# תכנות משחק ניחוש מספר

הבה ניגש ישר לעניין ונבנה פרוייקט מעשי בראסט! פרק זה יציג בפניכם כמה מושגי ראסט נפוצים באמצעות שימושם בתוכנית אמיתית. תלמדו על מילות המפתח `let`,`match`, מתודות, פונקציות משויכות, מכולות חיצוניות, ועוד! בפרקים הבאים נסקור רעיונות אלה בפירוט רב יותר; בפרק זה נתמקד בתירגול היסודות.

אנו ניישם בעיה קלאסית למתחילים בעולם תכנות: משחק ניחוש מספר. זהו תיאור המשחק: התוכנה תגריל מספר אקראי שלם בין 1 ל-100. אז היא תורה לשחקן להקליד את ניחושו. לאחר שהניחוש הוקלד, התוכנית תציין אם הניחוש נמוך מידי או גבוה מידי. אם הניחוש מדוייק, המשחק ידפיס הודעת ניצחון ויסתיים.

## התקנת פרוייקט חדש

על מנת להתקין פרוייקט חדש נווטו אל ספריית הפרוייקטים \*projects\* אשר יצרתם בפרק 1, וצרו פרוייקט חדש כך:

```console

cargo new guessing\_game $

cd guessing\_game $

```

הפקודה הראשונה, `cargo new`, מקבלת את שם הפרוייקט (`guessing\_game`) כארגומנט הראשון. הפקודה השניה נכנסת לתיקייה החדשה של הפרוייקט.

התבוננו בקובץ \*Cargo.toml\* שיוצר עבורכם אוטומטית:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial

rm -rf no-listing-01-cargo-new

cargo new no-listing-01-cargo-new --name guessing\_game

cd no-listing-01-cargo-new

cargo run > output.txt 2>&1

cd ../../..

-->

<span class="filename">Filename: Cargo.toml</span>

```toml

{{#include ../listings/ch02-guessing-game-tutorial/no-listing-01-cargo-new/Cargo.toml}}

```

כפי שראיתם בפרק 1, `cargo new` מכין תוכנת "!Hello, world" עבורכם. תוכלו לראות זאת ע"י בדיקת הקובץ \*src/main.rs\*:

<span class="filename">Filename: src/main.rs</span>

```rust

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/no-listing-01-cargo-new/src/main.rs}}

```

עכשיו, הבה נקמפל את תוכנית "!Hello, world" ונריץ אותה תוך שימוש בפקודה אחת, דהיינו `cargo run`:

```console

{{#include ../listings/ch02-guessing-game-tutorial/no-listing-01-cargo-new/output.txt}}

```

הפקודה `run` שימושית כאשר רוצים להתקדם בפרוייקט במהירות, כפי שנעשה בפיתוח המשחק. כך ניתן לבדוק בזריזות כל איטרצית (קרי, שלב) פיתוח איטרציה בטרם עוברים לשלב הבא.

פתחו שוב את הקובץ \*src/main.rs\*. כל הקוד שלכם ייכתב בקובץ זה.

## עיבוד ניחוש

החלק הראשון של תכנית משחק הניחוש יבקש מהמשתמש קלט, יעבד את הקלט, ויבדוק האם הקלט התקבל בתצורה המצופה. כדי להתחיל, נאפשר לשחקן להקליד ניחוש. כתוב את הקוד מרשימה 2-1 לקובץ \*src/main.rs\*.

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:all}}

```

<span class="caption">רשימה 2-1: קוד שמקבל ניחוש מהמשתמש ומדפיס אותו</span>

קוד זה מכיל אלמנטים רבים של השפה, עליהם נעבור כעת שורה אחר שורה. על מנת לקבל קלט מהמשתמש, ואז להדפיס את התוצאה כפלט, אנו צריכים לכלול את ספריית הקלט/פלט `io` למרחב (scope)העבודה הנוכחי. הספריה `io` מגיעה מהספריה הסטנדרטית, הידועה בשם `std`:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:io}}

```

כברירת מחדל, לראסט יש אוסף של עצמים המוגדרים בספריה הסטנדרטית שהיא מכניסה למתחם של כל תוכנית. אוסף זה נקרא \*הפרליוד\* (prelude), ואתם יכולים לראות את כל תוכנו [בתיעוד של הספריה הסטנדרטית][prelude].

אם טיפוס מסויים שאתם מעוניינים להשתמש בו לא נמצא בפרליוד, תוכלו להכניס אותו למתחם בצורה מפורשת באמצעות פקודת `use`. שימוש בספריה `std::io` מקנה מספר יכולות שימושיות, כולל היכולת לקרוא קלט מהמשתמש.

כפי שראיתם בפרק 1, הפונקציה `main` היא נקודת הכניסה לתוכנית:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:main}}

```

הסימון `fn` מצהיר על פונקציה חדשה; הסוגריים, `()`, מציינים העדר פרמטרים; והסוגר המסולסל, `}`, מתחיל את גוף הפונקציה.

כפי שלמדתם בפרק 1, `println!` הוא מאקרו המדפיס מחרוזת למסך:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:print}}

```

קוד זה מדפיס הודעה פתיחה המציגה את המשחק ומבקשת קלט מהמשתמש.

### אכסון ערכים למשתנים

כעת ניצור \*משתנה\* לאכסון הקלט שהתקבל מהמשתמש, כך:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:string}}

```

עכשיו התכנית מתחילה להיות מעניינת! בשורה קצרה זו מתרחש די הרבה. אנו משתמשים בפקודה `let` בכדי ליצור משתנה. הינה דוגמא נוספת:

```rust,ignore

let apples = 5;

```

שורה זו יוצרת משתנה חדש בשם `apples` אותו היא מקשרת לערך 5. משתנים בראסט, כברירת מחדל, הם מנועי-שינוי (immutable), . כלומר, ברגע שמשייכים להם ערך, ערך זה אינו ניתן עוד לשינוי. אנו נדון בנושא זה בפירוט כשנגיע לסעיף ["משתנים וברות-שינוי"][variables-and-mutability]<!-- ignore -->

בפרק 3. על מנת ליצור משתנה בר-שינוי, יש להוסיף את מילת המפתח`mut` לפני שם המשתנה:

