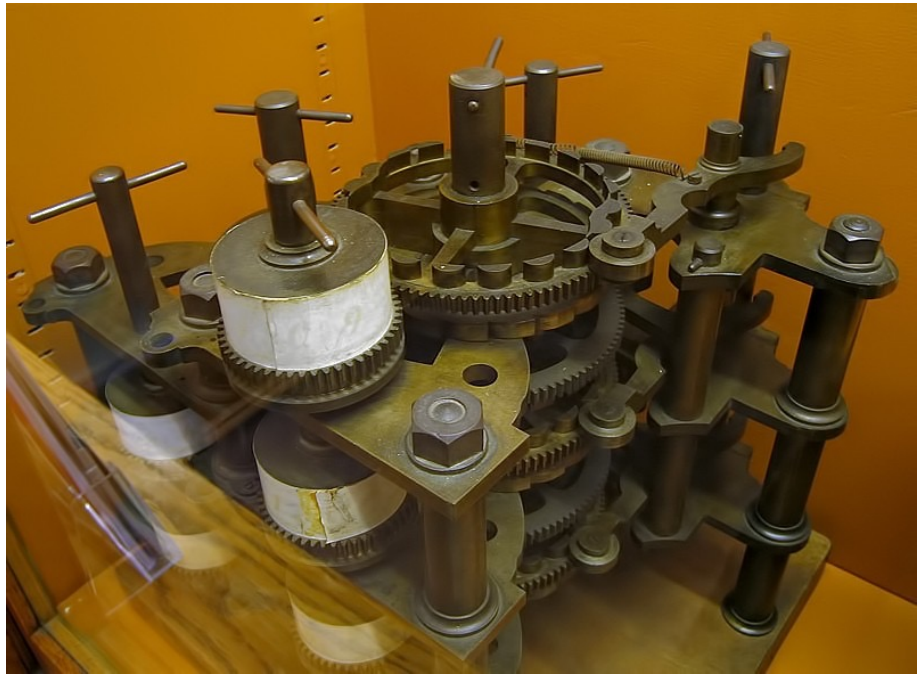
Defensive Programming

October 20, 2022

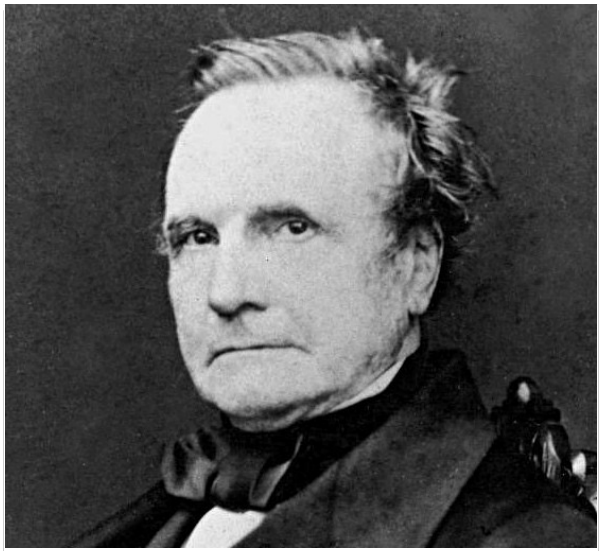
Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Garbage In, Garbage Out



