More Java Skill

RoboCode Special Interest Topic

Christian Riess, Eva Eibenberger

Pattern Recognition Lab (Computer Science Dep. 5)

Friedrich-Alexander-University Erlangen-Nuremberg

Themen

- Objekte instanziieren (`new`)
- Listen (fester Länge, `ArrayList`, `HashMap`)
- Ausnahmen (Exceptions)
- Unterklassen
- Schleifensteuerung (`break`, `continue`)
- Entscheidungen bei mehreren Alternativen (`switch`)

Objekte instanziieren

- Roboter sind in Klassen organisiert, so wie alle Java-Programme
- Zur Benutzung einer Klasse muss der Code in den Speicher geladen und initialisiert werden. Dies nennt man „ein Objekt instanziieren".
- Wenn z.B. in einer Arena ein Roboter gegen sich selbst kämpt (zweimal geladen wird), werden aus der einen Roboter-Klasse zwei Objekte instanziiert.

- -> Bei den Robotern kann uns das egal sein, die Arena kümmert sich um die Instanziierung

- Für uns wichtig: Wenn wir innerhalb des Roboters andere Klassen benutzen wollen, müssen wir diese instanziieren.
- Das Schlüsselwort zur Objektinstanziierung ist `new`.
- Beispiel: Instanziierung einer Liste (in der wir z.B. die letzten Koordinaten des Gegners speichern)

Objekte instanziieren

- Das Schlüsselwort zur Objektinstanziierung ist **new**.
- Beispiel: Instanziierung einer Liste (in der wir z.B. die letzten Koordinaten des Gegners speichern)

```
import java.util.ArrayList;

public class RockHard extends AdvancedRobot {
  ArrayList<Integer> lastEnemyX;
  ArrayList<Integer> lastEnemyY;

  public void run() {
    lastEnemyX = new ArrayList<Integer>();
    lastEnemyY = new ArrayList<Integer>();

    lastEnemyX.add(new Integer(-1));
    lastEnemyY.add(new Integer(-1));
    // (..mehr code hier..)
  }
}
```

Hier steht nur: „Der Bezeichner (Symbol) **lastEnemyX** wird irgendwann für eine Liste von Ganzzahlen benutzt".

Mit **new** wird die Liste im Speicher angelegt, und mit dem Symbol **lastEnemyX** verbunden.

Die Liste kann jetzt benutzt werden, wir fügen ein Element hinzu.

Holla! Das eingefügte Integer-Objekt muss auch instanziiert werden, also noch ein **new**!

Wie kann man seine Daten verwalten?

- Listen („Arrays“)
  - Listen fester Größe
  - ArrayList: Listen variabler Größe
  - HashMap („Assoziative Liste“ bzw. „Hash“): Elemente über einen Schlüssel benutzen
- Listen fester Größe:
  - Deklaration: **`<Datentyp>[] Bezeichner`**, Definition: **`new <Klasse>[Listenlänge]`**
  - Wenn <Datentyp> ein primitiver Datentyp (int, char, …) ist: fertig.
  - Wenn <Datentyp> eine Klasse ist, muss jeder Listeneintrag mit **`new`** erzeugt werden

`numbers` wird eine Liste primitiver Typen (**`int`**), **`coordinates`** wird eine Liste von Objekten (**`Point2D.Double`**)

Beide Listen werden mit 10 Einträgen angelegt

`numbers` (weil primitiver Typ) kann man sofort benutzen
Die Einträge von **`coordinates`** (weil Objekte) müssen erst noch mit **`new`** angelegt werden.
(sonst: Bei Benutzung NullPointerException)

```
import java.awt.geom.Point2D;

// (..)
int[] numbers;
Point2D.Double[] coordinates;

numbers = new int[10];
coordinates = new Point2D.Double[10];

for (int i=0; i < numbers.length(); i++) {
  numbers[i] = i*2;
  coordinates[i] = new Point2D.Double();
  coordinates[i].x = i*2;
  coordinates[i].y = i+2;
}
```

Wie kann man seine Daten verwalten?