```rust,ignore

let apples = 5; // immutable

let mut bananas = 5; // mutable

```

> הערה: הסימן `//` משמש לציין התחלת הערה (comment) הנמשכת עד סוף השורה. ראסט מתעלמת מכל מה שמופיע בהערות. אנו נדון בהערות בפירוט [בפרק 3][comments]<!-- ignore -->.

בחזרה למשחק ניחוש המספר. עכשיו אתם יודעים שהשורה `let mut guess` תיצור משתנה מנוע-שינוי חדש בשם `guess`. סימן השוויון (`=`) אומר לראסט שבכוונתנו לקשור דבר מה למשתנה. באגף ימין של סימן השוויון נמצא הערך אליו יקושר המשתנה `guess`. ערך זה הוא תוצאת הקריאה ל- `String::new`, פונקציה המחזירה מופע (instance) חדש של `String`. [`String`][string]<!-- ignore --> הוא טיפוס מחרוזת המסופק ע"י הספריה הסטנדרטית, והוא מייצג פיסת טקסט בגודל בלתי-קבוע, בקידוד UTF-8.

הסימון `::` בשורה `::new` מציין ש-`new` היא פונקציה משויכת מטיפוס `String`. \*פונקציה משויכת \* היא פונקציה המיושמת על טיפוס, במקרה זה הטיפוס הוא `String`. הפונקציה `new` יוצרת מחזורת ריקה חדשה. פונקציות בשם `new` מייושמות עבור טיפוסים רבים כי זהו שם נפוץ עבור פונקציה שמייצרת ערך מסוג נתון.

בקיצור, מלוא מובן השורה `let mut guess = String::new();` הוא יצירת משתנה בר-שינוי המקושר למופע ריק חדש של `String`. זה די הרבה!

### איך לקבל קלט מהמשתמש

כזכור, כללנו במרחב העבודה פונקציונליות קלט/פלט מהספריה הסטנדרטית באמצעות השורה `use std::io;` המתנוסת בראש התכנית. עכשיו נקרא לפונקציה `stdin` מהמודול `io`, שתאפר לנו לטפל בקלט מהמשתמש:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:read}}

```

אלמלא יבאנו את הספריה `io` באמצעות `use std::io;` בתחילת התוכנית עדיין היינו יכולים להשתמש בפונקציה ע"י כתיבת הקריאה כ-`std::io::stdin`. הפונקציה `stdin` מחזירה מופע של [`std::io:Stdin`][iostdin]<!-- ignore -->, שהוא טיפוס המייצג מנהל לקלט הסטנדרטי של הטרמינל.

השורה הבאה, `.read\_line(&mut guess)`, קוראת למתודה [`read\_line`][read\_line]על מנהל הקלט הסטנדרטי בכדי לקבל קלט מהמשתמש. אנחנו גם מעבירים את `&mut guess` כארגומנט למתודה `read\_line` בכדי לומר למתודה באיזה משתנה לאחסן את הקלט שהתקבל. התפקיד המלא של `read\_line` הוא לקחת כל מה שהמשתמש מקליד לקלט הסטנדרטי ולהוסיף אותו למחרוזת (מבלי למחוק את תוכנה). לכן אנו מעבירים מחרוזת זו כארגומנט. ארגומנט מחרוזת זה חייב להיות בר-שינוי כדי שהמתודה תוכל לשנות את תוכנו.

הסימון `&` מציין שהארגומנט הוא הפניה (reference), דבר המאפשר לחלקים שונים של הקוד לגשת לאותה פיסת מידע ללא צורך ליצור עותקים שלו בזיכרון. הפניות הם נושא מורכב, ואחד היתרונות הגדולים של ראסט הוא שהיא מאפשרת שימוש בטוח ופשוט בהפניות. אין צורך להכביר בפרטים אלו כדי לסיים את התוכנית. לעת עתה, כל שאתם צריכים לדעת הוא שהפניות, כמו משתנים, הם מנועי-שינוי כברירת מחדל. לכן, יש לכתוב `&mut guess` ולא `&guess` כל מנת להתייחס להפניה ברת-שינוי. (פרק 4 ירחיב על נושא ההפניות.)

<!-- Old heading. Do not remove or links may break. -->

<a id="handling-potential-failure-with-the-result-type"></a>

### ניהול כשלונות באמצעות `Result`

אנחנו עדיין מתמקדים בשורת קוד זו ובה נסתכל כעט על שורת הטקסט השלישית, אולם שימו לב שהיא עדיין חלק משורת קוד יחידה. החלק הבא הוא המתודה:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:expect}}

```

היינו יכולים לכתוב קוד זה גם כך:

```rust,ignore

io::stdin().read\_line(&mut guess).expect("Failed to read line");

```

אבל, שורה ארוכה כזו קשה לקריאה, ולכן עדיף לפרק אותה לחלקים קצרים יותר. שימוש מושכל בשורות חדשות ורווחים מאוד מסייע לפצל שורות ארוכות כאשר משתשמים בנוטצית ה-`.method\_name()`. הבה נבין מה שורת קוד זו עושה.

כפי שהוזכר קודם, המתודה `read\_line` מאחסנת את כל מה שהמשתמש מקליד לתוך המחרוזת המועברת אליה. בנוסף, היא גם מחזירה ערך מטיפוס [`Result`][result]. זהו טיפוס מסוג [\*מבחר\*][enums] (enumeration)<!-- ignore --> ז"א טיפוס שיכול להיות באחד מכמה מצבים. כל מצב אפשרי כזה נקרא \*ווריאנט\*.

[פרק 6][enums]<!-- ignore --> יעסוק במבחרים ביתר פירוט. מטרת טיפוסי `Result` אלה היא לקודד מידע לניהול שגיאות.

הווריאנטים האפשריים של `Result` הם `Ok` ו-`Err`. הווריאנט `Ok` מציין פעולה שבוצעה בהצלחה, ובתוך ה-`Ok` נמצא הערך שהופק בהצלחה. הווריאנט `Err` משמעו שהפעולה נכשלה, ואז `Err` מכיל מידע אודות כישלון הפעולה.

על הערכים של הטיפוס `Result`, כמו כל טיפוס אחד, מוגדרות כל מיני מתודות. למופע של `Result` יש [מתודת `expect`][expect] <!-- ignore --> שניתן לקרוא לה. אם מופע זה של `Result` הוא הווריאנט `Err`, אז `Expect` תגרום לתכנית לקרוס ותוצג ההודעה שמעבירים ל-`Expect` כארגומנט. אם מתודת ה-`read\_line` מחזירה `Err`, זה כנראה בשל שגיאה שמקורה במערכת ההפעלה. אם מופע זה של `Result` הוא הווריאנט `Ok`, אז `Expect` תיקח את הערך המוחזר ש-`Ok` מחזיק ותחזיר רק את הערך הזה על מנת שתוכלו להשתמש בו. במקרה זה, הערך המוחזר הוא מספר הבייטים בקלט המשתמש.

במידה ולא תקראו ל-`expect`, התוכנית כן תעבור קומפילציה אבל תתקבל הודעת האזהרה:

```console

{{#include ../listings/ch02-guessing-game-tutorial/no-listing-02-without-expect/output.txt}}

```

ראסט מזהירה אתכם שלא עשיתם שימוש בערך שב-`Result` שהוחזר מהקריאה ל-`read\_line`, ולכן לא טיפלתם בשגיאה אפשרית.

הדרך הנכונה להיפתר מהודעת האזהרה היא לכתוב קוד שכן מטפל בשגיאות, אבל במקרה שלנו אנו רוצים שהתכנית תקרוס במידה ויש בעיה, כך ששימוש ב-`expect` מספיק. עוד על החלמה משגיאות תלמדו [בפרק 9][recover]<!-- ignore -->.

### הדפסת ערכים עם `!println` ומצייני מקום

למעט מהסוגר המסולסל הסוגר נותר לדון רק בעוד שורה אחת בקוד:

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-01/src/main.rs:print\_guess}}

```

שורה זו מדפיסה את המחרוזת שבשלב זה מכילה את הקלט מהמשתמש. הסוגריים המסולסלים `{}` מהווים מציין מקום: חשבו על `{}` כעל צבתות קטנות של סרטן שאוחזות בערך כלשהו. כשמדפיסים את הערך של משתנה, ניתן לשים את שם המשתנה בתוך הסוגריים המסולסלים. כשמדפיסים תוצאה של הערכה של ביטוי, מקמו סוגריים מסולסלים ריקים במחרוזת הפורמט, ואז, אחרי מחרוזת הפורמט, ברשימה מופרדת ע"י פסיקים, ספקו את הביטויים להדפסה בכל אחד מהסוגריים מצייני המקום, באותו סדר. הדפסה של משתנה ושל תוצאה של ביטוי בקריאה יחידה ל-`println!` תראה כך:

```rust

let x = 5;

let y = 10;

println!("x = {x} and y + 2 = {}", y + 2);

```

קוד זה ידפיס `x = 5 and y + 2 = 12`.

### בדיקת החלק הראשון

הבה נבדוק את החלק הראשון של משחק הניחוש. הריצו אותו באמצעות `cargo run`:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-01/

cargo clean

cargo run

input 6 -->

```console

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 6.44s

Running `target/debug/guessing\_game`

Guess the number!

Please input your guess.

6

You guessed: 6

```

בשלב זה החלק הראשון של המשחק מוכן: אנחנו מקבלים קלט מהמשתמש ומדפיסים אותו.

## ייצור מספר סודי

עכשיו עלינו לייצר מספר סודי שהמשתמש ייצטרך לנסות ולנחש. על המספר הסודי להשתנות בכל הרצה של המשחק, כך שיהיה מהנה לשחק יותר מפעם אחת. אנחנו נשתמש במספר אקראי בין 1 ל-100 כדי שהמשחק לא יהיה יותר מידי קשה. הספריה הסטנדרטית של ראסט עדיין אינה כוללת פונקציונאליות לייצור ושימוש במספרים אקראיים. אבל צוות ראסט מספק את [המכולה`rand`][randcrate] שכן מספקת פונקציונאליות זו.

### שימוש במכולה להרחבת פונקציונאליות

זכרו שמכולה היא אוסף של קבצי קוד ראסט. הפרוייקט שאנו בונים הוא \*מכולה בינארית\*, והיא ניתנת להרצה. המכולה `rand` היא \*מכולת ספריה\* שמכילה קוד המיועד לשימוש בתוכניות אחרות ולא ניתן להריץ אותו לבדו.

כשזה מגיע לניהול מכולות חיצוניות, קארגו באמת עושה עבודה מבריקה. לפני שניתן יהיה לכתוב קוד שמשתמש ב-`rand` יש לשנות את הקובץ \*Cargo.toml\* כדי שיכיל את המכולה `rand` כתלות. פתחו קובץ זה כעת והוסיפו את השורה הבאה בתחתית הקובץ, מתחת לכותר הסעיף `[dependencies]` שקארגו יצר עבורכם. וודאו להוסיף את `rand` בדיוק כמופיע כאן, יחד עם מספר הגרסה. אחרת דוגמאות הקוד להלן עלולות שלא לעבוד:

<!-- When updating the version of `rand` used, also update the version of

`rand` used in these files so they all match:

\* ch07-04-bringing-paths-into-scope-with-the-use-keyword.md

\* ch14-03-cargo-workspaces.md

-->

<span class="filename">Filename: Cargo.toml</span>

```toml

{{#include ../listings/ch02-guessing-game-tutorial/listing-02-02/Cargo.toml:8:}}

```

בקובץ \*Cargo.toml\* כל מה שמופיע תחת כותרת של סעיף ועד לתחילת הסעיף הבא הינו חלק מהסעיף. בסעיף `[dependencies]` אתם אומרים לקארגו באלו מכולות חיצוניות הפרוייקט שלכם תלוי ואלו גרסאות של מכולות אלה נדרשות. במקרה זה אנו מציינים את המכולה `rand` עם מציין הגרסה הסמנטי `0.8.5`. קארגו מבין [שפת גרסאות סמנטית][semver]<!-- ignore --> (לפעמים קוראים לזה \*SemVer\*), שזה סטנדרט לכתיבת מספרי גרסאות. המציין `0.8.5` הוא למעשה קיצור ל-`^0.8.5`, שמשמעו הוא כל גרסה החל מגרסה 0.8.5 ועד 0.9.0 (לא כולל).