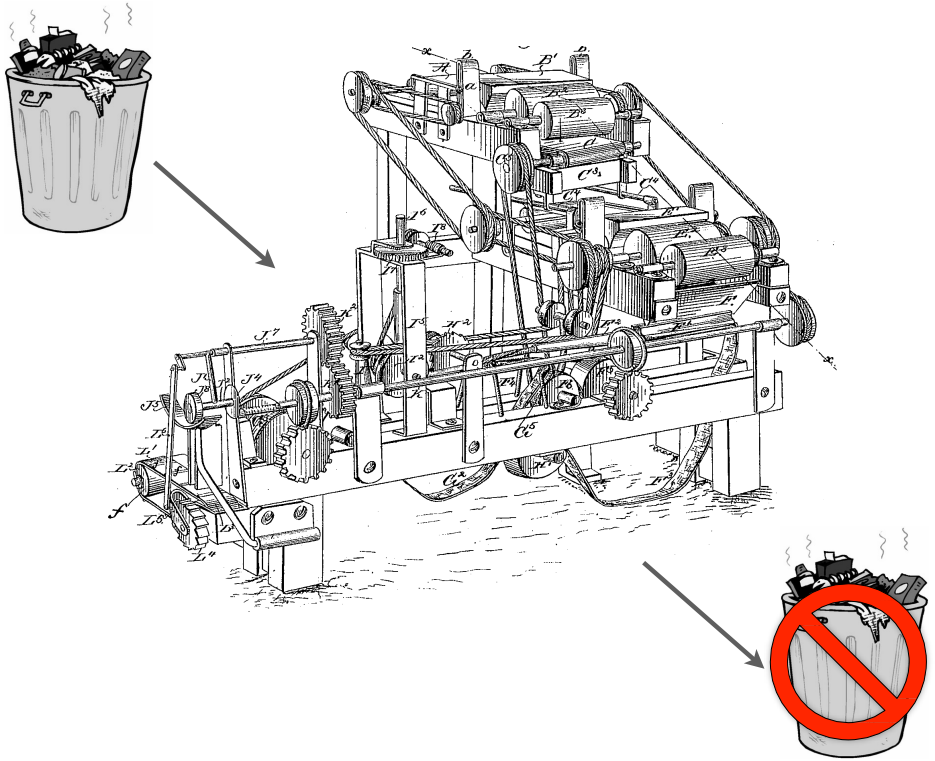
Babbage Difference Engine



On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage
"Passages from the life of a philosopher" (1864)

Our Goal: Garbage In, Non-garbage Out



- Sources of garbage
  - Uncontrollable external sources
  - Method parameters
  - Corrupt state
- Options for dealing with garbage:
  - Garbage in, nothing out
  - Garbage in, error message out
  - Turn garbage input into clean input

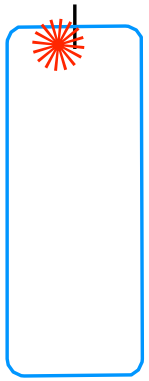
Dealing with Invalid Inputs

```
/**
 *Returns a BigInteger whose value is (this mod m). This method
 *differs from the remainder method in that it always returns a
 *nonnegative BigInteger.
 *
 *@param   m the modulus, which must be positive.
 *@return this mod m.
 *@throws IllegalArgumentException if m <= 0.
 */
public BigInteger mod(BigInteger m) {
   if (m.signum() <= 0) {
      throw new IllegalArgumentException("Modulus not positive");
   }

   // …
}
```

- Check inputs for validity
- Things to check
  - Reference is not null
  - Input param values are within valid range
  - Stream status
  - File access type: read, write, both
- Throw exception if bad input
  - Document pre-conditions of a method “contract”