- ArrayList: Listen Variabler Größe
  - Wichtig: Enthaltenen Datentyp (die Klasse!) mit angeben, im Beispiel **`Double`**
- HashMap: Auf Elemente über ein anderes Element zugreifen
  - Wichtig: Datentyp von Schlüssel und Wert (die Klassen!) mit angeben, im Beispiel String und Integer

```
import java.util.ArrayList;
// (..)
ArrayList<Double> myPIs;
myPIs = new ArrayList<Double>();

myPIs.add(Math.PI);
myPIs.add(new Double(22/7.0));

System.out.println(„PI1: " + myPIs.get(0));
System.out.println(„PI2: " + myPIs.get(1));

// remove provided PI:
myPIs.remove(0);
System.out.println(„PI2: " + myPIs.get(0));
```

```
import java.util.HashMap;
// (..)
HashMap<String, Integer> age;
age = new HashMap<String, Integer>();

String guy1 = new String("gollum");
age.put(new String(„Bilbo"), new Integer(111));
age.put(guy1, new Integer(3000));

if (age.containsKey(guy1)) {
  System.out.println(„age of " + guy1 + „: " +
    age.get(guy1));
}
```

- google „ArrayList java 1.6" -> führt zu der Java-Dokumentation (Version 1.6!)

Fehler: „Ausnahmen“ bzw. „Exceptions“

- Exception: Benachrichtigung bei Durchführung ungültiger Operationen
  - Sprachregelung: Bei Fehler wird eine Exception „geworfen“ (throw), zur Fehlerbehandlung kann man Exceptions „fangen“ (catch)
  - Normale Exceptions muss man fangen
  - „Runtime Exceptions“: Fangen ist optional. Nicht-Fangen führt zu Programmabsturz.

    Die RoboCode-Arena fängt alle Runtime-Exceptions eines Roboters für uns, d.h. die Arena (mit dem Roboter darin) stürzt nicht ab bei falsch programmierten Robotern.

  - Z.B. Division durch 0 wirft eine ArithmeticException
  - Zugriff auf nicht instanziierte Objekte führt zu einer NullPointerException

  - Wir müssen an dieser Stelle nicht wissen, wie man
    - Eigene Exceptions definiert
    - Exceptions fängt
    - Exceptions wirft

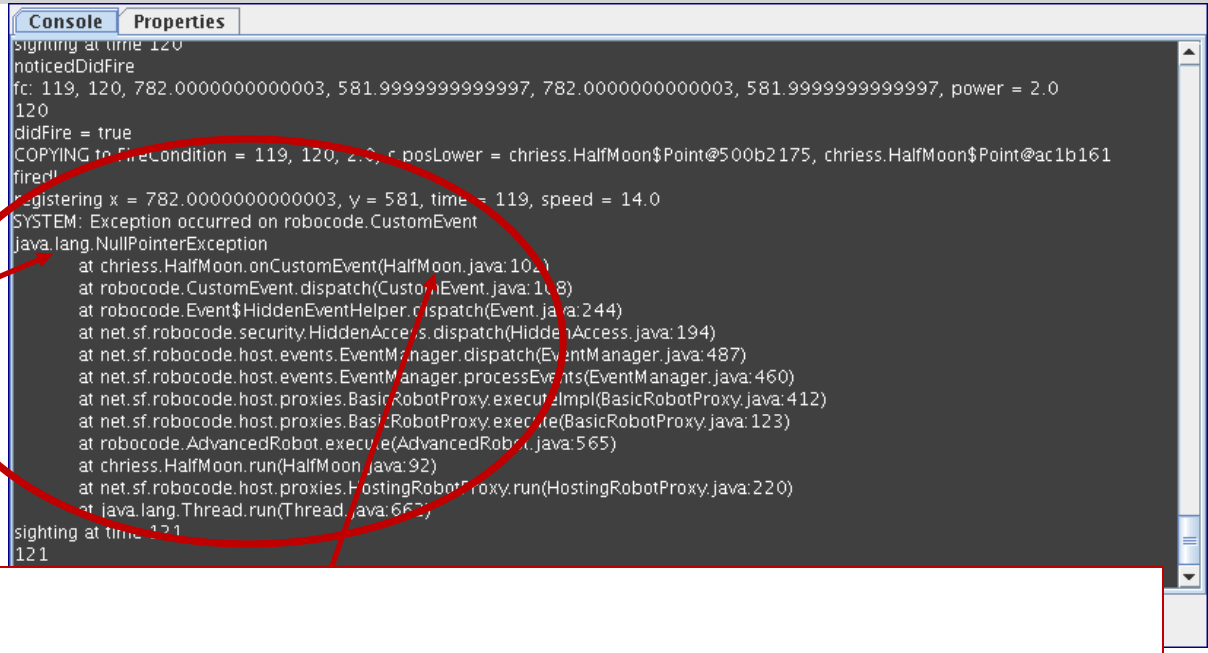
Momentan genügt es zu erkennen, wo und warum der Roboter eine Exception auslöst

Fehler: „Ausnahmen“ bzw. „Exceptions“

- Beispielausgabe
  (in der Roboter-Konsole,
  siehe Debugging-Folien)

Welche Exception?