מבחינת קארגו, משמעות הדבר היא שגרסאות אלה חושפות למתכנת ממשקי תכנות אפליקציה (API) פומביים המותאמים לגרסה 0.8.5, ומפרט (specification) זה מבטיח שימוש בטלאי העדכני ביותר שעדיין תואם את הקוד המופיע בפרק זה. גרסאות 0.9.0 ומעלה עשויות שלא לממש את אותו הממשק עליו מסתמכות הדוגמאות הבאות, ולכן עלינו להקפיד על מציין גרסה כמתואר בפסקה הקודמת.

עכשיו, מבלי לשנות דבר בקוד, הבה נבנה את הפרוייקט, כפי שמוצג ברשימה 2-2.

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-02/

rm Cargo.lock

cargo clean

cargo build -->

```console

$ cargo build

Updating crates.io index

Downloaded rand v0.8.5

Downloaded libc v0.2.127

Downloaded getrandom v0.2.7

Downloaded cfg-if v1.0.0

Downloaded ppv-lite86 v0.2.16

Downloaded rand\_chacha v0.3.1

Downloaded rand\_core v0.6.3

Compiling libc v0.2.127

Compiling getrandom v0.2.7

Compiling cfg-if v1.0.0

Compiling ppv-lite86 v0.2.16

Compiling rand\_core v0.6.3

Compiling rand\_chacha v0.3.1

Compiling rand v0.8.5

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 2.53s

```

<span class="caption">רשימה 2-2: הפלט המתקבל מהרצת `cargo build` לאחר הוספת המכולה `rand` כתלות</span>

ייתכן שתראו מספרי גרסאות שונים (אבל מובטח שהן תואמות את הקוד, הודות ל-SemVer!) ושורות שונות (בתלות במערכת ההפעלה), ויתכן גם שהשורות יופיעו בסדר שונה.

כאשר מתקינים תלות חיצונית, קארגו מביא את הגרסאות העדכניות ביותר של כל מה שהתלות צריכה ישירות מ- \*registry\*, שהוא עותק של הדאטה מ-[Crates.io][cratesio]. Crates.io הוא המקום בוא חברי קהילת ראסט מפרסמים את פרוייקטי הראסט שלכם כקוד פתוח לשימושם של אחרים.

לאחר עדכון הרישום (registry), קארגו בודק את סעיף ה-`[dependencies]` ומוריד את המכולות הרשומות שלא הורדו עדיין. במקרה זה, למרות שרק הוספנו את `rand` כתלות, קארגו הוריד מכולות אחרות להן `rand` זקוק. לאחר הורדת המכולות ראסט מקמפלת אותן ואז, משהתלותות זמינות, מקמפלת את הפרוייקט כולו.

אם מייד תריצו `cargo build` שוב, ללא שום שינוי, כל שתקבלו יהיה שורת ה- `Finished`. קארגו מזהה שהוא כבר הוריד וקימפל את התלותות, ושלא ביצעתם אף שינוי בקשר אליהן בקובץ \*Cargo.toml\*. קארגו גם יודע שלא שיניתם דבר בקוד, ועל כן הוא לא מבצע שום קומפילציה. כיוון שאין דבר לעשות, קארגו פשוט מסיים את פעולתו.

אם תפתחו את הקובץ \*src/main.rs\*, תבצעו שינוי כלשהו, ואז תשמרו את הקובץ ותבנו שוב, כל שתראו הוא את שתי שורות הפלט:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-02/

touch src/main.rs

cargo build -->

```console

$ cargo build

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs

```

שורות אלה מראות שקארגו רק עדכן את הבניה בתוספת השינוי הקטן לקובץ \*src/main.rs\*. עם זאת, התלותות שלכם לא השתנו, ולכן קארגו יודע שהוא יכול להשתמש שוב בכל מה שהוא כבר הוריד וקימפל, ואין צורך לעדכן אף תלות חיצונית.

#### הבטחת בניות ניתנות-לשחזור באמצעות הקובץ \*Cargo.lock\*

בקארגו יש מנגנון שמוודא שניתן לבנות מחדש את אותו מוצר מטרה בכל פעם שאתם, או אחרים, בונים את הקוד: קארגו ישתמש אך ורק בגרסאות של התלותות שאתם ציינתם, עד שתחליטו אחרת. לדוגמא, נניח ששבוע הבא גרסה 0.8.6 של המכולה `rand` תהיה זמינה, ושגרסה זו כוללת תיקון חשוב של באג, אבל היא גם כוללת רגרסיה שתקלקל את הקוד שלכם. על מנת לנהל מצב זה, קארגו יוצר את הקובץ \*Cargo.lock\* במהלך הריצה הראשונה של `cargo build`, כך שעכשיו יש לנו את הקובץ הזה בתיקייה \*guessing\_game\*.

כאשר בונים את הפרוייקט בפעם הראשונה, קארגו מפענח מהן כל הגרסאות של התלותות שעונות לקריטריונים ואז כותב מידע זה לקובץ \*Cargo.lock\*. כשתבנו את הפרוייקט שלכם שוב בעתיד, קארגו יראה שהקובץ \*Cargo.lock\* קיים ולכן ישתמש בכל הגרסאות המצויינות שם ויימנע מפיענוח-מחדש של הגרסאות. מנגנון זה מבטיח בניות ניתנות-לשחזור באופן אוטומטי. במילים אחרות, הפרוייקט שלכם ישאר בגרסה 0.8.5 עד שתחליטו, הודות לקובץ \*Cargo.lock\*, לעדכנו באופן מפורש. בשל החשיבות של הקובץ \*Cargo.lock\* לשחזור בניות, מוסיפים אותו, בדר"כ, לבקרת קוד יחד עם כל שאר הקוד בפרוייקט.

