Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Java basics, functional correctness

Charlie Garrod Michael Hilton

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due Thursday 11:59 p.m.
- Everyone must read and sign our collaboration policy

Key concepts from last Thursday

Key concepts from last Thursday

- Polymorphism
- Information hiding
- Contracts

A polymorphism example

```
interface Dice {
  void roll();
  int getFaceValue();
}
```

A polymorphism example

```
interface Dice {
  void roll();
  int getFaceValue();
}
```

```
public class RandomDice implements Dice {
  private int faceValue;
  @Over
  publi
    fac
  }
  @Over
  publi
}
```

```
public class LoadedDice implements Dice {
  @Override public void roll() { }
  @Override
  public int getFaceValue() {
    if (Math.random() < 0.5) {
      return 4;
    }
    return 2;
  }
}
```

A polymorphism example

```
interface Dice {
  void roll();
  int getFaceValue();
}
```

```
public class RandomDice implements Dice {
  private int faceValue;
  @Over
  publi
    fac
  }
  @Over
  publi
}
```

```
public class LoadedDice implements Dice {
  @Override public void roll() { }
  @Overr
  public
    if (
      re
    }
    retu
  }
}
```

```
static void play(Dice d1, Dice d2) {
  d1.roll();
  d2.roll();
  if (d1.getFaceValue() + d2.getFaceValue() == 7) {
    System.out.println("You win!");
  } else {
    System.out.println("You lose.");
  }
}
```

Reading assignment due today

- Effective Java, Items 13 and 14

Upcoming reading assignments

- Thursday (optional):
  - Effective Java, Items 8, 9, and 56
  - UML and Patterns, Ch. 16
- Next Tuesday:
  - Effective Java, Items 15 and 39

Today

- Introduction to Java
  - JVM basics
  - Dynamic dispatch
  - Collections
- Functional correctness
  - JUnit and friends

Java: A virtual machine architecture

- You first compile the source file:
  - `javac HelloWorld.java`
    - Produces `HelloWorld.class`
- Then run the class file with a Java Virtual Machine (JVM):
  - `java HelloWorld`
    - Executes the `main` method

Java type system

- Primitive types
  - `int`, `long`, `double`, `boolean`, `short`, `char`, `float`, `byte`
- Object types
  - Classes, interfaces, arrays, enums, annotations
  - Identity (==) is conceptually distinct from equality (`.equals(…)`)

Java type system

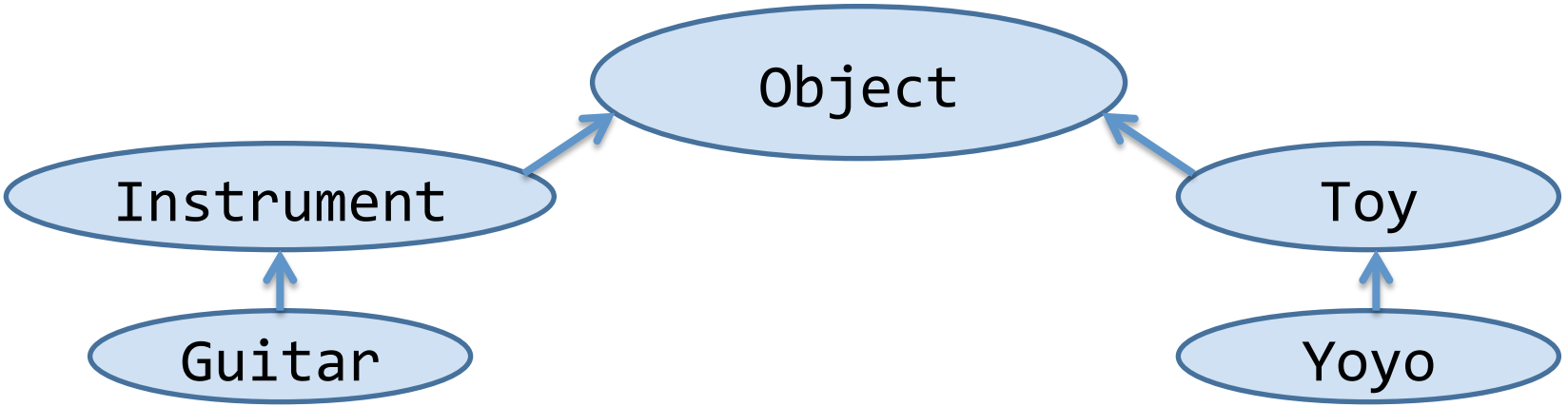
- Primitive types
  - `int`, `long`, `double`, `boolean`, `short`, `char`, `float`, `byte`
- Object types
  - Classes, interfaces, arrays, enums, annotations
  - Identity (==) is conceptually distinct from equality (`.equals(…)`)
- Java sometimes converts between primitive and object types
  - `Integer`, `Long`, `Double`, `Boolean`, `Short`, `Char`, `Float`, `Byte`
  - "Autoboxing" and "unboxing"