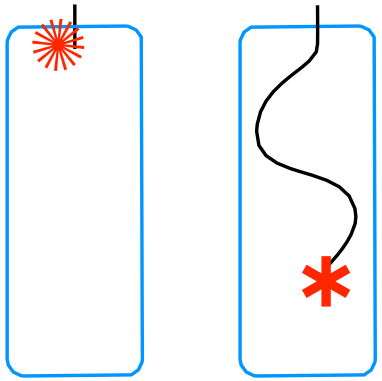
Things That Can Go Wrong

- No check => garbage out

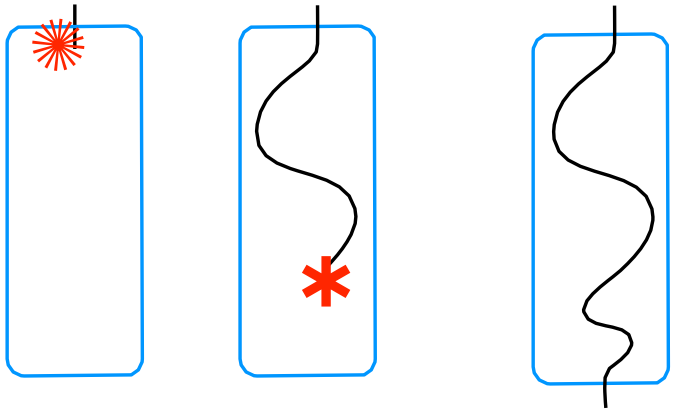


Things That Can Go Wrong

- No check => garbage out
  - Fail with confusing exception later

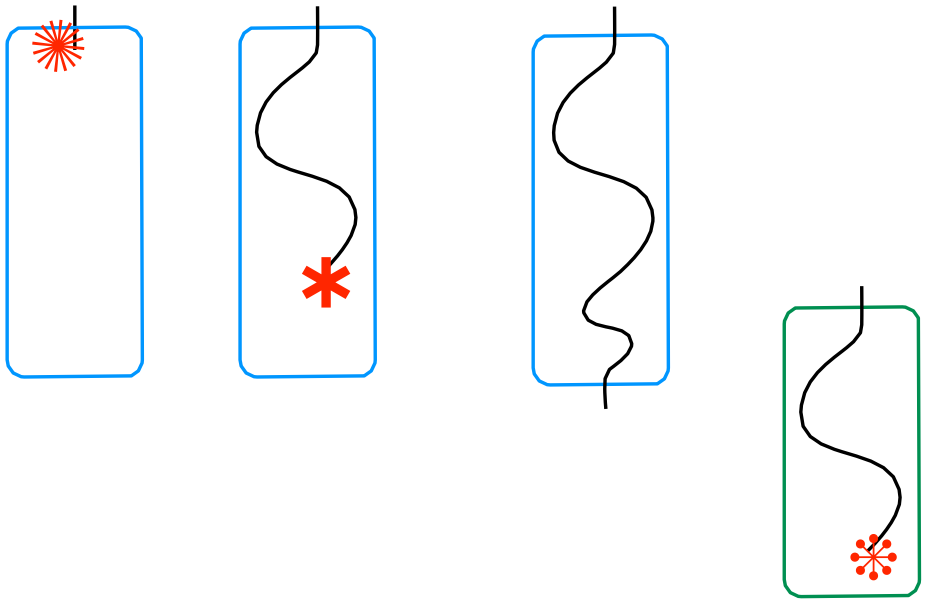


Things That Can Go Wrong



- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value

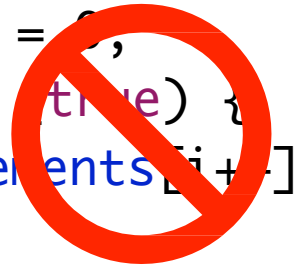
Things That Can Go Wrong



- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value
  - Return normally but compromise some other obj

Things That Can Go Wrong

```
try {
    int i = 0;
    while (true) {
        elements[i++].operation();
    }
} catch (ArrayIndexOutOfBoundsException e) { }
```



```
for (Element el : elements) {
    el.operation();
}
```

- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value
  - Return normally but compromise some other obj
- Exceptions <= exceptional situations
  - Do not abuse the exception mechanism

Exceptions To The Rule

```
private void sortList(List<Object> objects) {
   // ...
   Collections.sort(objects, new MyComparator());
}
```

- Avoid checking when…
  - Validity check is too expensive/impractical and it is implicitly done anyway

