety of formats
- For example, to write to SD format

```
FileWriter w1 = new FileWriter(new File("molecule.sdf"));
try {
    MDLWriter mw = new MDLWriter(w1);
    mw.write(molecule);
    mw.close();
} catch (Exception e) {
    System.out.println(e.toString());
}
```


SD Tags

- When a molecule is read in from an SD file we can get the value of a given tag by doing

```
Object value = molecule.getProperty("myProperty");
```

- When writing a molecule we can supply a set of tags in a HashMap

```
HashMap tags = new HashMap();  
tags.put("myProperty", new Double(1.2));
```

SD Tags

- To ensure tags are written to disk we would do

```
FileWriter w1 = new FileWriter(new File("molecule.sdf"));
try {
    MDLWriter mw = new MDLWriter(w1);
    mw.setSdFields( tags );
    mw.write( molecule );
    mw.close();
} catch (Exception e) {
    System.out.println( e.toString() );
}
```

- A molecule might have several properties associated with it.
- We can avoid creating an extra HashMap, when writing them *all* to disk

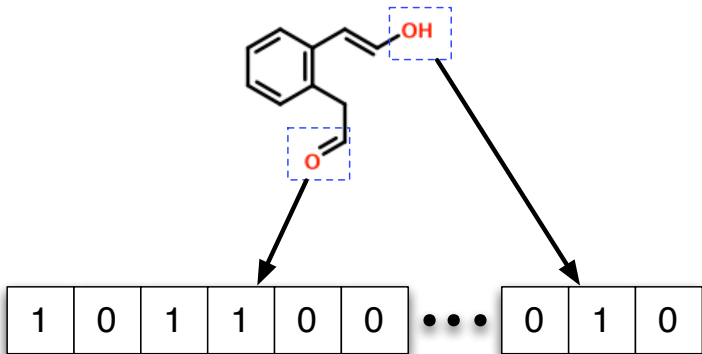
```
MDLWriter mw = new MDLWriter(w1);  
mw.setSdFields( molecule.getProperties() );  
mw.write(molecule);  
mw.close();
```

Generating Canonical SMILES

- The `SmilesGenerator` class will create canonical SMILES

```
String smiles = "C(C)(C)CC=CC(C(CC(C))CC)CC";  
SmilesParser sp = new SmilesParser();  
IMolecule mol = sp.parseSmiles(smiles);  
  
SmilesGenerator sg = new SmilesGenerator();  
String canSmi = sg.createSMILES(mol);
```

Fingerprints



- Used in many areas - database screening, diversity analysis, modeling

Fingerprints

- The simplest fingerprint looks for specific substructures and sets a specific bit
 - ▶ MACCSFingerprinter
 - ▶ SubstructureFingerprinter
- Hashed fingerprints don't maintain a correspondence between bit position and substructural feature

Fingerprints

```
int L ← depth
int S ← size
List paths
for atom in molecules
    paths ← get all paths of length L

BitSet b
for path in paths
    code ← path.hasCode()
    position ← Random(code).nextInt(S)
    b.set(position)
```

Fingerprints

- What is a path?
- CDK represents them as normalized strings
- C-C=C-N#C
- Care must be taken when dealing with two letter symbols such as Cl or Br
- Replace them with predefined letters
- Allows us to check that a path is the reverse of another path

Getting Fingerprints

- The CDK can generate binary [fingerprints](#)
- This gives a default fingerprint of 1024 bits - but you can specify smaller or larger fingerprints
- The CDK also provides variants of the default fingerprint
 - ▶ [MACCSFingerprint](#)
 - ▶ [ExtendedFingerprinter](#) - includes bits related to rings
 - ▶ [GraphOnlyFingerprinter](#) - simplified version of fingerprinter that ignores bond order
 - ▶ [SubstructureFingerprinter](#) - generates fingerprints based on a specified set of substructures

```
IFingerprinter fprinter = new Fingerprinter()  
BitSet fingerprint = fprinter.getFingerprint(molecule);
```

Evaluating Similarity

- Given two molecules we are interested in determining how similar they are
- A common approach is to evaluate their fingerprints
- Calculate a similarity metric (e.g., Tanimoto coefficient)

```
BitSet fp1 = Fingerprinter.getFingerprint(molecule1);  
BitSet fp2 = Fingerprinter.getFingerprint(molecule2);  
  
float tc = Tanimoto.calculate(fp1, fp2);
```

Molecular Descriptors

- If you know which descriptor to calculate, say [ZagrebIndexDescriptor](#)

```
ZagrebIndexDescriptor desc = new ZagrebIndexDescriptor();  
DescriptorValue value = desc.calculate(atomContainer);  
DoubleResult result = (DoubleResult) value.getValue();  
  
System.out.println("value = "+result.doubleValue());
```

- The `value` object contains more than just the value
- Stores information such as
 - ▶ who wrote the descriptor
 - ▶ a link to an OWL dictionary

Molecular Descriptors

- It's possible to calculate them all automatically

```
IMolecule molecule;  
DescriptorEngine engine = new DescriptorEngine ();  
engine.process(molecule);
```

- Descriptors are placed as properties of the molecule
- Access them by the `getProperty` or `getProperties` method of the molecule object