NullPointerException:
Vermutlich ein **new** vergessen



Wo im Code?

Hier:
Zeile 102,
Datei HalfMoon.java,
in der Methode onCustomEvent
in der Klasse HalfMoon

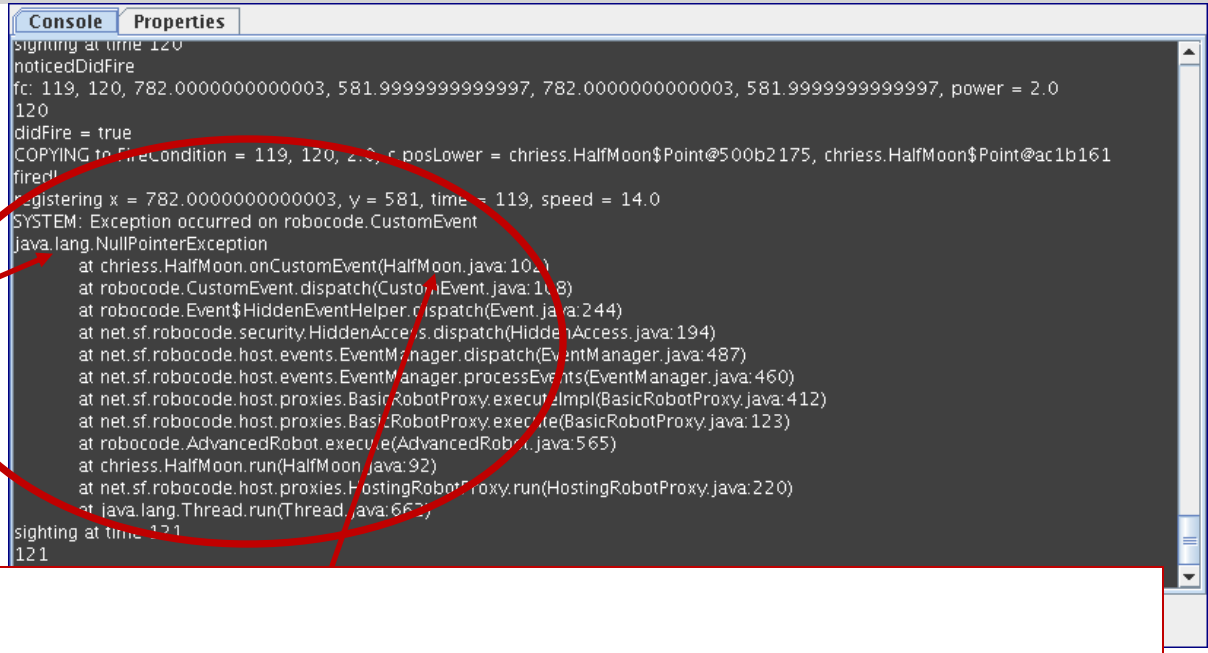
-> dort ist der Fehler aufgetreten (in diesem Fall der Zugriff auf ein nicht initialisiertes Objekt), vielleicht muss man den Fehler an einer anderen Stelle beheben, aber das kann die Java-Umgebung nicht wissen

Exception-Beispiel

- Beispielausgabe
  (in der Roboter-Konsole,
  siehe Debugging-Folien)

Welche Exception?

NullPointerException:
Vermutlich ein **`new`**
vergessen



Wo im Code?