Java type system

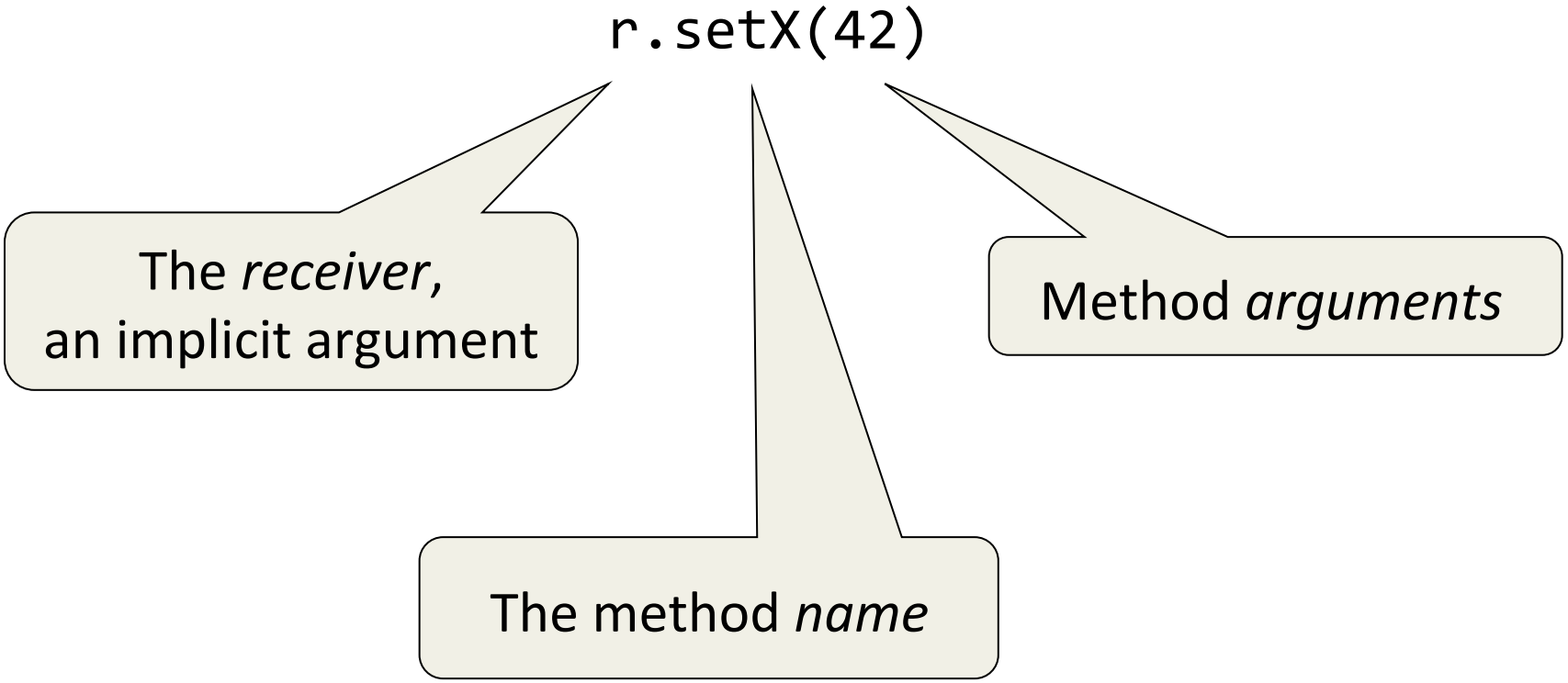
- Primitive types
  - `int`, `long`, `double`, `boolean`, `short`, `char`, `float`, `byte`
- Object types
  - Classes, interfaces, arrays, enums, annotations
  - Identity (==) is conceptually distinct from equality (`.equals(…)`)
- Java sometimes converts between primitive and object types
  - `Integer`, `Long`, `Double`, `Boolean`, `Short`, `Char`, `Float`, `Byte`
  - "Autoboxing" and "unboxing"
- *Generic* types (a.k.a. Parameterized types)
  - e.g. `List<Integer>`, `HashMap<Bicycle,Double>`

