Interactive Development Environments

- A system that covers many development tasks
  - Editor – usually with nice syntax coloring, indentation
  - Compiler – automatic compilation, errors linked to code
  - Debugger – step through source code
  - Etc... – Testing, search, code transformations, ...

- Examples: DrJava, NetBeans, Eclipse, Visual Studio, emacs

Dr. Java

- Light-weight IDE
- Editing
  - Syntax coloring, auto-indent, brace matching
- Testing
  - Integrates with Junit testing framework
    - Uses suite() or auto-generated suite
  - Interaction panel allows interactive method invocations
- Debugging
  - Integrates with Java debugger
  - Interactions panel also useful

Debugging

- My program doesn’t work: why?
- Use the scientific method:
  - Study the data
    - Some tests work, some don’t
  - Hypothesize what could be wrong
  - Run experiments to check your hypotheses
    - Testing!
  - Iterate

Starting to Debug

- What are the symptoms of the misbehavior?
  - Input/output
  - Stack trace (from thrown exception)
- Where did the program fail?
- What could have led to this failure?
- Test possible causes, narrow down the problem

Checking that Properties Hold

- Print statements
  - Check whether values are correct
    - E.g., look at value of i to check if i > 0
  - Check whether control-flow is correct
    - E.g., see if f(i) is called after g(i)
- Automatic debugger
  - Allows you to step through the program interactively
  - Verify expected properties
    - Don’t need to put in print statements and recompile
  - Use as part of testing
**Dr. Java Interactions Pane**

- Can evaluate Java expressions interactively
  - Can bind variables, execute expressions/statements
- **Benefits**
  - Make sure that methods work as expected
  - Test invariants by constructing expressions not in program text
  - Combines with interactive debugger

**Dr. Java’s Automatic Debugger**

- Set execution breakpoints
- Step through execution
  - **into**, over, and **out** of method calls
- Examine the stack
- Examine variable contents
- Set watchpoints
  - Notified when variable contents change

**Using the Debugger**

- Set debug mode to on
  - Turns on debug panel with state information
- Set break point(s) in Java source
- Run the program

**Tips**

- Make bug reproducible
  - If it’s not reproducible, what does that imply?
- Boil down to smallest program that reproduces bug
  - Reveals the core problem
- Explain problem to someone else (i.e., instructor or TA)
  - Explaining may reveal the flaw in your logic
- Keep notes: don’t make the same mistake twice

**Defensive Programming**

- Assume that other methods/classes are broken
  - They will misuse your interface
  ```java
  public Vector(int initialCapacity, int capacityIncrement) {
      super();
      if (initialCapacity < 0)
          throw new IllegalArgumentException("Illegal Capacity: +" + initialCapacity);
      ...
  }
  ```
- Goal: Identify errors as soon as possible

**Avoiding Errors**

- Codify your assumptions
  - Include checks when entering/exiting functions, iterating on loops
- Test as you go
  - Using Junit
  - Using the on-line debugger
- Re-test when you fix a bug
  - Be sure you didn’t introduce a new bug
- Do not ignore possible error states
  - Deal with exceptions appropriately
Abstraction and Parametric Polymorphism
September 22, 2004

Data Abstraction
- Data abstraction = objects + operations
  - List + { addFirst, addLast, removeFirst, ... }
  - Set + { add, contains, ... }
- Categories of operations
  - Constructors (creators/producers)
  - Mutators/Modifiers
  - Observers

Abstraction Function
- Specification for data structure is abstract
- Implementation of data structure is concrete
- How do you know if implementation meets the spec?
- Abstraction function : concrete → abstract
  - Specifies how the representation of an abstract data type is interpreted as an abstract value.

Example
```
class IntSet {
  int[] elts; ...}
```
- AF(s) = \{ x.elts[i] | 0 <= i <= elts.length \}

- You always need an abstraction function when you build a data abstraction
  - Often it’s implicit

Representation Invariant(s)
- Constraints that characterize whether a concrete instance of an abstract data type is well formed
- Constraints must hold
  - After the constructor has finished
  - Before and after each operation
```
class IntSet {
  // rep inv: elts contains no duplicates
  int[] elts; ...
}
```
- Part of the (internal) specification

Implementing the Rep Invariant
- Interesting idea: Write a function to check the rep
```
public boolean repOK() {
  ...check for duplicates in elts...
}
```
- Where can you use this?
  - Can add wherever you expect rep to hold
  - Can call during unit testing
- Cost?
### Graphical Example

- Every element in A has at least one rep in R
- Some elements in R not mapped to any element in A

### Exposing the Rep

- Be careful of exposing the representation
  ```java
class IntSet {
    int[] elts;
    int[] getElements () { return elts; }
}
```
- Why?
  - Other people may rely on implementation details
  - Clients may violate the rep invariant

### Polymorphism

- Recall that B is a subtype of A if, everywhere you expect an A, you can accept a B
  - Subtypes come from subclassing with extends
  - Subtypes come from interfaces with implements

- This is a kind of type polymorphism
  - Methods can accept objects of many types, not just one
  - This is usually called subtype polymorphism