Molecular Descriptors

- Note that a descriptor can return different types and numbers of values
 - ▶ Chi descriptors can return 10 or more
 - ▶ AlogP returns 2
- The `DescriptorResult` class has several subclasses representing these situations
- You should use `instanceof` to identify what type of data a descriptor actually returns

```
DescriptorResult retval = descriptor.calculate(ac);  
if (retval instanceof DoubleArrayResult) {  
    // process  
} else if (retval instanceof DoubleResult) {  
    // process  
}  
...  
...
```

Substructure Searching

- Substructure searching is a very common task
- *How many compounds in my database contain an acetyl group?*
- The simplest form of substructure search specifies a SMILES as the query
- *How many compounds in my database contain $C(=O)C$*
- The CDK supports this with the SMARTSQueryTool

Substructure Searching

- Identify the presence/absence of a substructure in a molecule

```
IAtomContainer mol = ...;  
SMARTSQueryTool sqt = new SMARTSQueryTool("C");  
sqt.setSmarts("C(=O)C");  
boolean matched = sqt.matches(mol);
```

- In many cases we're interested in all the possible matches

```
List<List<Integer>> maps;  
maps = sqt.getUniqueMatchingAtoms();
```

- `maps.size()` gives you the number of matches
- Each element of `maps` is a sequence of atom ID's

Substructure Searching - SMARTS

- Using a SMILES as a query is easy but limited
- More generic queries are desirable
 - ▶ Find all molecules with an aromatic ring
 - ▶ Find molecules with a N singly bonded to 3 carbons
 - ▶ Find molecules where a C is a part of 3 ring systems
- You can't specify a SMILES for any of these queries
- Instead we use the SMARTS language

SMARTS are (like) regular expressions for chemical structures

Substructure Searching - SMARTS

- Invented at Daylight CIS
- [Theory](#) and [Examples](#)
 - ▶ All toolkits will support the spec linked to
 - ▶ Some toolkits provide extensions - read the docs
- Any SMILES is a valid SMARTS pattern, but the converse is not true
- They can become very complex and difficult to read
- Using a [SMARTS depicter](#) is very handy to get a SMARTS working
- Three classes of SMARTS symbols
 - ▶ Atom primitives
 - ▶ Bond primitives
 - ▶ Logical operators

SMARTS - Atom Primitives

- *, a, A - any atom, aromatic atom, aliphatic atom respectively
- R<n> - atom is a member of n rings
- r<n> - atom is in a ring of size n
- D<n> - atom has n explicit connections
- #<n> - atom with atomic number n
- X<n> - atom with n total connections

SMARTS - Examples

- C or [#6] matches any carbon
- [R] matches any atom in a ring
- [D3] matches any atom with 3 **explicit** bonds
 - ▶ matches once in CCCNCCC(=O)OCC
- [X3] matches atoms with 3 connections
 - ▶ matches twice in CCCNCCC(=O)OCC

SMARTS - Bond Primitives

- All the SMILES bond symbols are valid SMARTS symbols
 - ~ - indicates any type of bond
 - @ - indicates a ring bond
-
- CC - matches two adjacent ring carbons
 - CCC - matches 2 adjacent ring carbons joined to a non-ring carbon
 - C N - matches any adjacent C and N

SMARTS - Logical Operators

- ! - not
 - & - and (high precedence)
 - , - or
 - ; - and (low precedence)
-
- CC(=O) [C,N] - identify acetyl or amide linkages
 - [CD3&r4] - identify carbon with 3 connections and part of a 4-member ring
 - [CD3&r] - identify carbon with 3 connections and part of any ring
- In general, you can get by without using SMARTS
- When you do use it (correctly), make life **much** easier

Pharmacophores

- We'll look at constructing a pharmacophore query by hand
- Generally, you'd read it from a file
- Given a query we can search for that in a target molecule
- The target molecule should have 3D coordinates
- Generally you'll examine multiple conformers for a given molecule

- Construct pharmacophore groups

```
PharmacophoreQueryAtom o =  
    new PharmacophoreQueryAtom( "D", "[!H0;#7,#8,#9]" );  
PharmacophoreQueryAtom n1 =  
    new PharmacophoreQueryAtom( "A", "c1ccccc1" );  
PharmacophoreQueryAtom n2 =  
    new PharmacophoreQueryAtom( "A", "c1ccccc1" );
```

- Construct pharmacophore constraints

```
PharmacophoreQueryBond b1 =  
    new PharmacophoreQueryBond(o, n1, 4.0, 4.5);  
PharmacophoreQueryBond b2 =  
    new PharmacophoreQueryBond(o, n2, 4.0, 5.0);  
PharmacophoreQueryBond b3 =  
    new PharmacophoreQueryBond(n1, n2, 5.4, 5.8);
```

Pharmacophores

- Given pharmacophore groups and constraints, we construct a pharmacophore query
- Design to have the same semantics as a normal molecule

```
QueryAtomContainer query =  
    new QueryAtomContainer ();
```

```
query .addAtom (o );  
query .addAtom (n1 );  
query .addAtom (n2 );
```

```
query .addBond (b1 );  
query .addBond (b2 );  
query .addBond (b3 );
```

Pharmacophores

- Given the query we can now perform a pharmacophore search

```
IAtomContainer aMolecule;  
  
// load in the molecule  
  
PharmacophoreMatcher matcher =  
    new PharmacophoreMatcher(query);  
boolean status = matcher.matches(aMolecule);  
if (status) {  
    // get the matching pharmacophore groups  
    List<List<PharmacophoreAtom>> pmatches =  
        matcher.getUniqueMatchingPharmacophoreAtoms();  
}
```