The class hierarchy

- The root is `Object` (all non-primitives are `Objects`)
- All classes except `Object` have one parent class
  - Specified with an `extends` clause:
    ```
    class Guitar extends Instrument { ... }
    ```
  - If `extends` clause is omitted, defaults to `Object`
- A class is an instance of all its superclasses



Anatomy of a method call



Which method is actually executed?

```
interface Dice {
  void roll();
  int getFaceValue();
}
```

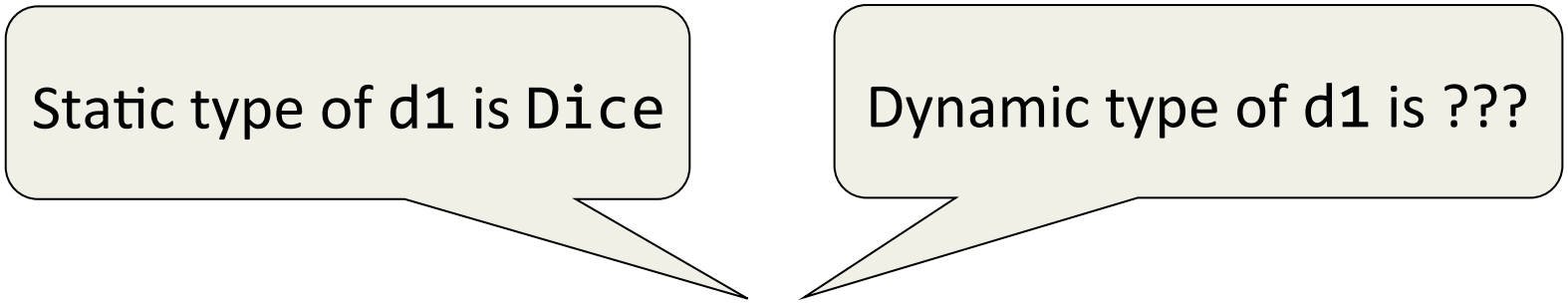
```
public class RandomDice implements Dice {
  private int faceValue;
  @Over
  publi
    fac
  }
  @Over
  publi
}
```

```
public class LoadedDice implements Dice {
  @Override public void roll() { }
  @Overr
  public
    if (
      re
    }
    retu
  }
}
```

```
static void play(Dice d1, Dice d2) {
  d1.roll();
  d2.roll();
  if (d1.getFaceValue() + d2.getFaceValue() == 7) {
    System.out.println("You win!");
  } else {
    System.out.println("You lose.");
  }
}
```

Static types vs. dynamic types

- *Static* type: The compile-time type declaration of a variable
- *Dynamic* type: The actual run-time type of the object in memory




```
static void play(Dice d1, Dice d2) {
  d1.roll();
  d2.roll();
  if (d1.getFaceValue() + d2.getFaceValue() == 7) {
    System.out.println("You win!");
  } else {
    System.out.println("You lose.");
  }
}
```