#### עדכון מכולה לגרסה חדשה

כאשר \*כן\* מעוניינים לעדכן מכולה, קארגו מספק את הפקודה `update` אשר מתעלמת מהקובץ \*Cargo.lock\* ולכן כן מבצעת את תהליך פענוח כל הגרסאות הדרושות כדי להתאים לנתונים בקובץ \*Cargo.toml\*. . קארגו אז יכתוב גרסאות אלה לקובץ \*Cargo.lock\*. אחרת, כברירת מחדל, קארגו יחפש גרסאות מאוחרות יותר מ-0.8.5 ומוקדמות מ-0.9.0. אם למכולה `rand` הופצו גרסאות 0.8.6 וגם 0.9.0, אז תראו את המידע הבא לכשתריצו `cargo update`:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-02/

cargo update

assuming there is a new 0.8.x version of rand; otherwise use another update

as a guide to creating the hypothetical output shown here -->

```console

$ cargo update

Updating crates.io index

Updating rand v0.8.5 -> v0.8.6

```

קארגו מתעלם מהפצה 0.9.0. בנקודה זו גם תשימו לב לשינוי בקובץ \*Cargo.lock\* שמציין שהגרסה של מכולת ה-`rand` בהם אתם משתמשים עכשיו היא 0.8.6. בכדי להשתמש בגרסה 0.9.0 של `rand`, או בכל גרסה מסדרת 0.9.\*x\*, תצטרכו לעדכן את הקובץ \*Cargo.toml\* כדי שיראה כך:

```toml

[dependencies]

rand = "0.9.0"

```

בפעם הבאה שתריצו `cargo build`, קארגו יעדכן את רשימת המכולות הזמינות ויבצע הערכה-מחדש לדרישות של `rand` בהתבסס על הגרסה החדשה שציינתם.

יש עוד הרבה מה לאמר אודות [Cargo][doccargo]<!-- ignore --> [והכלים הבנויים סביבו][doccratesio]<!-- ignore -->, ועל כך נרחיב את הדיון בפרק 14, אבל לעכשיו זה כל שעליכם לדעת. קארגו מקל מאוד על שימוש חוזר בספריות, ולכן ראסטיונרים יכולים לכתוב פרוייקטים קטנים יותר המורכבים יחדיו ממספר חבילות.

### ייצור מספר אקראי

הבה נתחיל להשתמש ב-`rand` כדי לייצר מספר לניחוש. הצעד הבא הוא לעדכן את \*src/main.rs\*, כפי שמופיע ברשימה 2-3.

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-03/src/main.rs:all}}

```

<span class="caption">רשימה 2-3: הוספת קוד לייצור מספר אקראי</span>

ראשית, נוסיף את השורה `use rand::Rng;`. התכונה `Rng` מגדירה מתודות בהן משתמשים כלים המייצרים מספרים אקראיים. על מנת שנוכל להשתמש במתודות אלה, תכונה זו חייבת להיכלל במרחב. פרק 10 ידון בתכונות אלו בהרחבה.

עכשיו, אנו מוסיפים שתי שורות באמצע. בשורה הראשונה אנו קוראים לפונקציה `rand::thread\_rng` שמספקת לנו את מנוע המספרים האקראיים בו נשתמש: מנוע מקומי לתהליכון החישוב הנוכחי של הריצה, אשר מקבל את הגרעין שלו ממערכת ההפעלה. לאחריה, אנו קוראים למתודה `gen\_range` על מנוע המספרים האקראיים. מתודה זו מוגדרת על ידי התכונה `Rng` שהכנסנו למתחם באמצעות הפקודה `use rand::Rng;`. המתודה `gen\_range` לוקחת ביטוי של טווח כארגומנט ומייצרת מספר אקראי בטווח זה. סוג ביטוי הטווח בו אנו משתמשים כאן הוא מהצורה `start..=end` והוא כולל את הקצה התחתון והעליון, כך שעלינו לציין `1..=100` בכדי לבקש מספר בין 1 ל-100.

> הערה: אי אפשר לדעת כך סתם באלו תכונות להשתמש ואיזה מתודות ופונקציות לקרוא ממכולה, לכן לכל מכולה יש תיעוד עם הוראות הפעלה. יכולת נחמדה נוספת של קארגו היא שהרצה של פקודת `cargo doc

--open` תבנה מקומית את התיעוד המסופק ע"י כל התלותות, ואף תפתח אותו בדפדפן שלכם. אם אתם מעוניינים ביכולות אחרות של המכולה `rand`, למשל, הריצו `cargo doc --open` ובחרו את `rand` בסרגל הצידי משמאל.

השורה החדשה השניה מדפיסה את המספר הסודי. נוח לעשות זאת בעודנו בשלב הפיתוח של התוכנית, בכדי להקל על תהליך הבדיקה. מן הסתם, אנו נמחק שורה זו בגרסה הסופית. לא יהיה זה משחק מאתגר במיוחד אם התכנית מדפיסה את התשובה מיד בתחילתה!

נסו להריץ את התוכנית מספר פעמים:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-03/

cargo run

4

cargo run

5

-->

```console

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 2.53s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 7

Please input your guess.

4

You guessed: 4

$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in 0.02s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 83

Please input your guess.

5

You guessed: 5

```

צפוי שתקבלו מספרים שונים מאלו, וכולם צריכים להיות בין 1 ל-100. עבודה טובה!

## השוואת הניחוש למספר הסודי

עכשיו שיש לנו קלט מהמשתמש ומספר אקראי אנחנו יכולים להשוות אותם. צעד זה מוצג ברשימה 2-4. שימו לב שהקוד הזה לא עובר קומפילציה, כפי שנסביר מייד.

<span class="filename">Filename: src/main.rs</span>