Preserve Abstraction

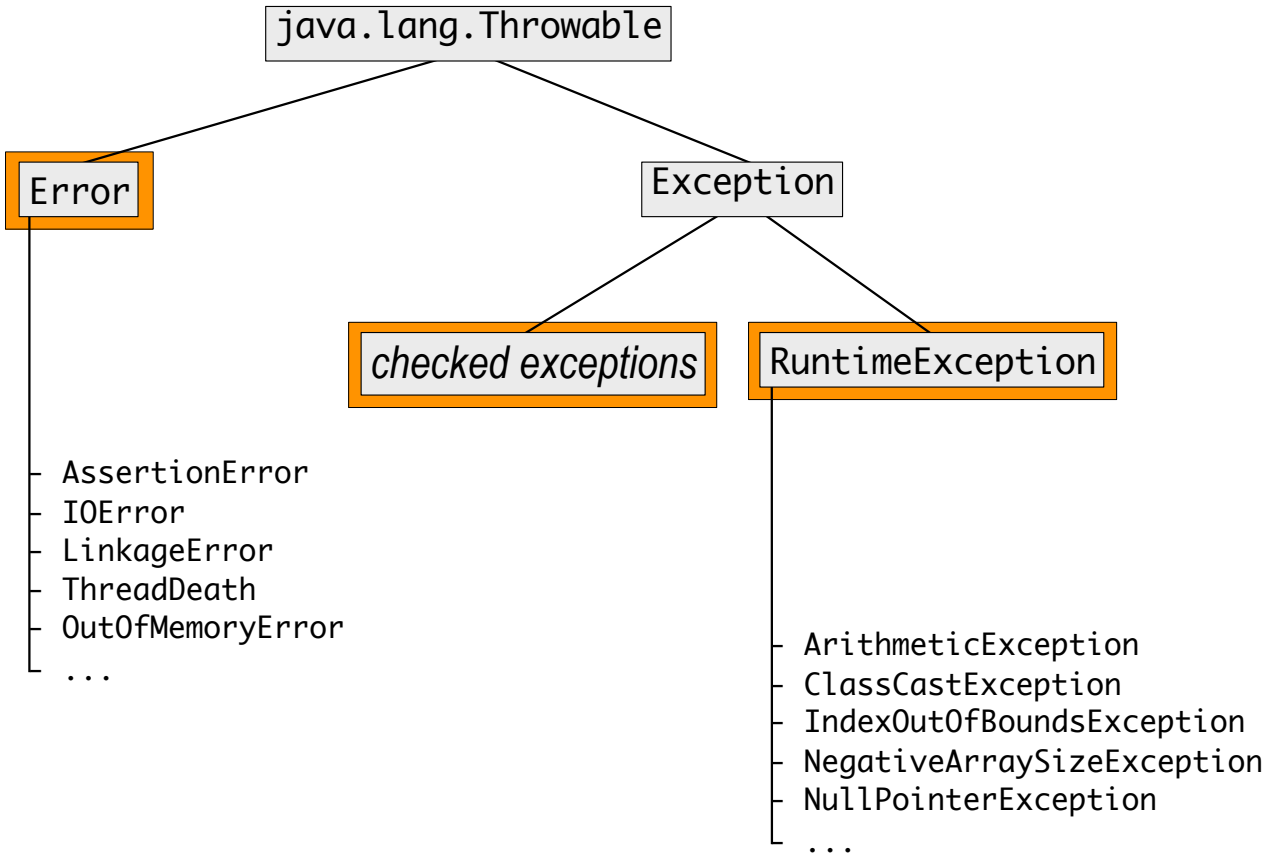
```
User me() throws NotLoggedInException {
    // ...
}

User me() throws NotLoggedInException, IOException {
    // ...
}

User me() throws NotLoggedInException, UserDBException {
    // ...
}
```

- Throw at right level of abstraction
- Aim for informative exceptions
  - Include the context of the condition
  - Put yourself in the shoes of the catcher

```
User search(String keyword) throws NotFoundException {
    // ...
}
```

- Is an exception needed?