Dynamic method dispatch (conceptually)

```
d1.roll()
```

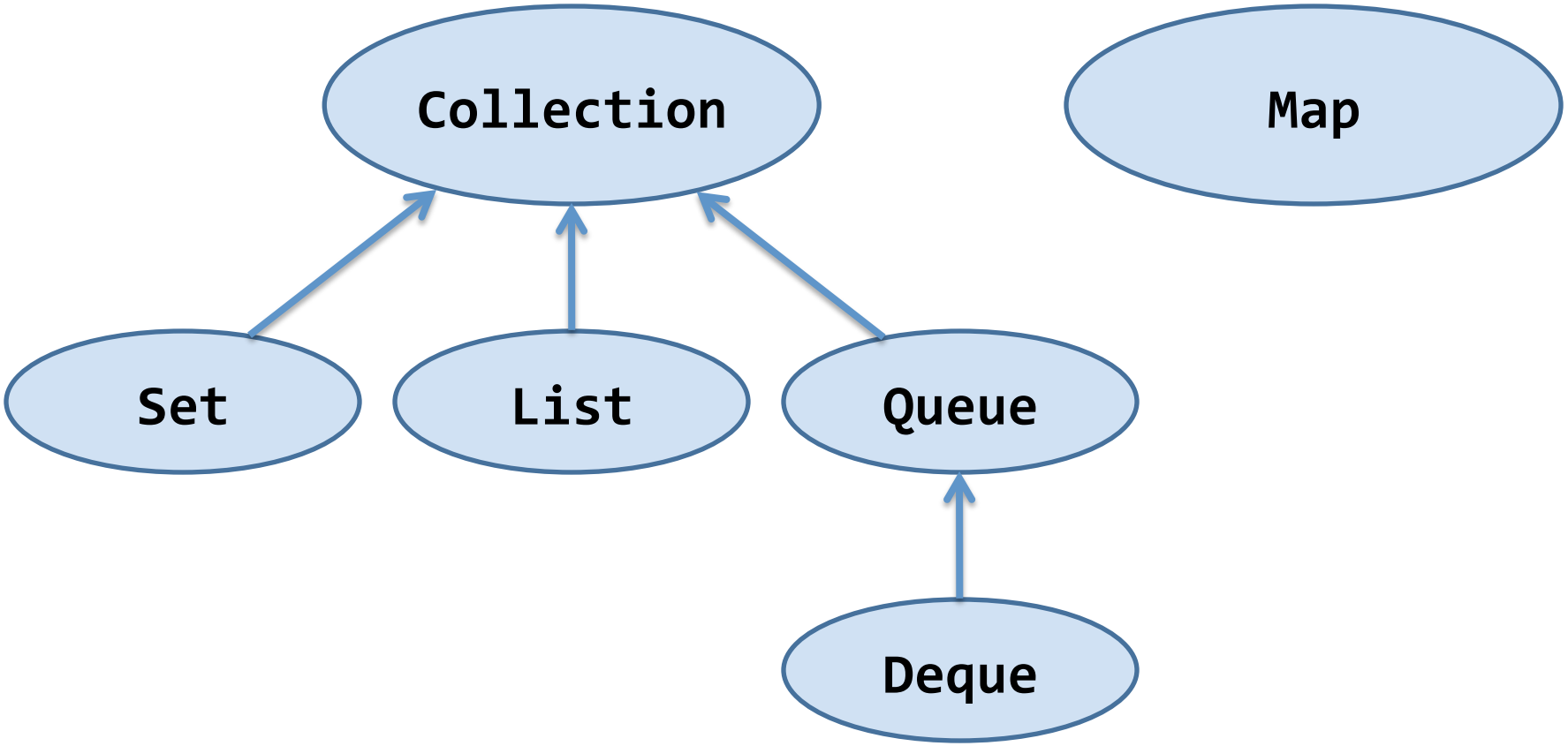
- Step 1 (compile time): Determine which type to look in
  - Static type of the receiver
- Step 2 (compile time): Find the best matching method
  - Find method in class/interface with matching name and argument types
    - Search parent classes/interfaces if necessary
    - Record the *signature* of the method call found