```rust,ignore,does\_not\_compile

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-04/src/main.rs:here}}

```

<span class="caption">רשימה 2-4: טיפול בערכים המוחזרים האפשרים כשמשווים שני מספרים</span>

קודם אנו מוסיפים עוד פקודת `use` המכניסה את הטיפוס `std::cmp::Ordering` מהספריה הסטנדרטית למתחם. הטיפוס `Ordering` הוא מבחר נוסף ויש בו את הווריאנטים `Less`, `Greater` ו-`Equal`. אלו שלושת התוצאות האפשריות כאשר משווים שני ערכים.

לאחר מכן נוסיף בתחתית הקוד חמש שורות המשתמשות בטיפוס `Ordering`. המתודה `cmp` משווה בין שני ערכים וניתן לקרוא לה על כל דבר בר-השוואה. היא מקבלת הפנייה לדבר מולו אתם רוצים להשוות: כאן היא משווה את `guess` ל- `secret\_number`. אז היא מחזירה ווריאנט של המבחר `Ordering`, שהכנסנו למתחם באמצעות הפקודה `use`. אנחנו משתמשים בביטוי [`match`][match]<!-- ignore --> על מנת להחליט מה לעשות בצעד הבא בהתבסס על הווריאנט של `Ordering` שהוחזר מהקריאה ל-`cmp` עם הערכים ב-`guess` וב-`secret\_number`.

ביטוי `match` מורכב מכמה \*הסתעפויות\* (arms). הסתעפות מורכבת מדפוס, או תבנית, מולה ישווה הקוד את הביטוי, ומהקוד להרצה במידה והערך שהועבר ל- `match` תואם לדפוס. ראסט לוקחת את הערך שהועבר ל- `match` ומתבוננת בדפוס של ההסתעפויות לפי הסדר בו הן מופיעות בקוד. דפוסים ומבנה ה-`match` הם יכולות עצמתיות של ראסט: הם מאפשרים לבטא מגוון של סיטואציות בהן הקוד שלכם יכול להיתקל ומבטיחים שכולן יטופלו. עוד על תכונות אלה ידובר בפרק 6 ובפרק 18, בהתאמה.

הבה נעבור בפירוט על דוגמא עם ביטוי ה- `match` בו אנו משתמשים כאן. נניח שהמשתמש ניחש את המספר 50 ושהמספר הסודי האקראי בזמן ההרצה הוא 38.

כשאר הקוד משווה את 50 ל-38, המתודה `cmp` תחזיר `Ordering::Greater` כיוון ש-50 יותר גדול מ-38. ביטוי ה- `match` לוקח את הערך `Ordering::Greater` ומתחיל לבדוק את הדפוס בכל הסתעפות . הוא בודק את הדפוס בהסתעפות הראשונה, `Ordering::Less`, ורואה שהערך `Ordering::Greater` אינו תואם את `Ordering::Less`, . לכן הוא מתעלם מהקוד שתחת הסתעפות זו, ועובר להסתעפות הבאה. הדפוס של ההסתעפות הבאה הוא `Ordering::Greater`, וזה \*כן\* תואם את `Ordering::Greater`! הקוד הכלול בזרוע זו יבוצע וידפיס `Too big!` למסך. ביטוי ה-`match` מסתיים מייד לאחר התאמה מוצלחת, ולכן, במקרה זה, הוא כלל לא יתבונן בזרוע האחרונה.

אבל, הקוד ברשימה 2-4 לא יעבור קומפילציה. הבה ננסה זאת:

<!--

The error numbers in this output should be that of the code \*\*WITHOUT\*\* the

anchor or snip comments

-->

```console

{{#include ../listings/ch02-guessing-game-tutorial/listing-02-04/output.txt}}

```

בקצרה, השגיאה מציינת שישנם בקוד \*טיפוסים לא-מותאמים\* (mismatched types). לראסט יש מערכת טיפוסים סטטית חזקה. בד-בבד, ראסט גם מבצעת הסקת טיפוס. כשכתבנו `let mut guess = String::new()`, ראסט הצליחה להסיק ש- `guess` צריך להיות מטיפוס `String` והיא לא הכריחה אותנו לכתוב זאת מפורשות. המשתנה `secret\_number`, לעומת זאת, הוא מטיפוס מספר. יש כמה טיפוסי מספר של ראסט שיכולים לאחסן מספר בין 1 ל-100: `i32`, מספר בן 32-ביטים, `u32`, מספר לא מסומן בן 32-ביטים, `i64`, מספר בן 64-ביטים, ואחרים. ללא ציון מפורש, ברירת המחדל של ראסט היא `i32`. לכן, בהעדר מידע נוסף אודות הטיפוס בצורה שתאפשר לראסט להסיק טיפוס אחר, הטיפוס של `secret\_number` הוא `i32`. הסיבה לשגיאה היא שראסט לא יכולה להשוות בין טיפוסים של מחרוזת ומספר.

בסופו של דבר, אנו רוצים להמיר את המחרוזת שהתכנית קראה כקלט מטיפוס מחרוזת לטיפוס מספרי מתאים כדי שנוכל להשוות אותו נומרית למספר הסודי. אנו עושים זאת כך הוספת השורה הבאה לגוף הפונקציה `main`:

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/no-listing-03-convert-string-to-number/src/main.rs:here}}

```

השורה היא:

```rust,ignore

let guess: u32 = guess.trim().parse().expect("Please type a number!");

```

כך יצרנו משתנה בשם `guess`. אבל רגע אחד, האם אין כבר בתכנית משתנה בשם `guess`? כן, אבל ראסט מאפשרת לנו להאפיל (to shadow) על הערך הקודם של `guess` באמצעות ערך חדש. האפלה מאפשרת לבצע שימוש חוזר בשם המשתנה `guess` במקום להידרש ליצור שני משתנים, כמו `guess\_str` ו-`guess` למשל. אנו נדון בכך ביתר פירוט [בפרק 3][shadowing].<!-- ignore --> לבינתיים, דעו שבתכונה זו משתמשים פעמים רבות כאשר רוצים להפוך משתנה מטיפוס אחד לאחר.

אנו קושרים משתנה חדש זה לביטוי `guess.trim().parse()`. המשתנה `guess` בביטוי מתייחס למשתנה `guess` המקורי שמכיל את הקלט כמחרוזת. המתודה `trim` על מופע של `String` מקצצת את כל סימני הריווח בתחילתה וסופה של המחרוזת. אנו חייבים לעשות זאת כדי להיות מסוגלים להשוות את המחרוזת למשתנה מהטיפוס `u32`, שיכול להכיל רק ערכים מספריים. המשתמש חייב ללחוץ <span class="keystroke">enter</span> כדי לספק את `read\_line` ולהקליד את הניחוש, ודבר זה מוסיף תו שורה-חדשה למחרוזת. למשל, אם המשתמש מקליד <span class="keystroke">5</span> ואז לוחץ <span

class="keystroke">enter</span>, המחרוזת ב-`guess` תראה כך: `5\n`. התו `\n` מייצג "שורה-חדשה" (על Windows, לחיצה על <span

class="keystroke">enter</span> גורמת ל-carriage return בנוסף לשורה-חדשה.`\r\n`.) המתודה `trim` מקצצת גם את `\n` וגם את `\r\n`, והתוצאה היא `5`.