### IntegerStack Client

```java
IntegerStack is = new IntegerStack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

- This is OK, but what if we want other kinds of stacks?
  - Need to make one XStack for each kind of X
  - Problems: Code bloat, maintainability nightmare

### Polymorphism Using Object

```java
class IntegerStack {
  class Entry {
    Integer elt; Entry next;
    Entry(Integer i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(Integer i) {
    theStack = new Entry(i, theStack);
  }
  Integer pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      Integer i = theStack.elt;
      theStack = theStack.next;
      return i;
    }
  }
}
```

### Polymorphism Using Object

```java
class Stack {
  class Entry {
    Object elt; Entry next;
    Entry(Object i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(Object i) {
    theStack = new Entry(i, theStack);
  }
  Object pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      Object i = theStack.elt;
      theStack = theStack.next;
      return i;
    }
  }
}
Stack Client

```java
Stack is = new Stack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = (Integer) is.pop();
```

- Now Stacks are reusable
  - `push()` works the same
  - But now `pop()` returns an Object
    - Have to downcast back to `Integer`
    - Not checked until run-time

General Problem

- When we move from an `X` container to an `Object` container
  - Methods that take `X`'s as input parameters are OK
    - If you're allowed to pass `Object` in, you can pass any `X` in
  - Methods that return `X`'s as results require downcasts
    - You only get `Object`s out, which you need to cast down to `X`

- This is a general feature of `subtype` polymorphism

Parametric Polymorphism (for Classes)

- Idea: We can parameterize the Stack class by its element type
- Syntax:
  - Class declaration: `class A<T> { ... }`
    - `A` is the class name, as before
    - `T` is a type variable, can be used in body of class `(...)`
  - Client usage declaration: `A<Integer> x;`
    - We instantiate `A` with the `Integer` type

Parametric Polymorphism for Stack

```java
class Stack<Element> {
    class Entry {
        Element elt;
        Entry next;
        Entry(Element i, Entry n) { elt = i; next = n; }
    }
    Entry theStack;
    void push(Element i) { theStack = new Entry(i, theStack); }
    Element pop() throws EmptyStackException {
        if (theStack == null)
            throw new EmptyStackException();
        else {
            Element i = theStack.elt;
            theStack = theStack.next;
            return i;
        }
    }
}
```

Stack `<Element>` Client

```java
Stack<Integer> is = new Stack<Integer>();
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

- No downcasts
- Type-checked at compile time
- No need to duplicate Stack code for every usage

Parametric Polymorphism for Procedures

- Suppose `B` is a subtype of `A`
  1. static A id(A x) { return x; }
  2. static A id(B x) { return x; }
  3. static B id(A x) { return x; }
  4. static B id(B x) { return x; }

- Can’t pass an `A` to 2 or 4
- 3 doesn’t type check
- Can pass a `B` to 1 but you get an `A` out
Parametric Polymorphism, Again

- Observation: id() doesn’t care about the type of x
  - It works for any type

- So parameterize the id method:
  
  ```java
  static <T> T id(T x) { return x; }
  Integer i = id(new Integer(3));     // Notice no need to
  // instantiate id; compiler
  // figures it out
  ```

Summary: Kinds of Polymorphism

- Subtype polymorphism
  - Use subtype wherever supertype allowed

- Parametric polymorphism
  - When classes/methods work for any type; uses type variables

- Ad-hoc polymorphism
  - Overloading in Java

Parametric Polymorphism in Java

- Part of Java 1.5 (called “generics”)
  - Available in beta now
  - Comes with replacement for java.util.*
    - class LinkedList<A> { ... }  ==>  class LinkedList { ... }
    - class HashMap<A, B> { ... }  ==>  class HashMap { ... }
    - interface Collection<A> { ... }  ==>  interface Collection { ... }

- Available on linuxlab
  - In directory /usr/local/j2sdk1.5.0
  - Run /usr/local/j2sdk1.5.0/bin/javac to compile
  - Run /usr/local/j2sdk1.5.0/bin/java to execute
  - API at http://java.sun.com/j2se/1.5.0/docs/api

Implementation

- Generics translated into standard Java byte codes
  - Java VM hasn’t changed
  - Compiled programs can be run on any correct implementation of the JVM
  - Intuitively, generics “compiled out” of programs

Translation via Erasure

- (According to OOPSLA98 paper on gj)
- Replaces uses of type variables with Object
  - class A<T> { ...T x... }  ==>  class A { ...Object x... }
- Adds downcasts wherever necessary
  - Integer x = A<Integer>.get();  ==>  Integer x = (Integer) (A.get());
- Some complications with overloading
- Need to be careful with security
  - LinkedList<SecureChannel>
Using with Legacy Code

• Translation via type erasure
  – `class A<T> => class A`

• Thus class A is available as a “raw type”
  – `class A<T> { ... }`
  – `class B { A x; }`

• Sometimes useful with legacy code, but...
• Dangerous feature to use, plus unsafe
  – Relies on implementation of generics, not semantics