Dynamic method dispatch (conceptually)

```
d1.roll()
```

- Step 1 (compile time): Determine which type to look in
  - Static type of the receiver
- Step 2 (compile time): Find the best matching method
  - Find method in class/interface with matching name and argument types
    - Search parent classes/interfaces if necessary
    - Record the *signature* of the method call found
- Step 3 (run time): Determine dynamic type of the receiver
  - Dynamic type of each object stored in memory at run-time
- Step 4 (run time): Locate the method to invoke
  - Method in run-time class that matches signature found in Step 2

Primary collection interfaces



Primary collection implementations

Interface	Implementation
`Set`	`HashSet`
`List`	`ArrayList`
`Queue`	`ArrayDeque`
`Deque`	`ArrayDeque`
`[stack]`	`ArrayDeque`
`Map`	`HashMap`

Other noteworthy collection implementations

Interface	Implementation(s)

`Set`	`LinkedHashSet`
	`TreeSet`
	`EnumSet`

`Queue`	`PriorityQueue`

`Map`	`LinkedHashMap`
	`TreeMap`
	`EnumMap`

Collections usage example 1

Squeeze duplicate words out of command line

```
public class Squeeze {
   public static void main(String[] args) {
       Set<String> s = new LinkedHashSet<>();
       for (String word : args)
           s.add(word);
       System.out.println(s);
   }
}

$ java Squeeze I came I saw I conquered
[I, came, saw, conquered]
```

Collections usage example 2

Print unique words in lexicographic order

```
public class Lexicon {
   public static void main(String[] args) {
       Set<String> s = new TreeSet<>();
       for (String word : args)
           s.add(word);
       System.out.println(s);
   }
}

$ java Lexicon I came I saw I conquered
[I, came, conquered, saw]
```

Collections usage example 3

Print index of first occurrence of each word

```
class Index {
    public static void main(String[] args) {
        Map<String, Integer> index = new TreeMap<>();

        // Iterate backwards so first occurrence wins
        for (int i = args.length - 1; i >= 0; i--) {
            index.put(args[i], i);
        }
        System.out.println(index);
    }
}
```

```
$ java Index if it is to be it is up to me to do it
{be=4, do=11, if=0, is=2, it=1, me=9, to=3, up=7}
```

What about arrays?

- Arrays aren't really a part of the collections framework
  - There is an adapter: `Arrays.asList`
- Arrays and collections don't mix
  - If you get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
  - See *Effective Java* Item 25 for details

More information on collections

- For much more information on collections, see: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html