בשימוש על מחרוזות, [המתודה `parse` ][parse]<!-- ignore --> ממירה מחרוזת לטיפוס אחר. במקרה שלנו אנו משתמשים בה כדי להמיר ממחרוזת למספר. עלינו לומר לראסט בדיוק איזה טיפוס מספר אנו רוצים, ואנו עושים זאת באמצעות `let guess: u32`. סימן הנקודותיים (`:`) אחרי `guess` אומר לראסט שאנחנו מבארים במפורש את הטיפוס של המשתנה. לראסט יש כמה טיפוסים מובנים, והטיפוס-המובנה `u32` בו אנו משתמשים כאן הוא מספר שלם לא-מסומן, בן 32-ביטים. זו ברירת מחדל טובה עבור מספר חיובי קטן. אודות טיפוסי מספרים אחרים תלמדו [בפרק 3][integers]<!-- ignore -->.

בנוסף, השימוש ב-`u32` בתכנית פשוטה זו וההשוואה עם `secret\_number` גורמים לראסט להסיק ש-`secret\_number` אמור להיות `u32` גם כן. על כן, ההשוואה תתבצע כעת בין שני ערכים מאותו טיפוס!

יש להדגיש שהמתודה `parse` מסוגלת לפעול רק על תווים שהמסוגלים, מעצם טיבם, לעבור המרה למספרים, ולכן יכולה בקלות להעלות שגיאות. אם, למשל, המחרוזת כוללת `A👍%`,, אזי אין דרך להמיר תווים אלה למספר. כיוון שהמתודה יכולה להיכשל, `parse` מחזירה ערך מטיפוס `Result`, בדומה לדרך פעולה המתודה `read\_line` (כפי שדובר לעיל בסעיף [“Handling Potential Failure with `Result`”](#handling-potential-failure-with-result)<!-- ignore-->). אנו נטפל ב- `Result` באותה דרך, שוב ע"י שימוש במתודה `expect`. אם `parse` תחזיר את הווריאנט `Err` של `Result` בגלל שהיא אינה מסוגלת ליצור מספר מהמחרוזת, אז הקריאה ל- `expect` תגרום למשחק לקרוס ולהדפיס את ההודעה המצוינת בקוד. לעומת זאת, אם `parse` אכן מסוגלת להמיר בהצלחה את המחרוזת למספר, היא תחזיר את הווריאנט `Ok` של `Result`, והקריאה ל- `expect` תחזיר את המספר המבוקש מתוך הערך `Ok`.

כעת, הבה נריץ את התוכנית:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/no-listing-03-convert-string-to-number/

cargo run

76

-->

```console

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 0.43s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 58

Please input your guess.

76

You guessed: 76

Too big!

```

יפה! למרות שתווי רווח הוספו לפני הניחוש עצמו, התכנית עדיין הצליחה לפענח שהמשתמש ניחש 76. הריצו את התכנית מספר פעמים על מנת לוודא את ההתנהגות השונה על קלטים מסוגים שונים: נחשו את המספר במדוייק, נחשו מספר גבוה מדי, ונחשו מספר נמוך מדי.

רוב המשחק פועל היטב, אבל המשתמש יכול לבצע רק ניחוש אחד. הבה נשנה זאת ע"י הוספת לולאה!

## ריבוי ניחושים באמצעות לולאה

מילת המפתח `loop` יוצרת לולאה אינסופית. נוסיף לולאה על מנת לאפשר למשתמשים סיכויים נוספים לנחש את המספר:

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/no-listing-04-looping/src/main.rs:here}}

```

כפי שאתם רואים, הכנסנו ללולאה את כל הקוד, החל מן הבקשה להקליד ניחוש. ודאו להזיח כל אחת מהשורות שבתוך הלולאה ארבעה רווחים קדימה, והריצו את התוכנית. כעת, התוכנית תבקש ניחוש נוסף שוב ושוב, לנצח. זאת, כמובן, בעיה חדשה. נראה שאין למשתמש דרך לצאת מהתכנית!

המשתמש תמיד יכול להכריח את התוכנית לעצור ע"י שימוש בקיצור המקלדת <span class="keystroke">ctrl-c</span>. אבל יש דרך אחרת לצאת מממפלצת זו שאינה יודעת-שובע, כפי שהוסבר בדיון על `parse` ב-[“Comparing the Guess to the Secret Number”](#comparing-the-guess-to-the-secret-number)<!--

ignore -->: אם המשתמש יקליד דבר מה שאינו מספר, התכנית תקרוס. ניתן לנצל זאת כדי לאפשר למשתמש לצאת, כך:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/no-listing-04-looping/

cargo run

(too small guess)

(too big guess)

(correct guess)

quit

-->

```console

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 1.50s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 59

Please input your guess.

45

You guessed: 45

Too small!

Please input your guess.

60

You guessed: 60

Too big!

Please input your guess.

59

You guessed: 59

You win!

Please input your guess.

quit

thread 'main' panicked at 'Please type a number!: ParseIntError { kind: InvalidDigit }', src/main.rs:28:47

note: run with `RUST\_BACKTRACE=1` environment variable to display a backtrace

```

הקלדת `quit` תגרום למשחק להסתיים, אבל, כפי שתוכלו לבדוק, כך יקרה עם הקלדת כל קלט שאינו מספר. ניתן לומר, לכל הפחות, שזהו מצב שאינו אופטימלי; אנו רוצים שהמשחק יעצור ברגע שהמספר הנכון מוקלד כניחוש.

### יציאה לאחר ניחוש מדוייק

הבה נתכנת את המשחק לעצור כשהמשתמש מנצח ע"י, הוספת פקודת `break`:

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/no-listing-05-quitting/src/main.rs:here}}

```

הוספת הפקודה `break` שורה לאחר `You win!` גורמת לתוכנית לצאת מהלולאה לאחר שהמשתמש ניחש את המספר במדוייק. יציאה מהלולאה משמעה גם יציאה מהתכנית, זאת מכיוון שהלולאה היא גם החלק האחרון של `main`.