Hier:
Zeile 102,
Datei HalfMoon.java,
in der Methode onCustomEvent
in der Klasse HalfMoon

-> dort ist der Fehler aufgetreten (in diesem Fall der Zugriff auf ein nicht initialisiertes Objekt), vielleicht muss man den Fehler an einer anderen Stelle beheben, aber das kann die Java-Umgebung nicht wissen

Unterklassen zur Datenkapselung

- Um im Code aufzuräumen, kann man einen Teil seiner Daten und Methoden in Unterklassen auslagern
- Unterklassen funktionieren genauso wie Klassen, nur dass sie innerhalb einer anderen Klasse definiert werden
- Instanziierung genauso mit **new** wie andere Klassen auch
- In einer Unterklasse können beispielsweise Informationen über den Gegner gesammelt und ausgewertet werden

File Edit Compiler Window Help

Editing - /home/riess/robocode/robots/chriess/HalfMoon.java *

```
6  import java.util.HashMap;
7
8  public class HalfMoon extends AdvancedRobot
9  {
10     /* ..(lots of other code here).. */
11
12
13     // This is the Enemy subclass, to store opponent's data here.
14     // we use another subclass, LastSighting (not shown in this snippet)
15     // to store radar scan results. The Enemy subclass does prediction on
16     // the next move of the enemy, and checks whether the enemy fired.
17     class Enemy {
18         int lastSightingIndex;
19         final int sightingBacklog = 5;
20         LastSighting[] sightings;
21         double lastCheckedEnergy;
22         boolean didFire;
23         boolean isActive;
24
25         Enemy() {
26             isActive = false;
27             sightingCount = -1;
28             lastSightingIndex = -1;
29             lastCheckedEnergy = -1;
30             sightings = new LastSighting[sightingBacklog];
31             for (int i = 0; i < sightingBacklog; ++i) sightings[i] = new LastSighting();
32             didFire = false;
33         }
34         long getLastScanTime() {
35             if (!sightings[lastSightingIndex].valid) return 0;
36             return sightings[lastSightingIndex].time;
37         }
38     /* ..(lots of other code here).. */
39
40     }
41
```

Line: 7

Schleifenablauf genauer steuern

- Schleifen prüfen Abbruchbedingung einmal pro Durchlauf
- Oft wird der Code viel übersichtlicher, wenn man
  - eine Schleife sofort beenden kann: **`break`**
  - direkt zum nächsten Schleifendurchlauf springen kann: **`continue`**

Beispiel **`break`**:

```
// Addiere Zahlen aus einem Feld. Die Summe
// soll aber nicht groesser als 100 werden.
public int limitedSum(int[] numbers) {
  int sum = 0;
  for (int i=0; i < numbers.length(); i++) {
    if (numbers[i] + sum >= 100) {
      sum = 100;
      break;
    }
    sum += numbers[i];
  }
  return sum;
}
```

Beispiel **`continue`**:

```
// multipliziere Zahlen aus einem Feld. 0 soll
ausgelassen werden (sonst ist Ergebnis 0)
public int multiplyArray(int[] numbers) {
  int product = 1;
  for (int i=0; i < numbers.length(); i++) {
    if (numbers[i] == 0) {
      continue; // überspringe Rest des Blocks
        // (die Multiplikation), gehe direkt
        // zu nächstem i
    }
    product *= numbers[i];
  }
  return product;
}
```

Entscheidungen mit mehreren Alternativen

- Mehrere Alternativen gleichzeitig abfragen mit **`switch`** (geht nur mit ganzen Zahlen!)

Die Varianten (eingeleitet von **`case`**) müssen einzelne ganzzahlige Konstanten sein, gefolgt von einem Doppelpunkt.
Bedingte Ausdrücke sind nicht möglich.
Sobald eine Variante zutrifft, wird Code bis zum nächsten **`break`** ausgeführt.

Die getroffene Entscheidung ist bzgl. der Variable **`grade`**

Durch das fehlende **`break`** wird bei einer 5 sowohl „knapp wars, aber…" ausgegeben, als auch „durchgefallen" (von der 6).

`default` wird immer ausgeführt, es sei denn, es gab zuvor eine passende Bedingung mit anschließendem **`break`**.

```
public boolean printGrade(int grade) {
  switch (grade) {
    case 1: System.out.println(„Sehr gut"); break;
    case 2: System.out.println(„Gut"); break;
    case 3: System.out.println(„Befriedigend"); break;
    case 4: System.out.println(„Ausreichend"); break;
    case 5: System.out.println(„knapp wars, aber…");
    case 6: System.out.println(„durchgefallen");
            break;
    default:
      System.out.println(
          „Ungültige Note im System: " + grade);
      return false;
  }
  return true;
}
```