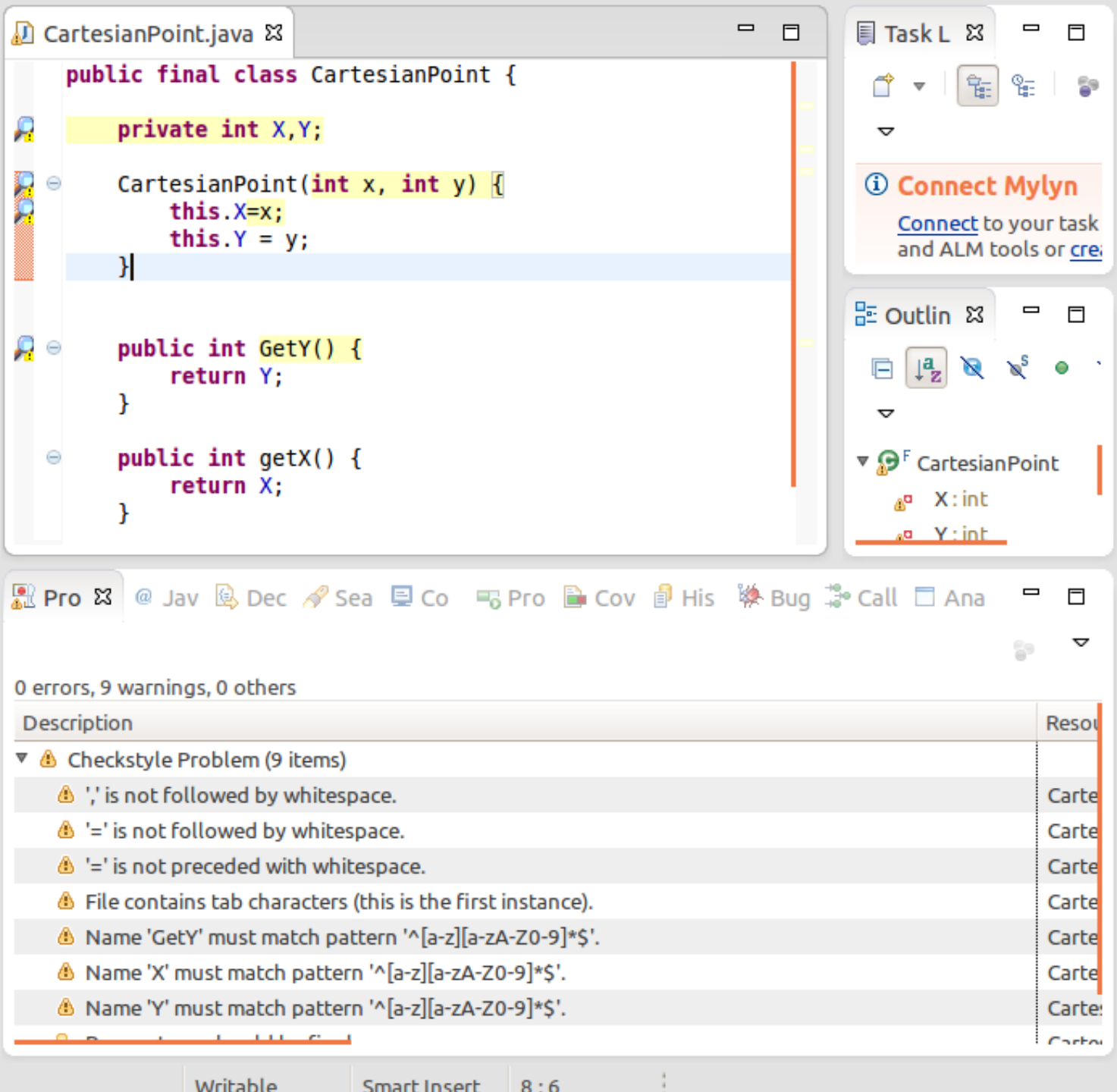
Today

- Introduction to Java
  - JVM basics
  - Dynamic dispatch
  - Collections
- Functional correctness
  - JUnit and friends