### טיפול בקלט שגוי

על מנת להוסיף ולשכלל את התנהגות המשחק, במקום לתת לתכנית לקרוס במידה והמשתמש מקליד קלט שאינו מספר, הבה נגרום למשחק להתעלם מקלט שאינו מספר כדי שהמשתמש יוכל להמשיך לנחש. נוכל לעשות זאת ע"י שינוי השורה שבה `guess` מומר מטיפוס `String` ל-`u32`, כפי שמוצג ברשימה 2-5.

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-05/src/main.rs:here}}

```

<span class="caption">רשימה 2-5: החלפת התרסקות התכנית בהתעלמות מניחוש שאינו מספר ובקשה מהמשתמש לניחוש נוסף</span>

אנו עוברים מקריאה ל- `expect` לשימוש בביטוי `match`. כך למעשה אנו עוברים מהתרסקות התכנית בעקבות שגיאה, לטיפול נאות בשגיאה. זכרו ש-`parse` מחזירה ערך מטיפוס `Result` וש- `Result` הוא בחירה עם הווריאנטים `Ok` ו-`Err`. אנחנו משתמשים כאן בביטוי `match`, כפי שעשינו בתוצאת ה- `Ordering` של המתודה `cmp`.

אם `parse` מצליחה להמיר בהצלחה את המחרוזת למספר, אז היא תחזיר ערך `Ok` שמכיל את המספר המומר. ערך ה-`Ok` הזה יותאם לדפוס של ההסתעפות הראשונה וביטוי ה- `match` פשוט יחזיר את הערך `num` אשר `parse` הפיקה ושמה בתוך ערך ה-`Ok`. מספר זה ימצא את דרכו בדיוק למקום בו אנו רוצים אותו, דהיינו המשתנה החדש `guess` שאנו יוצרים.

במידה ו-`parse` \*אינה\* מצליחה להפוך את המחרוזת למספר, היא תחזיר ערך `Err` שמכיל מידע נוסף אודות הבעיה. ערך ה- `Err` לא מותאם לדפוס ה- `Ok(num)` בהסתעפות הראשונה של ה- `match`, אבל הוא כן מותאם לדפוס `Err(\_)` אשר בהסתעפות השניה. המקף התחתון, `\_`, מתפקד כערך תופס כללי; בדוגמא זו אנו מציינים שאנחנו רוצים להתאים את כל ערכי ה-`Err`, ללא תלות במידע הנוסף שבתוכם. בצורה זו התכנית תבצע את הקוד שבהסתעפות השניה, ז"א `continue`, אשר מורה לתכנית לעבור לאיטרציה, קרי לסבב הבא של הלולאה, ולבקש מהמשתמש ניחוש נוסף. כך למעשה התכנית מתעלמת מכל השגיאות ש- `parse` עלולה להיתקל בהן!

עכשיו כל האלמנטים בתכנית אמורים לעבוד כצפוי. הבה נבדוק זאת:

<!-- manual-regeneration

cd listings/ch02-guessing-game-tutorial/listing-02-05/

cargo run

(too small guess)

(too big guess)

foo

(correct guess)

-->

```console

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished dev [unoptimized + debuginfo] target(s) in 4.45s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 61

Please input your guess.

10

You guessed: 10

Too small!

Please input your guess.

99

You guessed: 99

Too big!

Please input your guess.

foo

Please input your guess.

61

You guessed: 61

You win!

```

מעולה! עוד שינוי קל אחד ונסיים עם משחק הניחוש. שימו לב שהתכנית עדיין מדפיסה את המספר הסודי. זה היה נוח בזמן הפיתוח, אבל זה מקלקל את המשחק. הבה נסיר את הפקודה `println!` שמדפיסה את המספר הסודי. רשימה 2-6 מציגה את הקוד הסופי.

<span class="filename">Filename: src/main.rs</span>

```rust,ignore

{{#rustdoc\_include ../listings/ch02-guessing-game-tutorial/listing-02-06/src/main.rs}}

```

<span class="caption">רשימה 2-6: הקוד המלא של משחק הניחוש</span>

בנקודה זו בניתם בהצלחה את משחק הניחוש. כל הכבוד!

## סיכום

פרוייקט זה היווה דרך מעשית להפגיש אתכם עם מספר עקרונות של שפת התכנות ראסט: `let`, `match`, פונקציות, שימוש במכולות חיצוניות, ועוד. בפרקים הבאים תלמדו פרטים נוספים אודות מושגים אלה. פרק 3 מכסה מושגים הקיימים במרבית שפות התכנות כגון משתנים, טיפוסי נתונים, ופונקציות, ויתאר כיצד להשתמש בהם בראסט. פרק 4 סוקר את מושג הבעלות, יכולת העושה את ראסט שונה משפות אחרות. פרק 5 עוסק בתחביר של מבנים (structs) ומתודות (methods), ופרק 6 מסביר איך עובדים מבחרים (enums).

[prelude]: ../std/prelude/index.html

[variables-and-mutability]: ch03-01-variables-and-mutability.html#variables-and-mutability

[comments]: ch03-04-comments.html

[string]: ../std/string/struct.String.html

[iostdin]: ../std/io/struct.Stdin.html

[read\_line]: ../std/io/struct.Stdin.html#method.read\_line

[result]: ../std/result/enum.Result.html

[enums]: ch06-00-enums.html

[enums]: ch06-00-enums.html

[expect]: ../std/result/enum.Result.html#method.expect

[recover]: ch09-02-recoverable-errors-with-result.html

[randcrate]: https://crates.io/crates/rand

[semver]: http://semver.org

[cratesio]: https://crates.io/

[doccargo]: http://doc.crates.io

[doccratesio]: http://doc.crates.io/crates-io.html

[match]: ch06-02-match.html

[shadowing]: ch03-01-variables-and-mutability.html#shadowing

[parse]: ../std/primitive.str.html#method.parse

[integers]: ch03-02-data-types.html#integer-types