Choosing The Right Exception



- Checked Exceptions
  - Exceptional but recoverable conditions
  - Require try-catch

Fixing Invalid Data

- Ways to replace/fix invalid data
  - Use the previously used value
  - Use a neutral value
  - Use the next valid entry / element
  - Find closest legal value
- The key trade-off…
  - Cost of throwing exception
    vs.
    cost of getting the input wrong
    x probability of it being wrong

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a lowercase vowel");
while (!input.hasNext("[aeiou]")) {
   System.out.println("Not a vowel; skipping");
   input.next();
}
processVowel(input.next());
```

What Are Known Truths?

days in year $\overset{?}{\geq}$ 365

$59 \overset{?}{\geq} \text{seconds} \overset{?}{\geq} 0$

```
int *p,*r;
...some code
...more code
*p = 2;
*r = 3;
assert(*p + *r == 5);
```

```
x+1 > x
```

Assertions Check Assumptions

```
assert invariant: details
```

- Checks for “impossible” conditions
- Catch bugs during development
  - Mismatched interface assumptions
  - Errors caused by modified code
- Assertions serve as documentation
  - Insurance against future code evolution
- Sanity checks for your program
  - Check parameters of non-public methods
  - Verify code invariants
  - Fulfills a subset of the audit methods’ role

Assertions Check Assumptions

```
public class HashMap<K, V>
extends AbstractMap<K, V>
implements Map<K, V>, Cloneable, Serializable {
    // ...
    public HashMap(int initialCapacity, float loadFactor) {
       if (initialCapacity < 0) {
          throw new IllegalArgumentException("Illegal initial capacity: "
                + initialCapacity);
       }
       if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
          throw new IllegalArgumentException("Illegal load factor: "
                + loadFactor);
       }
       // ...
   }
   // ...

   void resize(int newCapacity) {
       assert newCapacity>table.length || table.length==MAXIMUM_CAPACITY;
       // ...
   }
```

Code Invariants

- “Invariant” means “always true”
  - Property that is purported to always hold
  - Generally restricted to a certain portion of code
  - Examples: loop invariant, class invariant

- Use asserts to enforce invariants

Loop Invariant

{ P }
while (b) S;
{ Q }

- Find an invariant, LI, such that
  1. P ⇒ LI // true initially
  2. { LI & b } S { LI } // true if the loop executes once
  3. (LI & ¬b) ⇒ Q // establishes the postcondition

- It is sufficient to know that if loop terminates, Q will hold. Finding the invariant is the key to reasoning about loops.

Loop Invariant Example

```
// assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

Loop Invariant Example

```
// assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

A suitable invariant: LI = x ≥ y

1. x ≥ 0 & y = 0 ⇒ LI // true initially
2. { LI & x ≠ y } y = y+1; { LI } // true if the loop executes once
3. (LI & ¬(x ≠ y)) ⇒ x = y // establishes the postcondition

```
enum Suit {
   CLUBS, DIAMONDS, HEARTS, SPADES;
}
// ...
switch(suit) {
 case CLUBS:
      // ...
      break;
 case DIAMONDS:
      // ...
      break;
 case HEARTS:
      // ...
      break;
 case SPADES:
      // ...
      break;
 default:
     throw new AssertionError(suit);
}
 // ...
```

Code Invariants

- "Invariant" means "always true"
  - Property that is purported to always hold
  - Generally restricted to a certain portion of code
  - Examples: loop invariant, class invariant
- Use asserts to enforce invariants
- Use assertions to catch the impossible
  - E.g., empty default statements

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
        this.start = dateStart;
        this.end = dateEnd;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
        this.start = dateStart;
        this.end = dateEnd;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
p.end().setYear(78);
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return (Date) start.clone();
    }
    public Date end() {
        return (Date) end.clone();
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
p.end().setYear(78);
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return (Date) start.clone();
    }
    public Date end() {
        return (Date) end.clone();
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
p.end().setYear(78);
```

Defensive Programming

- Check inputs
  - Can use exceptions for public methods, assertions for non-public ones
  - Discard bad inputs, repair bad inputs
- Document assumptions
  - Cannot control outside world, but can be explicit about what we assume about it
- Check code invariants
- Employ defensive copying