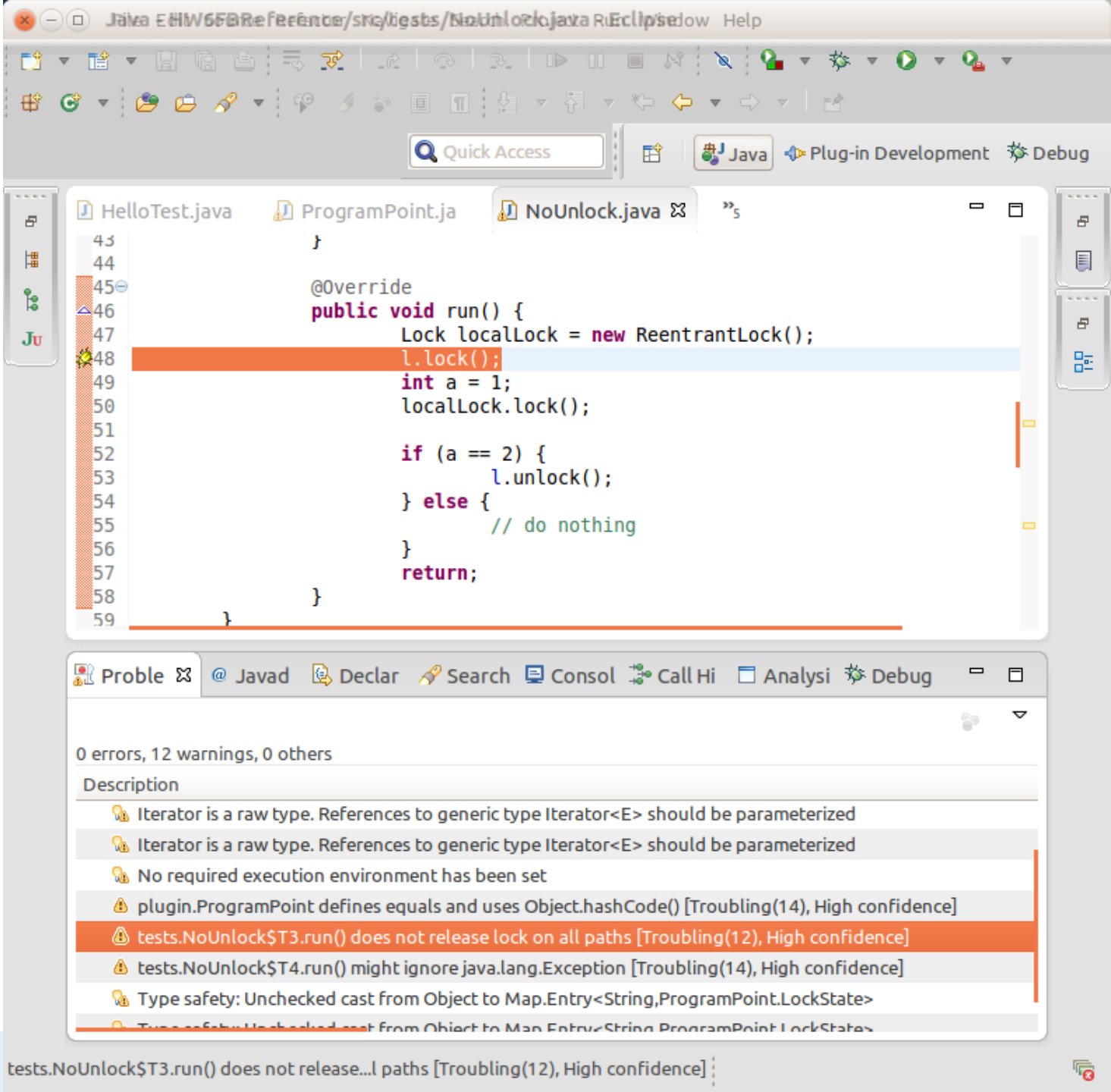
Functional correctness

- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- …

CheckStyle


CartesianPoint.java
public final class CartesianPoint {
private int X,Y;
CartesianPoint(int x, int y) {
this.X=x;
this.Y = y;
}
public int GetY() {
return Y;
}
public int getX() {
return X;
}
Task L
Connect Mylyn
Connect to your task
and ALM tools or
Outlin
CartesianPoint
X : int
Y : int
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 9 warnings, 0 others
Description
Checkstyle Problem (9 items)
',' is not followed by whitespace.
'=' is not followed by whitespace.
'=' is not preceded with whitespace.
File contains tab characters (this is the first instance).
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Writable
Smart Insert
8 : 6

FindBugs


Java EE - HW6FBReference/src/tests/NoUnlock.java - Eclipse
Quick Access
Java
Plug-in Development
Debug
HelloTest.java
ProgramPoint.ja
NoUnlock.java
@Override
public void run() {
Lock localLock = new ReentrantLock();
l.lock();
int a = 1;
localLock.lock();
if (a == 2) {
l.unlock();
} else {
// do nothing
}
return;
}
}
Proble
Javad
Declar
Search
Consol
Call Hi
Analysi
Debug
0 errors, 12 warnings, 0 others
Description
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
No required execution environment has been set
plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubling(14), High confidence]
tests.NoUnlock$T3.run() does not release lock on all paths [Troubling(12), High confidence]
tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubling(14), High confidence]
Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>
tests.NoUnlock$T3.run() does not release...l paths [Troubling(12), High confidence]

Functional correctness

- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- Formal verification
  - Mathematically prove code matches its specification
- Testing
  - Execute program with select inputs in a controlled environment
- …

Formal verification vs. testing?

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald Knuth, 1977

"Testing shows the presence, not the absence of bugs.”

Edsger W. Dijkstra, 1969

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

Fails if

low + high > MAXINT ($2^{31} - 1$)

Sum overflows to negative value

Comparing strategies for correctness

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)
- Static Analysis
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete
- Proofs (formal verification)
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

Which strategies to use in your project?

Manual testing

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select "Create new Message"	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select "Insert Picture"	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select "Send Message"	Message is correctly sent

- Live system or a testing system?
- How to check output / assertions?
- What are the costs?
- Are bugs reproducible?

Automate testing

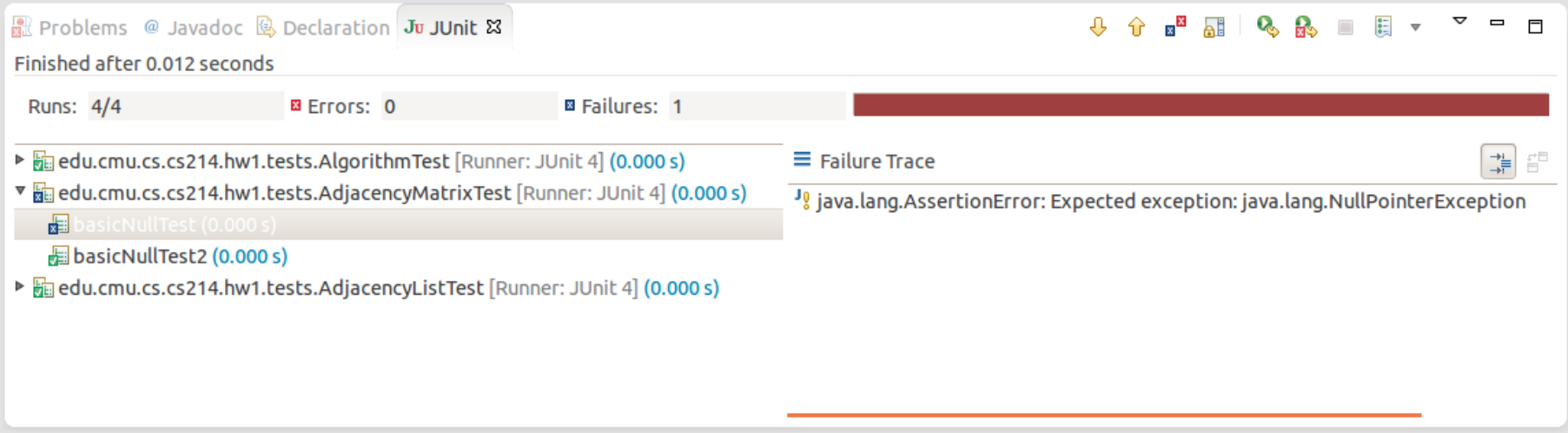
- Execute a program with specific inputs, check output for expected values
- Set up testing infrastructure
- Execute tests regularly
  - After *every* change

Unit testing

- Tests for small units: methods, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment

JUnit

- A popular, easy-to-use, unit-testing framework for Java



A JUnit example

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
        @Test
        public void testSanityTest(){
                Graph g1 = new AdjacencyListGraph(10);
                Vertex s1 = new Vertex("A");
                Vertex s2 = new Vertex("B");
                assertEquals(true, g1.addVertex(s1));
                assertEquals(true, g1.addVertex(s2));
                assertEquals(true, g1.addEdge(s1, s2));
                assertEquals(s2, g1.getNeighbors(s1)[0]);
        }

        @Test
        public void test….

        private int helperMethod…
}
```

Testing, to be continued…